

Jan Gulliksen Morton Borup Harning  
Philippe Palanque Gerrit C. van der Veer  
Janet Wesson (Eds.)

LNCSE 4940

# Engineering Interactive Systems

EIS 2007 Joint Working Conferences  
EHCI 2007, DSV-IS 2007, HCSE 2007  
Salamanca, Spain, March 2007, Selected Papers



ifip

 Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Alfred Kobsa

*University of California, Irvine, CA, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*University of Dortmund, Germany*

Madhu Sudan

*Massachusetts Institute of Technology, MA, USA*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Gerhard Weikum

*Max-Planck Institute of Computer Science, Saarbruecken, Germany*

Jan Gulliksen Morton Borup Harning  
Philippe Palanque Gerrit C. van der Veer  
Janet Wesson (Eds.)

# Engineering Interactive Systems

EIS 2007 Joint Working Conferences  
EHCI 2007, DSV-IS 2007, HCSE 2007  
Salamanca, Spain, March 22-24, 2007  
Selected Papers

Volume Editors

Jan Gulliksen  
Uppsala University, Uppsala, Sweden  
E-mail: jan.gulliksen@it.uu.se

Morton Borup Harning  
Priway ApS, Lyngby, Denmark  
E-mail: harning@se-hci.org

Philippe Palanque  
Institute of Research in Informatics of Toulouse (IRIT)  
University Paul Sabatier, Toulouse, France  
E-mail: palanque@irit.fr

Gerrit C. van der Veer  
School of Computer Science  
Open Universiteit Nederland  
Heerlen, The Netherlands  
E-mail: gerrit.vanderVeer@ou.nl

Janet Wesson  
Nelson Mandela Metropolitan University  
Port Elizabeth, South Africa  
E-mail: janet.wesson@nmmu.ac.za

Library of Congress Control Number: Applied for

CR Subject Classification (1998): H.5.2, H.5, D.2.2, D.3, F.3, I.6, K.6

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743  
ISBN-10 3-540-92697-6 Springer Berlin Heidelberg New York  
ISBN-13 978-3-540-92697-9 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

springer.com

© IFIP International Federation for Information Processing 2008  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India  
Printed on acid-free paper SPIN: 12592340 06/3180 5 4 3 2 1 0



# Preface

Engineering Interactive Systems 2007 is an IFIP working conference that brings together researchers and practitioners interested in strengthening the scientific foundations of user interface design, examining the relationship between software engineering (SE) and human-computer interaction (HCI) and on how user-centered design (UCD) could be strengthened as an essential part of the software engineering process.

Engineering Interactive Systems 2007 was created by merging three conferences:

- HCSE 2007 – Human-Centered Software Engineering held for the first time. The HCSE Working Conference is a multidisciplinary conference entirely dedicated to advancing the basic science and theory of human-centered software systems engineering. It is organized by IFIP WG 13.2 on Methodologies for User-Centered Systems Design.
- EHCI 2007 – Engineering Human Computer Interaction was held for the tenth time. EHCI aims to investigate the nature, concepts, and construction of user interfaces for software systems. It is organized by IFIP WG 13.4/2.7 on User Interface Engineering.
- DSV-IS 2007 – Design, Specification and Verification of Interactive Systems was held for the 13th time. DSV-IS provides a forum where researchers working on model-based techniques and tools for the design and development of interactive systems can come together with practitioners and with those working on HCI models and theories.

Almost half of the software in systems being developed today and 37%–50% of the efforts throughout the software lifecycle are related to the system's user interface. For this reason problems and methods from the field of HCI affect the overall process of SE tremendously, and vice versa. Yet despite these powerful reasons to practice and apply effective SE and HCI methods, major gaps of understanding still exist, both between the suggested practice, provided through methods, tools and models, and how software is actually being developed in industry (between theory and practice), and between the best practices of each of the fields.

The standard curricula for each field make little (if any) reference to the other field and certainly do not teach how to interact with the other field. There are major gaps of communication between the HCI and SE fields: the architectures, processes, methods, and vocabulary being used in each community are often foreign to the other community. As a result, product quality is not as high as it could be, and otherwise possibly avoidable re-work is frequently necessary.

SE technology used in building tomorrow's interactive systems must place a greater emphasis on designing usable systems that meet the needs of the users. HCI, SE, computer science, psychology as well as many other researchers from other related disciplines have developed, sometimes independently from the engineering lifecycle, various tools and techniques for achieving these goals. Unfortunately, even if big

software development organizations as well as a few enlightened practitioners have recognized their importance and/or have considered them when developing their products, these techniques are still relatively unknown, under used, difficult to master, and most fundamentally they are not well integrated in SE practices.

Despite all the knowledge on usability and user-centered systems design, most computer systems today are developed with a minimum of user involvement hence resulting in systems that do not fit the users' needs and expectations sufficiently. Similarly the scientific fields of SE (dealing with the processes by which systems are being developed) and HCI (dealing with the user's use of the system) rarely meet. There is a growing awareness that these two scientific fields need to meet on equal terms to discuss and resolve the potential conflicts in the approaches proposed by the two perspectives. This is the main reasons for our efforts to arrange a venue for these different fields to meet, interact, and share our knowledge and experiences, to increase the focus on users and usability in the SE processes, methods and tools, and to provide a deepened understanding among HCI researchers and practitioners of the emerging need to relate to the processes and practices of SE professionals.

The list of topics for the conference was compiled from the list of topics traditionally included for each of the three conferences, but with the added aim of creating a list of topics that would foster a fruitful discussion helping to bring SE issues and user interface design concerns as well UCD issues closer together.

#### Integration of SE and UCD

- Towards a theory for human-centered systems engineering
- Incorporating guidelines and principles for designing usable products into the development processes
- Usability through the requirements specification
- Representations for design in the development process
- Working with usability with commercial development processes such as Rational Unified Process (RUP), Dynamic Systems Development Method (DSDM), eXtreme Programming (XP), Agile processes, etc.
- Social and organizational aspects of software development in a lifecycle perspective

#### SE aspects of user interfaces

- Software architecture
- Formal methods in HCI
- HCI models and model-driven engineering
- Impact of distribution on user interfaces
- Portability, consistency, integration
- Development processes
- Case studies

#### User interface tools and techniques

- Adaptive and customizable systems
- Interfaces for restricted environments

- Interfaces for multiple devices
- Web-based systems
- Evaluation of user interfaces: technologies and tools

#### Engineering aspects of innovative user interfaces

- Interfaces for mobile devices
- Wearable computing
- New interface technologies
- Information visualization and navigation
- Multimodal user interfaces
- Interfaces for groupware
- Virtual reality, augmented reality
- Games

A total of 37 papers were selected for presentation forming sessions on analysis and verification, task and engineering models, design for use in context, architecture, models for reasoning, and finally patterns and guidelines.

Following the EHCI working conference tradition, the proceedings include transcripts of paper discussions.

Jan Gulliksen  
Morten Borup Harning

# Table of Contents

Performance Analysis of an Adaptive User Interface System Based on Mobile Agents . . . . .	1
<i>Nikola Mitrović, Jose A. Royo, and Eduardo Mena</i>	
Combining Human Error Verification and Timing Analysis . . . . .	18
<i>Rimvydas Rukšėnas, Paul Curzon, Ann Blandford, and Jonathan Back</i>	
Formal Testing of Multimodal Interactive Systems . . . . .	36
<i>Jullien Bouchet, Laya Madani, Laurence Nigay, Catherine Oriat, and Ioannis Parissis</i>	
Knowledge Representation Environments: An Investigation of the CASSMs between Creators, Composers and Consumers . . . . .	53
<i>Ann Blandford, Thomas R.G. Green, Iain Connell, and Tony Rose</i>	
Consistency between Task Models and Use Cases . . . . .	71
<i>Daniel Sinnig, Patrice Chalin, and Ferhat Khendek</i>	
Task-Based Design and Runtime Support for Multimodal User Interface Distribution . . . . .	89
<i>Tim Clerckx, Chris Vandervelpen, and Karin Coninx</i>	
A Comprehensive Model of Usability . . . . .	106
<i>Sebastian Winter, Stefan Wagner, and Florian Deissenboeck</i>	
Suitability of Software Engineering Models for the Production of Usable Software . . . . .	123
<i>Karsten Nebe and Dirk Zimmermann</i>	
A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces . . . . .	140
<i>Jean-Sébastien Sottet, Gaëlle Calvary, Joëlle Coutaz, and Jean-Marie Favre</i>	
Model-Driven Prototyping for Corporate Software Specification . . . . .	158
<i>Thomas Memmel, Carsten Bock, and Harald Reiterer</i>	
Getting SW Engineers on Board: Task Modelling with Activity Diagrams . . . . .	175
<i>Jens Brüning, Anke Dittmar, Peter Forbrig, and Daniel Reichart</i>	
Considering Context and Users in Interactive Systems Analysis . . . . .	193
<i>José Creissac Campos and Michael D. Harrison</i>	

XSED – XML-Based Description of Status–Event Components and Systems .....	210
<i>Alan Dix, Jair Leite, and Adrian Friday</i>	
Identifying Phenotypes and Genotypes: A Case Study Evaluating an In-Car Navigation System.....	227
<i>Georgios Papatzanis, Paul Curzon, and Ann Blandford</i>	
Factoring User Experience into the Design of Ambient and Mobile Systems .....	243
<i>Michael D. Harrison, Christian Kray, Zhiyu Sun, and Huiqiu Zhang</i>	
Visualisation of Personal Communication Patterns Using Mobile Phones .....	260
<i>Bradley van Tonder and Janet Wesson</i>	
Integration of Distributed User Input to Extend Interaction Possibilities with Local Applications.....	275
<i>Kay Kadner and Stephan Mueller</i>	
Reverse Engineering Cross-Modal User Interfaces for Ubiquitous Environments .....	285
<i>Renata Bandelloni, Fabio Paternò, and Carmen Santoro</i>	
Intelligent Support for End-User Web Interface Customization .....	303
<i>José A. Macías and Fabio Paternò</i>	
Improving Modularity of Interactive Software with the MDPC Architecture.....	321
<i>Stéphane Conversy, Eric Barboni, David Navarre, and Philippe Palanque</i>	
Toward Quality-Centered Design of Groupware Architectures .....	339
<i>James Wu and T.C. Nicholas Graham</i>	
Programs = Data + Algorithms + Architecture: Consequences for Interactive Software Engineering .....	356
<i>Stéphane Chatty</i>	
Towards an Extended Model of User Interface Adaptation: The ISATINE Framework .....	374
<i>Víctor López-Jaquero, Jean Vanderdonckt, Francisco Montero, and Pascual González</i>	
Towards a Universal Toolkit Model for Structures .....	393
<i>Prasun Dewan</i>	
Exploring Human Factors in Formal Diagram Usage.....	413
<i>Andrew Fish, Babak Khazaei, and Chris Roast</i>	

‘Aware of What?’ A Formal Model of Awareness Systems That Extends the Focus-Nimbus Model . . . . .	429
<i>Georgios Metaxas and Panos Markopoulos</i>	
Service-Interaction Descriptions: Augmenting Services with User Interface Models . . . . .	447
<i>Jo Vermeulen, Yves Vandriessche, Tim Clerckx, Kris Luyten, and Karin Coninx</i>	
A Design-Oriented Information-Flow Refinement of the ASUR Interaction Model . . . . .	465
<i>Emmanuel Dubois and Philip Gray</i>	
On the Process of Software Design: Sources of Complexity and Reasons for Muddling through . . . . .	483
<i>Morten Hertzum</i>	
Applying Graph Theory to Interaction Design . . . . .	501
<i>Harold Thimbleby and Jeremy Gow</i>	
Mathematical Mathematical User Interfaces . . . . .	520
<i>Harold Thimbleby and Will Thimbleby</i>	
Coupling Interaction Resources in Ambient Spaces: There Is More Than Meets the Eye! . . . . .	537
<i>Nicolas Barralon and Joëlle Coutaz</i>	
Building and Evaluating a Pattern Collection for the Domain of Workflow Modeling Tools . . . . .	555
<i>Kirstin Kohler and Daniel Kerkow</i>	
Do We Practise What We Preach in Formulating Our Design and Development Methods? . . . . .	567
<i>Paula Kotzé and Karen Renaud</i>	
Engaging Patterns: Challenges and Means Shown by an Example . . . . .	586
<i>Sabine Niebuhr, Kirstin Kohler, and Christian Graf</i>	
Organizing User Interface Patterns for e-Government Applications . . . . .	601
<i>Florence Pontico, Marco Winckler, and Quentin Limbourg</i>	
Including Heterogeneous Web Accessibility Guidelines in the Development Process . . . . .	620
<i>Myriam Arrue, Markel Vigo, and Julio Abascal</i>	
<b>Author Index</b> . . . . .	639

# Performance Analysis of an Adaptive User Interface System Based on Mobile Agents

Nikola Mitrović, Jose A. Royo, and Eduardo Mena

IIS Department, University of Zaragoza, Maria de Luna 1, 50018 Zaragoza, Spain  
mitrovic@prometeo.cps.unizar.es, joalroyo@unizar.es,  
emena@unizar.es

<http://www.cps.unizar.es/~mitrovic>

<http://www.cps.unizar.es/~jaroyo>

<http://www.cps.unizar.es/~mena>

**Abstract.** Adapting graphical user interfaces for various user devices is one of the most interesting topics in today's mobile computation. In this paper we present a system based on mobile agents that transparently adapts user interface specifications to the user device' capabilities and monitors user interaction. Specialized agents manage GUI specification according to the specific context and user preferences. We show how the user behavior can be monitored at runtime in a transparent way and how learning methods are applied to anticipate future user actions and to adapt the user interface accordingly. The feasibility and performance of our approach are shown by applying our approach to a non-trivial application and by performing tests with real users.

## 1 Introduction

Adapting graphical user interfaces (GUIs) to different devices and user preferences is one of the most challenging questions in mobile computing and GUI design. User devices have different capabilities, from small text-based screens and limited processing capabilities to laptops and high-end workstations. Another important challenge is to adapt user interfaces to user preferences, context, and GUI actions to be performed. Some of these parameters, user preferences, depends on the specific user while others, user's context or actions, do not. However all these parameters vary over time which makes them more difficult to manage.

Mobile environments are particularly challenging: mobile devices require applications with small footprints, written for specific proprietary platform that can execute on devices with very limited capabilities and resources. Mobile devices connect to other devices by using wireless networks which are more expensive<sup>1</sup>, unreliable, and slower, than their wired counterparts. Handling these problems is very difficult and applications are frequently written to accommodate specific devices and environment. Developing such applications requires a significant effort and expertise therefore portability across different user devices is a must.

---

<sup>1</sup> In the case of wireless WAN's.

To create user interfaces that can adapt to different devices and situations researchers use *abstract user interface definition languages* as a common ground. The abstract definition (usually specified in XML-based notation) is later rendered into a concrete (physical) user interface. Many abstract GUI definition languages exist: XUL [30], UIML [1], XIML [34], XForms [32], usiXML [31], just to name few. To adapt an abstract GUI definition to a real GUI researchers use client-server architectures [8], specialized tools to create separate GUIs for different platforms [22], and other take advantage of agent technology [18, 14].

Current GUI design methods lead to the re-design and re-implementation of applications for different devices. In addition, direct generation of user interfaces do not allow the system to monitor the user interaction which can be useful for adaptive systems. Our proposal to generate and manage adaptive GUIs is *ADUS (Adaptive User Interface System)* [18] which is based on an abstract graphical user interface definition language and a mobile agent architecture. Thus, while abstract a GUI definition language gives flexibility when describing a user interface, mobile agents allow flexible rendering of such a GUI definition and provide abstraction from other application layers (e.g., platform, connectivity problems, etc). Thus we adopt this approach as it enables the creation of flexible user interfaces that are able to adapt and move through the network. The ADUS system also enables adaptation to user preferences, context, and actions by monitoring and analyzing the user behavior [21]; such a collected knowledge is reused in future program executions to anticipate the user's actions.

In this paper we present the advantages of using ADUS in mobile computing applications, specifically, we show how learning from user actions on the generated GUI improves the performance of the system. For this task, we describe how ADUS has been used in a software retrieval service and the results of testing both versions (with and without ADUS) with real users.

The rest of this paper is as follows. In Section 2 we describe the main features of ADUS. Section 3 describes how ADUS learns from the user behavior and anticipates future user actions. In Section 4 we apply ADUS to a non-trivial sample application. Performance and usability evaluations of such a system are presented in Section 5. Section 6 gives an overview of the state of the art and the related work. Finally, conclusions and future work are presented in Section 7.

## 2 ADUS: Adaptive User Interface System

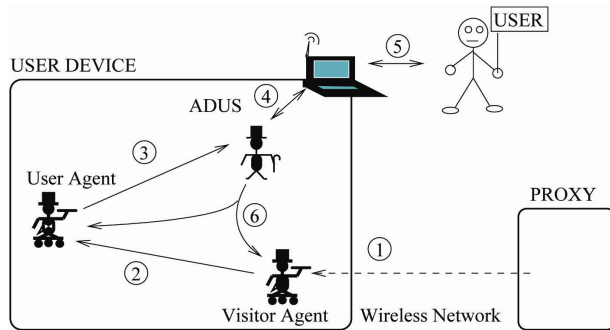
The ADaptive User interface System (ADUS) is an approach based on mobile agents that generates user interfaces adapted for different devices at run-time [18]. To provide this functionality, agents manage abstract descriptions of graphical user interfaces to be deployed. While abstract UI definition languages give flexibility in describing user interface, mobile agents allow flexible rendering of the UI definition and provide abstraction of other application layers (e.g., platform, connectivity problems, etc). We adopt this approach as it enables the creation of a flexible user interface capable of adapting and moving through the network. ADUS is part of the ANTARCTICA system [15] that provides users with different wireless data services aiming to enhance the capabilities of their mobile devices.



As GUI definition language we use XUL (*eXtensible User interface definition Language*) [30]. The GUI is specified in XUL and then transformed on the fly by mobile agents to a concrete user interface. Some of the GUI properties, such as window size, colors, and widgets used, are adapted on the fly. In addition, GUI sections and elements can be modified by mobile agents at the run time (see Section 3.4). The developed prototype can adapt such user interface descriptions to Java AWT, Java Swing, HTML, and WML clients, and supports limited plasticity [29]. GUI widgets are mapped to the concrete UI using CC/PP [4] and different transformation engines; further plasticity improvements are planned as future work.

The mobile agent technology eases automatic system adaptation to its execution environment. A mobile agent is a program that executes autonomously on a set of network hosts on behalf of an individual or organization [16, 17]. Mobile agents can bring computation wherever needed and minimize the network traffic, especially in wireless networks (expensive, slow, and unstable), without decreasing the performance of the system [33]. In our context, mobile agents are able to arrive at the user device and show their GUIs to the user in order to interact with her/him [18]. The deployment of mobile agents is automatic and has little performance overheads [33]. In our prototype we use the mobile agent system Voyager [9]; however any other mobile agent system could be used to implement our approach.

Our system uses indirect user interface generation [21] which is a method where several agents collaborate in order to transparently produce user interfaces adapted to users and devices. The main steps are (see Figure 1):



**Fig. 1.** Indirect generation of GUIs

1. A visitor agent arrives at the user device to interact with the user.
2. The visitor agent, instead of generating a GUI directly, generates a XUL [30] specification of the needed GUI, which is sent to the user agent who applies user-specific information to the GUI specification. This modification is based on user's preferences, context, or collected knowledge. For example, the user agent could use data from previous executions to automatically assign the values that were entered by the user to past visitor agents requesting the same information [15, 21]
3. The user agent creates an ADUS agent initialized with the new GUI specification.
4. The ADUS agent generates the GUI which will include the specific features for that user and for that user device.
5. The user interacts with the GUI.

6. The ADUS agent handles and propagates the GUI events to 1) the visitor agent, who should react to such events, and 2) the user agent, which in this way monitors and learns from such user actions.

The additional benefit of such a transparent user interface generation is the simplicity of software development – using our approach only one version of user interface and application code is developed (in XUL) but the corresponding GUIs are automatically generated for very different user devices without user or software developer intervention.

### 3 User Interaction Monitoring and Application: The Learning Process

One of the key features of our prototype is the ability to monitor and collect user interaction information at the run time [21]. The prototype monitors both GUI interaction and interaction between the visitor agent and the user using the indirect user interface generation model, as explained before. Such data can be used to examine user's behavior and apply the collected knowledge on the subsequently generated user interfaces. The monitoring mechanism does not depend on the type of application or platform. It is important to notice that, as the monitoring mechanism is based on mobile agents, it is distributed, mobile, and can be extended with security frameworks for mobile agents [20].

Our prototype uses data mining techniques to anticipate user's actions. In addition, our prototype utilizes task models as training data for data mining techniques. In the following paragraphs we present the techniques used in our prototype.

#### 3.1 Predicting User Behavior

Predicting the user behavior is a difficult task: a common methodology to predict users' behavior is predictive statistical models. These models are based on linear models, TFIDF (Term Frequency Inverse Document Frequency), Markov Models, Neural Methods, Classification, Rule Induction, or Bayesian Networks [35]. Evaluation of predictive statistical models is difficult -some perform better than other in specific contexts but are weaker in other contexts [35].

We advocate using Markov-based models as they behave better for our goal while retain satisfying prediction rates [24, 6, 19]. Specifically, in our prototype we use the Longest Repeating Subsequence (LRS) method [24]. A longest repeating subsequence is the longest repeating sequence of items (e.g. user tasks) where the number of consecutive items repeats more than some threshold  $T$  ( $T$  usually equals one).

#### 3.2 Task Models

Statistical models such as LRS can be beneficial for predicting user actions. However, there are two major drawbacks to such models: 1) in order to predict next actions, training data must be supplied before the first use, and 2) poor quality training data can potentially divert users from using preferred application paths.

Contrary to statistical models which are created at run-time, task models are created during the design phase of an application. Task models are often defined as a description of an interactive task to be performed by the user of an application

through the user interface of the application [13]. A task model represents the static information on users and application tasks and their relationships.

Many different approaches to defining task models have been developed [13]: Hierarchical Task Analysis (HTA) [26], ConcurTaskTrees (CTT) [23], Diane+ [2], MUSE [12], to name few. We use CTT, developed by Patterno [23], as it provides well developed tools for defining concurrent task trees.

Task models successfully describe static, pre-designed interaction with the users. However, it is very difficult (if not impossible) to describe with sufficient accuracy (for user behavior predictions) user-application interaction in case application tasks change dynamically. For example, if the application changes its tasks dynamically based on the information downloaded from the Internet, the task model of such an application would be a high-level description; task models would not be able to model precisely the dynamic tasks created as per downloaded information. This is because information used to create tasks from the Internet is not known to the software developer at the design time, and some generic task or interaction description would have to be used in the task model.

In our prototype we use specially crafted CTT models as pre-loaded training data to statistical learning modules. CTT models used are very basic and do not follow closely CTT standard notation; models are specifically customized for our use.

### 3.3 Learning Models in ADUS

Behavior analysis and learning in our system are provided by two separate knowledge modules. The first module treats user preferences and simple patterns (e.g. modifying the menus or font size). The second module is specialized in LRS-based behavior analysis. Both knowledge modules help the user agent make the necessary decisions that are later reflected on the user interface [21].

To improve LRS predictions we have developed a specialized converter utility that can convert specifically crafted CTT definition into LRS paths database. The converter utility is very basic – the CTT diagrams must be specifically prepared to accommodate our converter tool which involves supplying object tags as per our specification and designing trees with LRS in mind. In the current version of the prototype CTT diagrams are very basic and do not follow closely CTT task types. Previously prepared information from CTT can be then loaded into the LRS module as the default knowledge with a configurable weight (i.e. path preference). This has been designed to: 1) ensure existence of the initial training data (before the first use), and 2) to ensure that the paths supplied by the GUI designer have certain initial priority (weight) over dynamically collected paths. Such measures could improve overall user experience and could improve quality of dynamically collected data.

However, the learning mechanism implemented in ADUS is agnostic - different learning techniques can be implemented at the same time. Learning process is not limited to tasks, but can be extended (with different learning techniques) to any other type of learning.

### 3.4 Applications of Learning Features to the User Interface

Gathered knowledge (e.g., default values, color preferences, or previous actions and selections) is applied by the user agent to the GUI specification. The LRS method is more closely linked to tasks and user interaction paths and has been visually

implemented as a *predictive toolbar* (see Section 4.3 and Figure 4). The user agent automatically inserts this toolbar in the application window (unless otherwise specified) and it shows a configurable number of next-most-probable actions [19].

In cases when software developers anticipate that predictive toolbar would not be useful for the user (e.g. applications where the toolbar would not be visible, or where tasks are not executed through buttons), the LRS module could be used by the visitor agent through the user agent. Section 4.3 presents in detail usage modalities of the LRS module.

## 4 Using ADUS in a Sample Application

To show the benefits of learning techniques to GUI and complex GUI transformations we have applied the ADUS approach to a multi-agent application –the *Software Retrieval Service (SRS)* [15]. The Software Retrieval Service tries to solve one of the most frequent tasks for an average computer user: to search, download, and install new software.

In the following we briefly introduce the agents that participate in the SRS and then we describe how the ADUS approach is applied. The resulting system is tested by real users in Section 5.

### 4.1 The Software Retrieval Service (SRS)

The Software Retrieval Service [15] is an application that helps naive users to find, download, and install new software on their devices. The SRS is distributed between

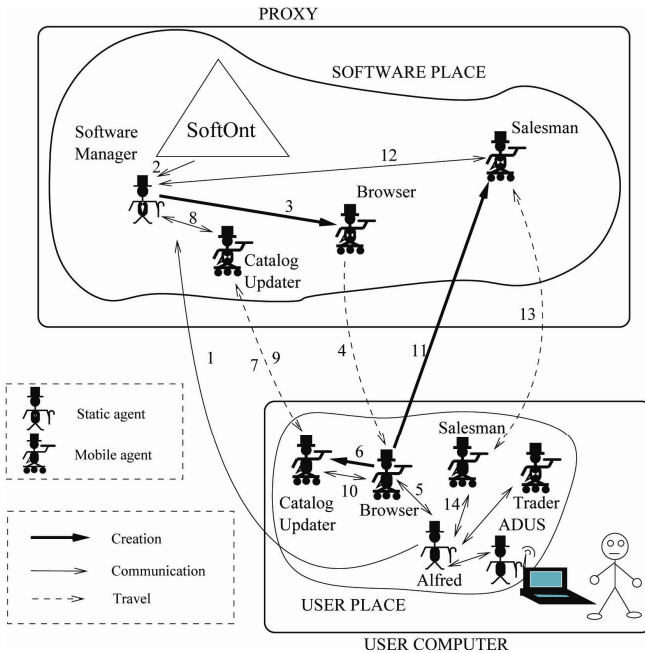


Fig. 2. Main architecture for the Software Retrieval Service

the user's device (also known as *user place*) and a proxy location (known as *software place*), as illustrated in Figure 2.

In the following paragraphs we briefly describe the main agents of the SRS (more details about this system can be found in [15]):

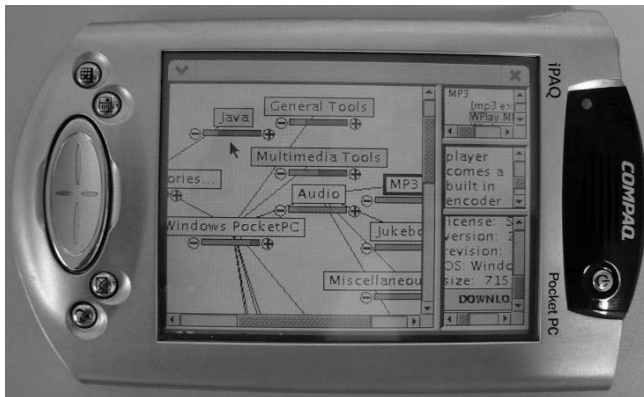
1. *The Alfred agent.* It is a user agent that serves the user and is in charge of storing as much information about the user equipment, preferences, and context as possible. Mobile agent technology allows that mobile agents can learn (e.g. using information from the Web) about previously unknown contexts.
2. *The Software Manager agent.* It creates and provides the Browser agent with a catalog of the available software, according to the requirements supplied by Alfred (on behalf of the user, step 1 in Figure 2), i.e., it is capable to obtain customized metadata about the underlying software.
3. *The Browser agent.* It travels to the user device (step 4) with aim to interact with the user (see Figure 3) in order to help her/him browse the software catalog (step 5).

Working in this way – without ADUS – the Browser agent directly generates its GUI on the user device without knowing user preferences and user device capabilities.

## 4.2 Using ADUS with the Software Retrieval Service

When applying the ADUS approach to the SRS application, Alfred plays the role of user agent and the Browser agent behaves as a visitor agent that arrives to the user device with the purpose of creating a GUI. An ADUS agent will be required to facilitate indirect user interface generation. The ADUS agent interacts with the SRS agents as follows:

1. The Browser agent (as depicted in Figure 2) sends the XUL specification of the GUI to Alfred.
2. Alfred amends the XUL specification according to the user preferences, context, and device capabilities. In this example, size and location of “split panes” are set by Alfred.



**Fig. 3.** Java Swing Browser GUI created indirectly on a PDA

3. Alfred delegates the generation of the GUI to an ADUS agent, who renders the GUI, interacts with the user, and feeds interaction data to Alfred (the user agent) and the Browser (the visitor agent). Figure 3 shows the Java GUI generated by the ADUS agent for a Pocket PC PDA.
4. GUI events and data received by the ADUS agent are communicated to Alfred and the Browser agent for further processing. Alfred stores and analyses such data to predict future user actions, and the Browser agent reacts to the selections or data entered by the user by generating new or updating the existing GUI.

The above process is repeated until the Browser (the visitor agent) finishes its tasks on the user device.

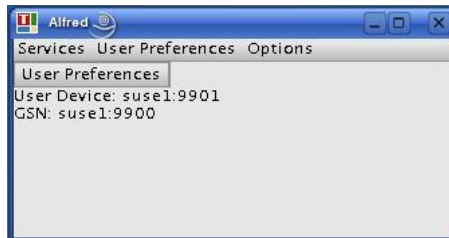
### 4.3 The Learning Process in the SRS

As described earlier, behavior analysis and learning are provided by the user agent (Alfred in the case of the SRS), which treats user preferences and predicts the user behavior following the stored patterns.

Once users start using the application, Alfred collects the necessary data by monitoring user-executed actions in an effort to predict the next task. In the current version of our prototype, the user agent Alfred monitors task execution only through button widgets. As the SRS Browser agent uses a customized interaction model, the visitor agent (the Browser agent in the example) can use the LRS module via the user agent (Alfred) to benefit from the learning features of the system (as described in Section 3.4).

The Browser agent uses the LRS module described earlier via Alfred to automatically expand or collapse browsing nodes (see Figure 3). The user agent will then expand the nodes that are identified as the next most probable nodes to be opened by the user<sup>2</sup>.

In addition to the SRS Browser agent GUI, Alfred has its own GUI that is designed for configuration of user preferences, service options, and execution of other services. This GUI features the predictive toolbar automatically generated by Alfred as described in Section 3.4 and depicted in Figure 4. To improve the quality of training data, and to provide initial training data to the LRS module in Alfred's GUI, we have developed a CTT task model (see Figure 5). The task paths are extracted from the model using a converter utility and path weight is assigned to the paths.



**Fig. 4.** Alfred's GUI – predictive toolbar

<sup>2</sup> The main task of the Browser agent is to help the user the user to browse a software catalogue to find a certain software.

## 5 Performance Evaluation

In this Section we present results of the performance tests and analyze differences in performance between using SRS with and without ADUS approach.

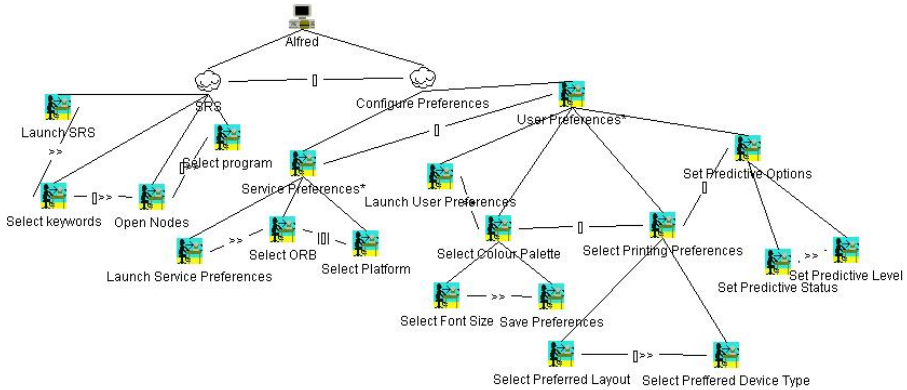


Fig. 5. CTT model for Alfred's GUI

In our test, users<sup>3</sup> were asked to retrieve several pieces of software using the SRS application. The first half of the participating users used the SRS application without the ADUS architecture (direct GUI generation). The second half used the SRS application with ADUS (indirect generation of GUIs). 50 users with mixed levels of skill participated in this test.

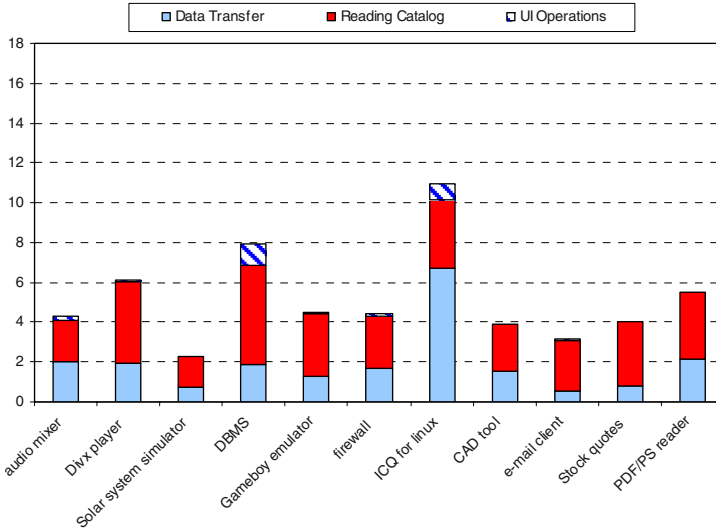
In the first test we compare how the learning features of ADUS improve the system from the point of view of time-consuming tasks. Measured times have been divided into three categories:

- *Data transfer*: this is the time spent by the system 1) to send the different software catalogs to the user device, 2) to move an agent across the network, and 3) to invoke remote procedure calls<sup>4</sup>.
- *Reading catalog*: this category represents the time spent by the user to read/browse the software catalog shown on the device screen; this time includes to open/close a catalog node to read its information.
- *UI operations*: This measure quantifies the time spent by the system on GUI generation (and monitoring, when ADUS is used).

In [21] we showed that just using ADUS (without any prediction) improved the performance of the SRS despite the small overhead due to the indirect GUI generation and monitoring. From Figures 7 and 8 we can observe that the use of the LRS method reduce the total time spent by users to find the software and even the time spent by the system to generate GUIs: when estimations of user behavior are correct, users save

<sup>3</sup> The authors would like to express their gratitude to all persons participating in this study.

<sup>4</sup> Intelligent (mobile) agents in the SRS decide between whether to use remote procedure call or movement approach depending on the execution environment.



**Fig. 6.** Time-consuming tasks for SRS without ADUS

several GUI interactions (and the system saves the corresponding (indirect) GUI generations). Figure 6 depicts times spent on the SRS application without ADUS.

When the predictive features are used ADUS utilizes the data obtained from monitoring interaction between the user and the Browser agent to predict the users' next most probable action (see Section 3). The SRS application then expands and collapses browsing nodes according to the next most probable action. This way, the user interface is generated fewer times: multiple nodes are expanded or collapsed at the same time with only one processing of UI. In the previous version, without predictive features, nodes are expanded by the user manually which triggered additional UI operations.

The second test gives indication of whether predictive features were used and if they were useful. In Figure 9 we present usage of predictive features and the ratio of correct predictions. “Right” represents the percentage of correct predictions that have been followed by users. “Wrong” represents misleading predictions that have not been followed by users. “Ignored” represents percentage of correct predictions that were ignored by the users (they follow a non-optimal path).

Figure 9 shows that the predictive features had a good ratio of successful predictions (on average 90.25%). The average percentage of wrong predictions was 9.74%. 69.74% (on average) of requests followed the correct prediction which implies that predictive features have been seen as useful by most of the users. A certain percentage of requests (20.51%) however did not see the features as useful or felt that the predictions are erroneous.

In the next test we can observe that due to the predictive features the SRS Browser agent loads a better sample of data leading to lower network utilization (cost saving if wireless networks are used) which also results in better processing of the information from the network as more relevant data are downloaded.



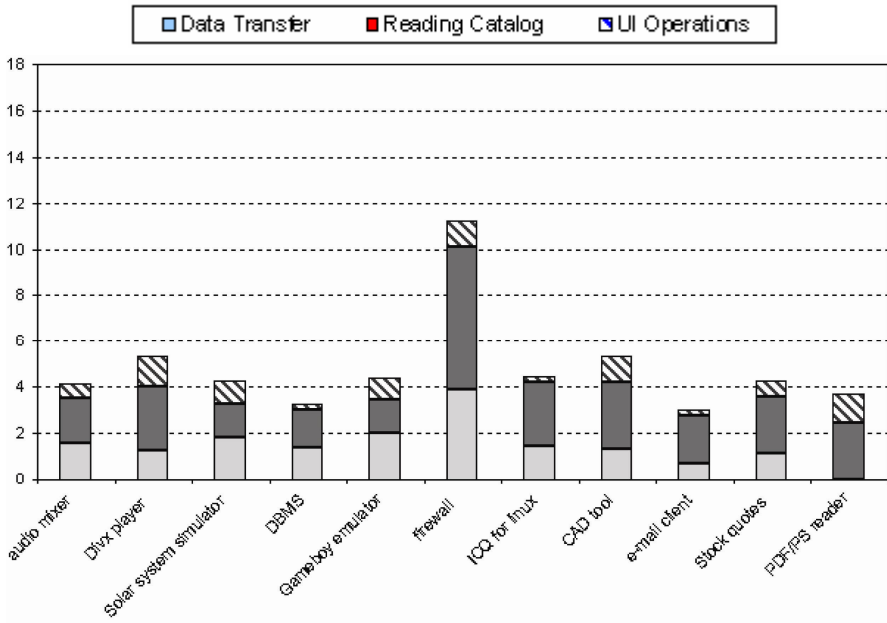


Fig. 7. Time-consuming tasks for SRS + ADUS without predictive features

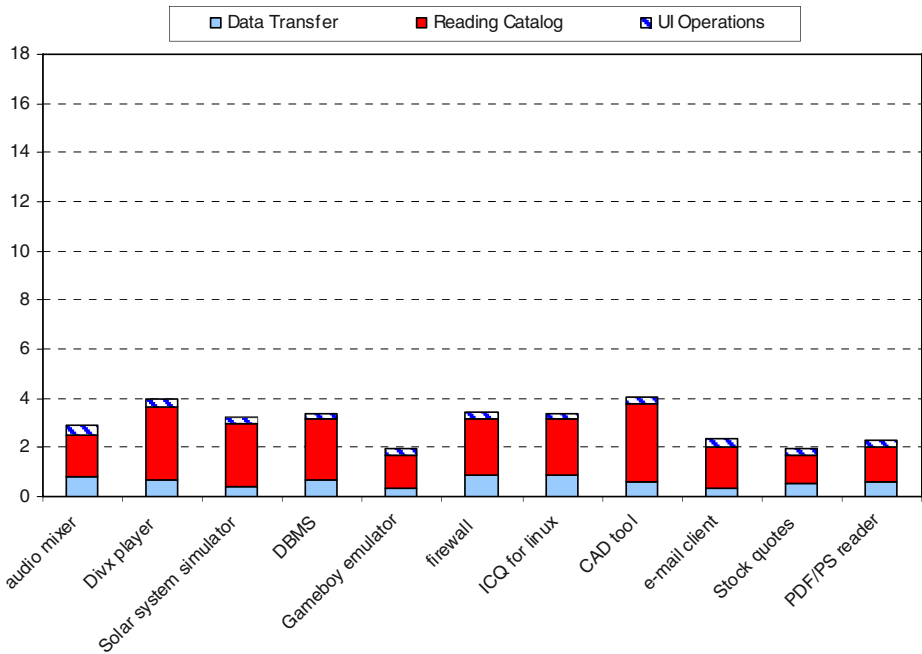


Fig. 8. Time-consuming tasks for SRS + ADUS with predictive features

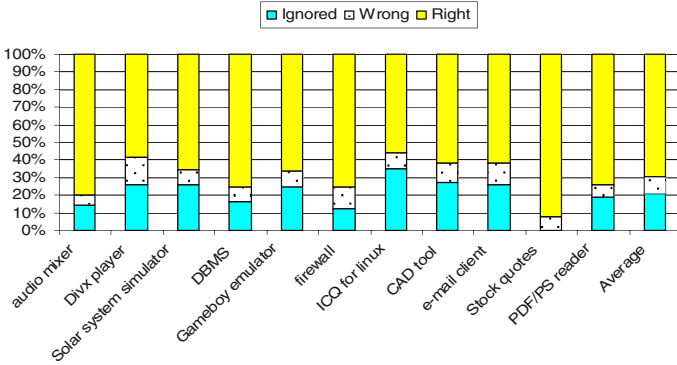


Fig. 9. Usage of Predictive Features

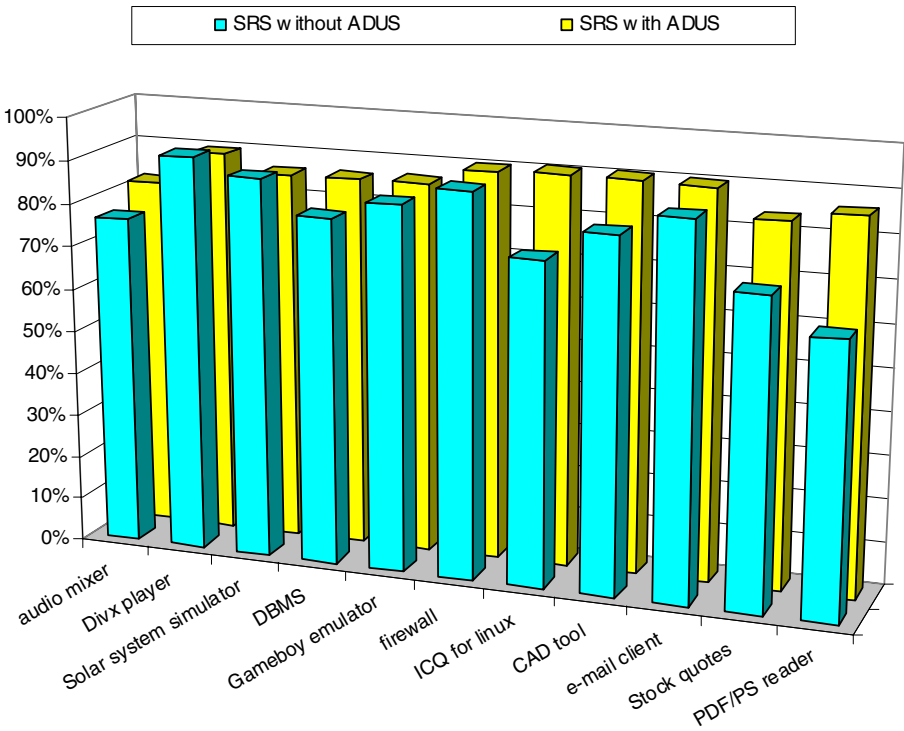


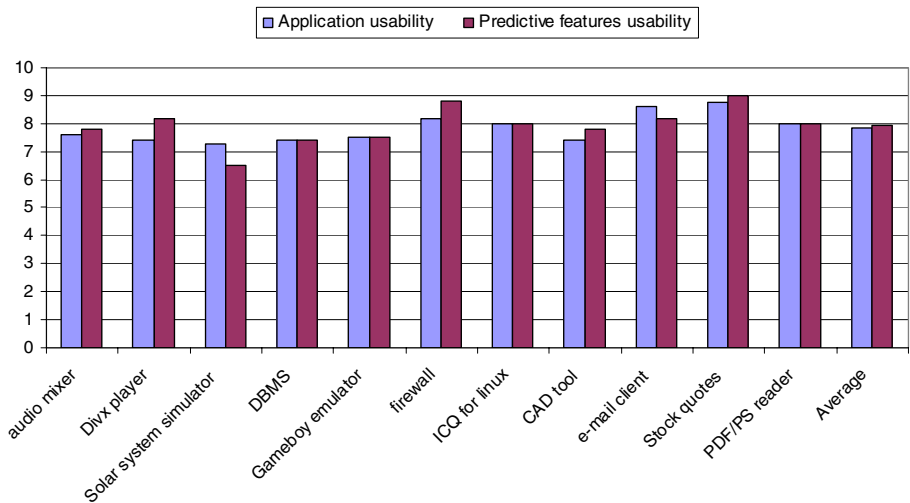
Fig. 10. Browser (agent) intelligence with and without predictive features

This measurement is defined as *Browser (agent) intelligence* [15] and represents efficiency in refining software catalogs shown to the user.

Figure 10 shows a comparison among two versions of the Browser agent intelligence; the higher percentage, the better network and processing usage. On average the improvement due to ADUS with predictive features ranged from -2% to 23% (average of averages was 7%). To conclude, time to find the requested application

using the SRS application with ADUS and predictive features has been improved through lower UI operations, network consumption and information processing due to correct predictions made by the system.

In addition to the measurable indicators we asked users to express the usefulness of the predictive features in the SRS application. Usability was measured in a relative way; users were asked to compare the SRS application without ADUS to the SRS application with ADUS with predictive features and the usability of predictive features in comparison with the original SRS without ADUS: scores range from 0 (not useful) to 10 (very useful). The score above 5 signifies that the ADUS versions of program are more preferred. Figure 11 shows the usability of 1) SRS with and without ADUS predictive features and 2) usability of predictive features alone in SRS with ADUS in comparison to the SRS without ADUS. The usability rating was surveyed for every task in order to understand better usability of predictive features relating to a particular task.



**Fig. 11.** Average usability of two SRS versions and predictive features

On average, the SRS version with ADUS and predictive features was seen as more usable than the version of SRS without ADUS. Similar results were obtained for the usability of predictive features. However, in some cases usability of predictive features has a much lower score than the application usability – this was typically a result of an erroneous prediction that confused users. In total, both the improved application and predictive features scored almost 3 points above the old system versions which shows that the improvements to the system have been seen as usable.

### Results Summary

Tests were conducted with 50 users to demonstrate quantifiable difference between two versions of the SRS application: without and with ADUS and predictive features. It has been demonstrated that, although general GUI processing is increased when following ADUS approach, *the actual processing time decreases due to the application of predictive features*. In addition, information processing and network operations are reduced, which lowers the operational and usage cost of mobile applications on wireless networks.

Tests were also designed to measure usability of the system improvements through time to download, usage ratio of predictive features and number of correct predictions by the system. All tests concluded that improvements to the original application were made; a good percentage of predictions were correct and the predictive features have been used by the testers.

Furthermore we have examined some subjective factors: relative usability of two applications and relative usability of predictive features. The survey showed that both the improved application and predictive features were seen more usable than the original versions.

## 6 State of the Art and Related Work

In this section we present several approaches related to the work presented in this paper. Various approaches to adapting user interfaces to different devices are present. The approaches can be grouped into two categories: web applications and classic applications. While the first category [5, 8] treats only web content and transformations of web content in order to be usable on other (mostly mobile) devices, the second category treats the problems of universally defining the user interface, so it can be later reproduced by various program implementations [1, 27, 11, 32, 22] —or middleware— on various platforms. Solutions are usually designed as client-server and are developed for specific platforms.

Some researchers use software agents (or software entities) [14, 7, 25] which should not be confused for mobile agents. Software agents are software programs that rarely offer any interoperability or mobility and are frequently specifically written for a particular case or application. Lumiere [7] system gives user behavior anticipation through the use of Bayesian models but does not offer any mobility and can be used only in Microsoft Office applications and with use of user profiles. Seo et al. [25] investigate software entities that are standalone, desktop applications. Such entities monitor use of the particular web browser application and provide some anticipation of interaction. The Eager system [28] anticipates user actions but does not offer any mobility and is written for specific operating system/application set. Execution of such system relies on generation of macro scripts within the used application set.

Improving user interface usability is a complex area and many approaches to improving usability exist. We will focus on three main approaches to improve user interface usability: user interface metrics, data mining – user behavior prediction, and task models. The basic concept is to collect user interface metrics for a web site [10]. Usually, collected data are used to perform traffic-based analysis (e.g., pages-per-visitor, visitors-per-page), time-based analysis (e.g., page view durations, click paths) or number of links and graphics on the web pages. These methods fail to give prediction of user behavior, and results can be influenced by many factors. In addition, such analysis is usually used during the UI design (and not in run-time) to improve existing or create new interfaces.

Many models that treat to predict user behavior are based on Markov chains [6]. Predictions are made based on the data from usage logs. More advanced models, like Longest Repeating Subsequence (LRS) [24] or Information Scent [3] perform data mining seeking to analyze navigation path based on server logs, similarity of pages, linking structure and user goals. These models incorporate parts of Markov models in order to give better results. Our prototype uses LRS model as described in Section 3.

Task models are often defined as a description of an interactive task to be performed by the user of an application through the application's user interface [13]. Task model is defined during the application design and gives information on user and application tasks and their relationships. Many different approaches to defining task models have been developed [13]: Hierarchical Task Analysis (HTA) [26], Concurrent Task Trees (CTT) [23], Diane+ [2], MUSE [12], to name few. Task models are typically used to help define and design user interface, and sometimes also to help create user interfaces during the design. In our prototype we use task models as source of training information for user interaction analysis.

## 7 Conclusions and Future Work

This paper presents results of performance and usability studies on ADUS, our proposal for adaptive user interface generation, which is based on mobile agents. In addition, it allows the user behavior monitoring due to its indirect user interface generation method. As summary, the main advantages of our approach are:

- Transparent adaptation of abstract user interface definition to concrete platforms, in an indirect way. GUIs supplied by visitor agents are generated correctly (according to the user preferences and device capabilities) if they are specified in XUL by visitor agents.
- Visitor agents do not need to know how to generate GUIs in different devices. Also the direct generation of GUIs by visitor agents can be easily avoided; direct GUI generation could undermine platform's efforts to improve user's experience and allow uncontrolled malicious behaviors such as phishing.
- User interfaces are adapted to meet the specific user's context and preferences without user or developer intervention.
- Any user interaction can be monitored by the system in order to help the user to interact with future invocations of services.
- The system learns from the user behavior to anticipate future user actions, with the goal of improving the performance and usability. The user behavior is analyzed and next most probable action is advertised. The prediction rate of the proposed algorithm used in our prototype is satisfactory. However, any other predictive algorithm or model could be used in ADUS.

Finally we have presented some performance and usability tests of the system. The performance results demonstrate that there are no significant processing overheads of the proposed architecture and that some performance benefits could be drawn by reducing GUI, network, and information processing operations through predicting future states of user interaction. The results of the usability survey show that users perceive a system more useful when it follows the ADUS architecture.

As future work we are considering some options for improving the exploitation of user interaction data stored by user agents and expanding user agents' ability to automatically recognize tasks from a wider range of GUI widgets.

## Acknowledgements

This work was supported by the CICYT project TIN2004-07999-C02-02.

## References

1. Abrams, M., Phanouriou, C., Batongbacal, A.L., William, S.M., Shuster, J.E.: Uiml: An appliance-independent XML user interface language. *WWW8 / Computer Networks* 31(11-16), 1695–1708 (1999)
2. Barthet, M.F., Tarby, J.C.: The diane+ method. In: *Computer-aided design of user interfaces*. Namur, Belgium, p. 95120 (1996)
3. Chi, E.H., Pirolli, P., Pitkow, J.: The scent of a site: A system for analyzing and predicting information scent, usage, and usability of a web site. In: *ACM CHI 2000 Conference on Human Factors in Computing Systems* (2000)
4. WWW Consortium, <http://www.w3.org/Mobile/CCPP/>
5. Microsoft Corp. Creating mobile web applications with mobile web forms in visual studio .net (2001), <http://msdn.microsoft.com/vstudio/technical/articles/mobilewebforms.asp>
6. Deshpande, M., Karypis, G.: Selective markov models for predicting web-page accesses. Technical report, University of Minnesota Tech. Report 00-056 (2000)
7. Horvitz, E., Breese, J., Heckerman, D., Hovel, D., Rommelse, K.: The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In: *Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, Madison, WI, pp. 256–265 (July 1998)
8. IBM. Ibm websphere transcoding publisher (2001), <http://www3.ibm.com/software/webserver/transcoding/>
9. Recursion Software Inc. (2006), <http://www.recursionsw.com/voyager.htm>
10. Ivory, M.Y., Sinha, R.R., Hearst, M.A.: Empirically validated web page design metrics. In: *SIGCHI* (2001)
11. Coninx, K., Lyten, K.: An XML runtime user interface description language for mobile computing devices. In: Johnson, C. (ed.) *DSV-IS 2001*. LNCS, vol. 2220. Springer, Heidelberg (2001)
12. Lim, K.Y., Long, J.: *The muse method for usability engineering*. Cambridge University Press, Cambridge (1994)
13. Limbourg, Q., Vanderdonckt, J.: *Comparing Task Models for User Interface Design*. Lawrence Erlbaum Associates, Mahwah (2003)
14. Liu, H., Lieberman, H., Selker, T.: A model of textual affect sensing using real-world knowledge. In: *2003 Int. Conference on Intelligent UIs* (January 2003)
15. Mena, E., Illarramendi, A., Royo, J.A., Goni, A.: A software retrieval service based on adaptive knowledge-driven agents for wireless environments. *Transactions on Autonomous and Adaptive Systems (TAAS)* 1(1) (September 2006)
16. Milojevic, D.: Mobile agent applications. *IEEE Concurrency* 7(3), 80–90 (1999)
17. Milojevic, D., Breugst, M., Busse, I., Campbell, J., Covaci, S., Friedman, B., Kosaka, K., Lange, D., Ono, K., Oshima, M., Tham, C., Virdhagriswaran, S., White, J.: MASIF: The OMG mobile agent system interoperability facility. In: Rothermel, K., Hohl, F. (eds.) *MA 1998*. LNCS (LNAI), vol. 1477. Springer, Heidelberg (1998)
18. Mitrovic, N., Mena, E.: Adaptive user interface for mobile devices. In: Forbrig, P., Limbourg, Q., Urban, B., Vanderdonckt, J. (eds.) *DSV-IS 2002*. LNCS, vol. 2545, pp. 29–43. Springer, Heidelberg (2002)
19. Mitrovic, N., Mena, E.: Improving user interface usability using mobile agents. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) *DSV-IS 2003*. LNCS, vol. 2844, pp. 273–287. Springer, Heidelberg (2003)

20. Mitrovic, N., Royo, J.A., Mena, E.: Adus: Indirect generation of user interfaces on wireless devices. In: 15th Int. Workshop on Database and Expert Systems Applications (DEXA 2004), 7th Int. Workshop Mobility in Databases and Distributed Systems (MDDS 2004). IEEE Computer Society, Los Alamitos (2004)
21. Mitrovic, N., Royo, J.A., Mena, E.: Adaptive user interfaces based on mobile agents: Monitoring the behavior of users in a wireless environment. In: I Symposium on Ubiquitous Computing and Ambient Intelligence, Spain, Thomson-Paraninfo (2005)
22. Molina, J.P., Melia, S., Pastor, O.: Just-ui: A user interface specification model. In: 4th International Conference on Computer-Aided Design of User Interfaces CADUI 2002. Kluwer, Dordrecht (2002)
23. Paterno, F., Santoro, C.: One model, many interfaces. In: Fourth International Conference on Computer-Aided Design of User Interfaces (CADUI 2002). Kluwer Academics, Dordrecht (2002)
24. Pitkow, J., Pirolli, P.: Mining longest repeatable subsequences to predict world wide web surfing. In: 2nd Usenix Symposium on Internet Technologies and Systems (USITS) (1999)
25. Seo, Y.-W., Zhang, B.-T.: Learning user's preferences by analyzing web-browsing behaviors. In: Int. Conf. on Autonomous Agents 2000 (2000)
26. Shepherd, A., Diaper, D.: Analysis and training in information technology tasks, Chicester. In: Task analysis for human-computer interaction (1989)
27. Stottner, H.: A platform-independent user interface description language, Technical Report 16, Institute for Practical Computer Science, Johannes Kepler University Linz (2001)
28. Eager system (1993),  
<http://www.acypher.com/wwid/Chapters/09Eager.html>
29. Thevenin, D., Coutaz, J.: Plasticity of user interfaces: Frame-work and research agenda. In: Proc. of IFIP TC 13 Int. Conf. on Human-Computer Interaction INTERACT 1999, Edinburgh (August 1999)
30. XUL Tutorial (2002), <http://www.xulplanet.com/tutorials/xultu/>
31. usiXML (2004), <http://www.usixml.org/>
32. W3C. Xforms (2000), <http://www.xforms.org/>
33. Wang, A., Srensen, C.-F., Indal, E.: A mobile agent architecture for heterogeneous devices. In: Proc. of the Third IASTED International Conference on Wireless and Optical Communications (WOC 2003) (2003)
34. XI ML (November 1999), <http://www.ximl.org/>
35. Zukerman, I., Albrecht, D.: Predictive statistical models for user modeling. In: Kobsa, A. (ed.) User Modeling and User Adapted Interaction (UMUAI) -The Journal of Personalization Research, volume Ten Anniversary Special Issue. Kluwer Academic Publishers, Dordrecht (2000)

## Questions

### *Jose Campos:*

*Question: When you change from laptop to PDA you might need to change dialogue control, not only the screen layout. Are your agents capable of this?*

Answer: This is an open problem and future work.

# Combining Human Error Verification and Timing Analysis

Rimvydas Rukšėnas<sup>1</sup>, Paul Curzon<sup>1</sup>, Ann Blandford<sup>2</sup>, and Jonathan Back<sup>2</sup>

<sup>1</sup> Department of Computer Science, Queen Mary, University of London  
{rimvydas,pc}@dcs.qmul.ac.uk

<sup>2</sup> University College London Interaction Centre  
{a.blandford,j.back}@ucl.ac.uk

**Abstract.** Designs can often be unacceptable on performance grounds. In this work, we integrate a GOMS-like ability to predict execution times into the generic cognitive architecture developed for the formal verification of human error related correctness properties. As a result, formal verification and GOMS-like timing analysis are combined within a unified framework. This allows one to judge whether a formally correct design is also acceptable on performance grounds, and vice versa. We illustrate our approach with an example based on a KLM style timing analysis.

**Keywords:** Human error, formal verification, execution time, GOMS, cognitive architecture, model checking, SAL.

## 1 Introduction

The correctness of interactive systems depends on the behaviour of both human and computer actors. Human behaviour cannot be fully captured by a formal model. However, it is a reasonable, and useful, approximation to assume that humans behave “rationally”: entering interactions with goals and domain knowledge likely to help them achieve their goals. If problems are discovered resulting from rational behaviour then such problems are liable to be systematic and deserve attention in the design. Whole classes of persistent, systematic user errors may occur due to modelable cognitive causes [1, 2]. Often opportunities for making such errors can be reduced with good design [3]. A methodology for detecting designs that allow users, when behaving in a rational way, to make systematic errors will improve such systems. In the case of safety-critical interactive systems, it is crucial that some tasks are performed within the limits of specified time intervals. A design can be judged as incorrect, if it does not satisfy such requirements. Even for everyday systems and devices, the time and/or the number of steps taken to achieve a task goal can be an indication of the usability or otherwise of a particular design.

We previously [4, 5] developed a generic formal user model from abstract cognitive principles, such as entering an interaction with knowledge of the task and its



subsidiary goals, showing its utility for detecting some systematic user error. So far we have concentrated on the verification of functional correctness (user achieving a task goal) and usability properties (the absence of post-completion errors). Also, the cognitive architecture was recently used to verify some security properties – detecting confidentiality leaks due to cognitive causes [6]. However, none of this work addressed the timing aspects of user interaction. For example, a successful verification that a task goal is achieved only meant that it is *eventually* achieved at some unspecified point in the future. This is obviously insufficient, if the goal of verification is to give evidence that a system satisfies specific timing requirements.

Timing analysis is one of the core concerns in the well-established GOMS methodology [7]. A GOMS model predicts the trace of operators and task completion time. However, since GOMS models are deterministic, this prediction assumes and applies to a single, usually considered as expert or optimal, sequence of operators. Such assumptions may be invalid for everyday interactive systems whose average users do not necessarily know or are trained to follow optimal procedures, or they simply might choose a less cognitively demanding method. Moreover, under pressure, even the operators (expert users) of safety-critical systems may choose sub-optimal and less likely plans of action. This suggests that a timing analysis of interactive systems should include a broader set of cognitively plausible behaviours.

The main goal of this paper is to add into our verification methodology, based on a generic cognitive architecture, a GOMS-like ability to predict execution times. For this, we intend to use timing data provided by HCI models such GOMS. It should be noted of course that such timings are only estimates so “proofs” based on such timings are *not* formal guarantees of a particular performance level. They are not proofs of any real use, just proofs that the GOMS execution times are values within a particular range. Provided that distinction is remembered they can still be of use.

Using the SAL verification tools [8], we combine this ability to prove properties of GOMS timings with the verification of human error related correctness properties based on the traversal of all cognitively plausible behaviours as defined by our user model. This way, rather than considering a single GOMS “run,” a whole series of runs are analyzed together, automatically generating a range of timings depending on the path taken. Such a setting allows one to do error (correctness) analysis first and then, once an error free design is created, do a broad timing analysis within a single integrated system. An advantage of doing so is that the GOMS timings can be used to argue that a systematically possible choice is “erroneous” on course performance grounds: the user model does achieve the goal but very inefficiently. If one potential method for achieving a goal was significantly slower, whilst the task completion would be proved, this might suggest design changes to either disable the possibility of choosing that method or change the design so that if it was taken then it would be easier to accomplish the goal. Similarly, a design chosen on performance grounds to eliminate a poor path might be rejected by our GOMS-like analysis due to its potential for systematic error discovered by the integrated human error analysis.

Many GOMS models support an explicit hierarchy of goals and subgoals. Our previous cognitive architecture was “flat” allowing only atomic user goals and actions. This meant that any hierarchy in user behaviour (task or goal structures) could be

specified only implicitly. In this work, we take a step towards supporting hierarchical specifications of user goals. When needed (e.g., to capture an expert behaviour within a complex interactive system), these can be structured in an appropriate way. Note however that this extension to our cognitive architecture does not necessarily impose hierarchical goal structures on specific user models. To represent unstructured goals, one can simply choose a “flat” hierarchy, as is done in this paper.

One indication of cognitively plausible behaviour is choosing options that are relevant to the task goals when there are several alternatives available. Currently our cognitive architecture is fully non-deterministic in the sense that any user goal or action that is possible according to the principles of cognition, and/or prompted by the interface might be selected for execution. Here we introduce a facility for correlating, in such situations, user choices and task goals, thus ensuring that the user model ignores available but irrelevant alternatives.

Summarising, the main goal and contribution of the work presented in this paper is the integration of user-centred timing analysis with formal verification approach originally developed for reasoning about human error. Our aim here is to demonstrate how this can be done and to indicate the potential of combining the approaches in this complementary way to analyse the behaviour of the interactive system in terms of timing and timing-related errors. More specifically:

- It provides a way of creating GOMS-like cognitively plausible variations of methods of performing a task that emerge from a formal model of behaviour.
- It provides a way of detecting methods that have potential for systematic human error occurring using the same initial GOMS-like specification.
- The GOMS-like predictions of timings open the possibility of detecting some (though not all) classes of specific errors that could occur due to those timings, whilst still doing in parallel time-free error analysis based on the verification of various correctness properties.
- It allows our concept of systematic error to be extended in an analysis to include “erroneous” choices in the sense of choosing an alternative that, whilst eventually achieving the result, is predicted to be slower than acceptable.
- It introduces into our cognitive architecture a correlation between task goals and user choices thus refining the notion of cognitive plausibility captured by the formal user model.

## 1.1 Related Work

There is a large body of work on the formal verification of interactive systems. Specific aims and focus vary. Here we concentrate on the work most directly linked to our work in this paper.

Whilst GOMS assume error-free performance, this does not preclude them from being used in a limited way to analyse erroneous performance. As noted by John and Kieras [9], GOMS can be used for example to give performance predictions for error recovery times. To do this one simply specifies GOMS models for the task of recovering from error rather than the original task, perhaps comparing predictions for different recovery mechanisms or determining whether recovery can be achieved with

minimal effort. With these approaches the analysis does not identify the potential for human error: the specific errors considered must be decided in advance by the analyst.

Beckert and Beuster [10] present a verification environment with a similar architecture to our user model – connecting a device specification, a user assumption module and a user action module. They use CMN-GOMS as the user action module. The selection rules of the GOMS model are driven by the assumption model and the actions drive the device model. This gives a way of exploring the effect of errors made by the user (incorrect selection decisions as specified in the user assumption module). However, the assumption module has no specific structure, so the decision of what kind of errors could be made is not systematic or formalized but left to the designers of the system. This differs from our approach where we use a cognitive model combined with aspects of a GOMS model. This allows us to reason about systematic error in a way that is based on formalised principles of cognition. They also have not specifically focused on predicting performance times using GOMS, but rather are using it as a formal hierarchical task model.

Bowman and Faconti [11] formally specify a cognitive architecture using the process calculus LOTOS, and then apply a temporal interval logic to analyse constraints, including timing ones, on the information flow and transformation between the different cognitive subsystems. Their approach is more detailed than ours, which abstracts from those cognitive processes.

In the area of safety-critical systems, Rushby *et al* [12] focus on mode errors and the ability of pilots to track mode changes. They formalize plausible mental models of systems and analyse them using the Mur $\phi$  verification tool. The mental models though are essentially abstracted system models; they do not rely upon structure provided by cognitive principles. Neither do they attempt timing analysis. Also using Mur $\phi$ , Fields [13] explicitly models observable manifestations of erroneous behaviour, analysing error patterns. A problem of this approach is the lack of discrimination between random and systematic errors. It also implicitly assumes there is a correct plan, from which deviations are errors.

Temporal aspects of usability have also been investigated in work based on the task models of user behaviour [14, 15]. Fields *et al* [14] focus on the analysis of situations where there are deadlines for completing some actions and where the user may have to perform several simultaneous actions. Their approach is based on Hierarchical Task Analysis and uses the CSP formalism to specify both tasks and system constraints. Lazace *et al* [15] add quantitative temporal elements to the ICO formalism and use this extension for performance analysis. Both these approaches consider specific interaction scenarios which contrasts to our verification technique supporting the analysis of all cognitively plausible behaviours. The efficiency of interaction, albeit not in terms of timing, is also explored by Thimbleby [16]. Using Mathematica and probabilistic distributions of usage of menu functions, he analyses interface complexity. The latter is measured as the number of actions needed to reach desired menu options.

## 2 HUM-GOMS Architecture

Our cognitive architecture is a higher-order logic formalisation of abstract principles of cognition and specifies a form of cognitively plausible behaviour [17]. The architecture specifies possible user behaviour (traces of actions) that can be justified in terms of specific results from the cognitive sciences. Real users can act outside this behaviour of course, about which the architecture says nothing. However, behaviour defined by the architecture can be regarded as potentially systematic, and so erroneous behaviour is similarly systematic in the design. The predictive power of the architecture is bounded by the situations where people act according to the principles specified. The architecture allows one to investigate what happens if a person acts in such plausible ways. The behaviour defined is neither “correct” nor “incorrect.” It could be either depending on the environment and task in question. We do not attempt to model the underlying neural architecture nor the higher-level cognitive architecture such as information processing. Instead our model is an abstract specification, intended for ease of reasoning.

### 2.1 Cognitive Principles

In the formal user model, we rely upon abstract cognitive principles that give a *knowledge level* description in the terms of Newell [18]. Their focus is on the internal goals and knowledge of a user. These principles are briefly discussed below.

*Non-determinism.* In any situation, any one of several cognitively plausible behaviours might be taken. It cannot be assumed that any specific plausible behaviour will be the one that a person will follow where there are alternatives.

*Relevance.* Presented with several options, a person chooses one that seems relevant to the task goals. For example, if the user goal is to get cash from an ATM, it would be cognitively implausible to choose the option allowing one to change a PIN. A person could of course press the wrong button by accident. Such classes of error are beyond the scope of our approach, focussing as it does on systematic slips.

*Mental versus physical actions.* There is a delay between the moment a person mentally commits to taking an action (either due to the internal goals or as a response to the interface prompts) and the moment when the corresponding physical action is taken. To capture the consequences of this delay, each *physical* action modelled is associated with an internal *mental* action that commits to taking it. Once a signal has been sent from the brain to the motor system to take an action, it cannot be revoked after a certain point even if the person becomes aware that it is wrong before the action is taken. To reflect this, we assume that a physical action immediately follows the committing action.

*Pre-determined goals.* A user enters an interaction with knowledge of the task and, in particular, task dependent sub-goals that must be discharged. These sub-goals might concern information that must be communicated to the device or items (such as bank cards) that must be inserted into the device. Given the opportunity, people may

attempt to discharge such goals, even when the device is prompting for a different action. Such *pre-determined* goals represent a partial plan that has arisen from knowledge of the task in hand, independent of the environment in which that task is performed. No fixed order other than a goal hierarchy is assumed over how pre-determined goals will be discharged.

*Reactive behaviour.* Users may react to an external stimulus, doing the action suggested by the stimulus. For example, if a flashing light comes on a user might, if the light is noticed, react by inserting coins in an adjacent slot.

*Goal based task completion.* Users intermittently, but persistently, terminate interactions as soon as their main goal has been achieved [3], even if subsidiary tasks generated in achieving the main goal have not been completed. A cash-point example is a person walking away with the cash but leaving the card.

*No-option based task termination.* If there is no apparent action that a person can take that will help to complete the task then the person may terminate the interaction. For example, if, on a ticket machine, the user wishes to buy a weekly season ticket, but the options presented include nothing about season tickets, then the person might give up, assuming the goal is not achievable.

## 2.2 Cognitive Architecture in SAL

We have formalised the cognitive principles within the SAL environment [8]. It provides a higher-order specification language and tools for analysing state machines specified as parametrised modules and composed either synchronously or asynchronously. The SAL notation we use here is given in Table 1. We also use the usual notation for the conjunction, disjunction and set membership operators. A slightly simplified version of the SAL specification of a transition relation that defines our user model is given in Fig. 1, where predicates in *italic* are shorthands explained later on. Below, whilst explaining this specification (SAL module *User*), we also discuss how it reflects our cognitive principles.

**Table 1.** A fragment of the SAL language

Notation	Meaning
$x:T$	$x$ has type $T$
$\lambda(x:T):e$	a function of $x$ with the value $e$
$x' = e$	an update: the new value of $x$ is that of the expression $e$
$\{x:T \mid p(x)\}$	a subset of $T$ such that the predicate $p(x)$ holds
$a[i]$	the $i$ -th element of the array $a$
$r.x$	the field $x$ of the record $r$
$r \text{ WITH } .x := e$	the record $r$ with the field $x$ replaced by the value of $e$
$g \rightarrow \text{upd}$	if $g$ is true then update according to $\text{upd}$
$c \square d$	non-deterministic choice between $c$ and $d$
$\square(i:T):c_i$	non-deterministic choice between the $c_i$ with $i$ in range $T$

*Guarded commands.* SAL specifications are transition systems. Non-determinism is represented by the non-deterministic choice,  $\square$ , between the named guarded commands (i.e. transitions). For example, *CommitAction* in Fig. 1 is the name of a family of transitions indexed by  $g$ . Each guarded command in the specification describes an action that a user *could* plausibly take. The pairs *CommitAction* – *PerformAction* of the corresponding transitions reflect the connection between the physical and mental actions. The first of the pair models committing to a goal, the second actually taking the corresponding action (see below).

*Goals structure.* The main concepts in our cognitive architecture are those of user goals and aims. A user aim is a predicate that partially specifies model states that the user intends to achieve by executing some goal. User goals are organised as a hierarchical (tree like) goal–subgoals structure. The nodes of this tree are either compound or atomic:

**Atomic.** Goals at the bottom of the structure (tree leaves) are atomic: they consist of (map to) an action, for example, a device action.

**Compound.** All other goals are compound: they are modelled as a set of task subgoals.

In this paper, we consider an essentially flat goal structure with the top goal consisting of atomic subgoals only. We will explore the potential for using hierarchical goal structures in subsequent work.

In SAL, user goals and aims are modelled as arrays, respectively, *Goals* and *Aims*, which are parameters of the *User* module. Each element in *Goals* is a record with the following fields:

**Guard.** A predicate, denoted *grd*, that specifies when the goal is enabled, for example, due to the relevant device prompts.

**Choice.** A predicate (choice strategy), denoted *choice*, that models a high-level ordering of goals by specifying when a goal can be chosen. An example of the choice strategy is: “choose only if this goal has not been chosen before.”

**Aims.** A set of records consisting of two fields, denoted *aims*, that essentially models the principle of relevance. The first one, *state*, is a reference to an aim (predicate) in the array *Aims*. The conjunction of all the predicates referred to in the set *aims*, defined by the predicate *Achieved*( $g$ ) for a goal  $g$ , fully specifies the model states the user intends to achieve by executing this goal. For the top goal, denoted *TopGoal*, this conjunction coincides with the main task goal. The second field, *ignore*, specifies a set of goals that are irrelevant to the aim specified by the corresponding field *state*. Note that the same effect could be achieved by providing a set of “promising” actions. However, since in our approach the relevance of a goal is generally interpreted in a very wide sense, we expect that the “ignore” set will be a more concise way of specifying the same thing.

**Subgoals.** A data structure, denoted *subgoals*, that specifies the subgoals of the goal. It takes the form *comp*(*gls*) when the goal consists of a set of subgoals *gls*. If the goal is atomic, its subgoals are represented by a reference, denoted *atom*(*act*) to an action in the array *Actions* (see below).

## TRANSITION

```

[] (g:GoalRange, p:AimRange) : CommitAction:
  NOT(comm) ^
  finished = notf ^
  atom?(Goals[g].subgoals) ^
  Goals[g].grd(in, mem, env) ^
  Goals[g].choice(status, g) ^
  (g ≠ ExitGoal ^ Relevant(g, p)
  ∨
  g = ExitGoal ^ MayExit)
  →
  commit'[act(Goals[g].subgoals)]
  = committed;
  t' = t + CogOverhead;
  status' = status
  WITH .trace[g] := TRUE
  WITH .length := status.length + 1
[]
[] (a:ActionRange) : PerformAction:
  commit[a] = committed →
  commit'[a] = ready;
  Transition(a)
[]
ExitTask:
  Achieved(TopGoal)(in, mem) ^
  NOT(comm) ^
  finished = notf
  →
  finished' = ok
[]
Abort:
  NOT(EnabledRelevant(in, mem, env)) ^
  NOT(Achieved(TopGoal)(in, mem)) ^
  NOT(comm) ^
  finished = notf
  →
  finished' =
  IF Wait(in, mem)
  THEN notf
  ELSE abort ENDIF
[]
Idle:
  finished = notf →

```

Fig. 1. User model in SAL (simplified)

*Goal execution.* To see how the execution of an atomic goal is modelled in SAL consider the guarded command *PerformAction* for doing a user action that has been previously committed to:

```

commit[a] = committed →
  commit'[a] = ready;
  Transition(a)

```

The left-hand side of  $\rightarrow$  is the guard of this command. It says that the rule will only activate if the associated action has already been committed to, as indicated by the element  $a$  of the local variable array `commit` holding value `committed`. If the rule is then non-deterministically chosen to fire, this value is changed to `ready` to indicate there are now no commitments to physical actions outstanding and the user model can select another goal. Finally, *Transition*( $a$ ) represents the state updates associated with this particular action  $a$ .

The state space of the user model consists of three parts: input variable `in`, output variable `out`, and global variable (memory) `mem`; the environment is modelled by a global variable, `env`. All of these are specified using type variables and are instantiated for each concrete interactive system. The state updates associated with an atomic goal are specified as an action. The latter is modelled as a record with the fields `tout`, `tmem`, `tenv` and `time`; the array `Actions` is a collection of all user actions.

The `time` field gives the time value associated with this action (see Section 2.3). The remaining fields are relations from old to new states that describe how two components of the user model state (outputs `out` and memory `mem`) and environment `env` are updated by executing this action. These relations, provided when the generic user model is instantiated, are used to specify *Transition(a)* as follows:

```
t' = t + Actions[a].time;
out' ∈ {x:Out | Actions[a].tout(in, out, mem)(x)};
mem' ∈ {x:Memory | Actions[a].tmem(in, mem, out')(x)};
env' ∈ {x:Env | Actions[a].tenv(in, mem, env)(x) ∧ possessions}
```

Since we are modelling the cognitive aspects of user actions, all three state updates depend on the initial values of inputs (perceptions) and memory. In addition, each update depends on the old value of the component updated. The memory update also depends on the new value (`out'`) of the outputs, since we usually assume the user remembers the actions just taken. The update of `env` must also satisfy a generic relation, *possessions*. It specifies universal physical constraints on possessions and their value, linking the events of taking and giving up a possession item with the corresponding increase or decrease in the number (counter) of items possessed. For example, it specifies that if an item is not given up then the user still has it. The counters of possession items are modelled as environment components.

*PerformAction* is enabled by executing the guarded command for selecting an atomic goal, *CommitAction*, which switches the commit flag for some action `a` to committed thus *committing* to this action (enabling *PerformAction*). The fact that a goal `g` is atomic is denoted `atom?(Goals[g].subgoals)`. An atomic goal `g` may be selected only when its guard is enabled and the choice strategy for `g` is true. For the reactive actions (goals), their choice strategy is a predicate that is always true. In the case of pre-determined goals, we will frequently use the strategy “choose only if this goal has not been chosen before.” When the user model discharges such a goal, it will not do the related action again without an additional reason such as a device prompt.

The last conjunct in the guard of *CommitAction* distinguishes the cases when the selected goal is `ExitGoal` or not. `ExitGoal` (given as a parameter of the `User` module) represents such options as “cancel” or “exit,” available in some form in most of interactive systems. Thus, a goal `g` that is not `ExitGoal` may be selected only if there exists a relevant aim `p` in the set `Goals[g].aims`, denoted *Relevant(g, p)*. We omit here the formal definition of the relevance condition. On the other hand, if `g` is `ExitGoal` then it can be selected only when either the task goal has been achieved (user does not intend to finish interaction before achieving main goal), or there are no enabled relevant goals (the user will try relevant options if such are available). Again, we omit the formal definition of these conditions here just denoting them *MayExit*.

When an atomic goal `g` is selected, the user model commits to the corresponding action `act(Goals[g].subgoals)`. The time variable `t` is increased by the value associated with “cognitive overhead” (see Section 2.3). The record `status` keeps track of a history of selected goals. Thus, the element `g` of the array `status.trace` is set to true to indicate that the goal `g` has been selected, and the counter of selected goals, `status.length`, is increased. In addition to time-based analysis, this counter provides another way of analysing the behaviour of the user model.



*Task completion.* There are essentially two cases when the user model terminates an interaction: (i) goal based completion when the user terminates upon achieving the task goal, and (ii) no-option based termination when the user terminates since there are no enabled relevant goals to continue. Goal based completion (`finished` is set to `ok`) is achieved by simply “going away” from the interactive device (see the *Exit-Task* command). No-option based termination (`finished` is set to `abort`) models random user behaviour (see the *Abort* command).

The guarded command *ExitTask* states that the user may complete the interaction once the predicate `Achieved(TopGoal)` becomes true and there are no commitments to actions. This action may still not be taken because the choice between enabled guarded commands is non-deterministic. The value of `finished` being `notf` means that the execution of the task continues.

In the guarded command *Abort*, the no-option condition is expressed as the negation of the predicate `EnabledRelevant`. Note that, in such a case, a possible action that a person could take is to wait. However, they will only do so given some cognitively plausible reason such as a displayed “please wait” message. The waiting conditions are represented in the specification by predicate parameter `Wait`. If `Wait` is false, `finished` is set to `abort` to model a user giving up and terminating the task.

### 2.3 Timing Aspects

Following GOMS models, we extend our cognitive architecture with timing information concerning user actions. On an abstract level, three GOMS models, KLM, CMN-GOMS and NGOMSL, are similar in their treatment of execution time [7]. The main difference is that NGOMSL adds, for each user action, a fixed “cognitive overhead” associated with the production-rule cycling. In our model, this corresponds to the goal selection commands (*CommitAction*). Hence, the time variable is increased by the value `CogOverhead` which is a parameter of our user model. For KLM or CMN-GOMS-like analysis, this parameter can be set to 0. In this case, the time variable is increased (*PerformAction* command) only by the value associated with the actual execution of action and specified as `Actions[a].time`. All three GOMS models differ in the way they distribute “mental time” among user actions, but this need only be considered when our cognitive architecture is instantiated to concrete user models. In general, any of the three approaches (or even their combination) can be chosen at this point. In this paper, we will give an example of KLM like timing analysis.

## 3 An Example

To illustrate how the extended cognitive architecture could be used for the analysis of execution time, we consider interaction with a cash machine.

### 3.1 Cash Machine

For simplicity of presentation, we assume a simple design of cash machine. After inserting a bank card, its user can select one of the two options: withdraw cash or

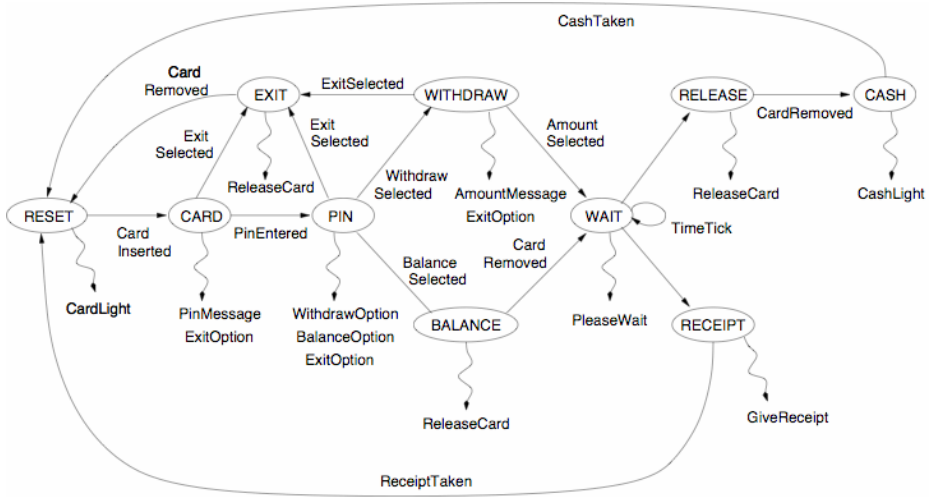


Fig. 2. A specification of the cash machine

checkz balance (see Fig. 2). If the balance option is selected, the machine releases the card and, once the card has been removed and after some delay, prints a receipt with the balance information. If the withdraw option is selected, the user can select the desired amount. Again, after some delay, the machine releases the card and, once it has been removed, provides cash. Note that users are allowed to cancel an interaction with our machine before entering the PIN, and selecting the withdraw option, balance option, or amount, i.e., while the machine is in the CARD, PIN, or WITHDRAW state. If they choose to do so, their card is released.

### 3.2 User Model

Next, we instantiate our cognitive architecture to model cash machine users.

*User aims.* We assume there are two aims, denoted *CashAim* and *BalanceAim*, which might compel a person to use this cash machine. These predicates provide values for the array *Aims*. As an example, the predicate *BalanceAim* is as follows:

$$\lambda(\text{in}, \text{mem}, \text{env}) : \text{env.Receipts} \geq 1 \vee \text{mem.BalanceRead}$$

It states that the balance is checked when either the user has at least one receipt (these are modelled as possession items), or they read the balance on the display and have recorded this fact in their memory.

*User goals.* Taking account of the aims specified, we assume that the machine users, based on the previous experience, have the following pre-determined goals: *Insert-CardGoal*, *SelectBalanceGoal*, *SelectWithdrawGoal*, and *SelectAmountGoal*. As an example, *SelectBalanceGoal* is the following record (the others are similar):

```

grd := λ(in, mem, env) : in.OptionBalance
choice := NotYetDischarged
aims := {}
subgoals := atom(SelectBalance)

```

Thus, this goal may be selected only when a balance option is provided by the interface. The choice strategy `NotYetDischarged` is a pre-defined predicate that allows one to choose a goal only when it has not been chosen before. Since this is an atomic goal, the set `aims` is empty, whereas its subgoal is the actual action (an operator in GOMS terms) of selecting the balance option (see below).

In response to machine signals, the user may form the following reactive goals: `EnterPinGoal`, `TakeReceiptGoal`, `ReadBalanceGoal`, `RemoveCardGoal`, `TakeCashGoal`, and `SelectExitGoal`. Their definitions are similar to those of the pre-determined goals, except that, in this case, the choice strategy always permits their selection.

*User actions.* To fulfil these goals, users will perform an action referred to in the corresponding goal definition. Thus, we have to specify an action for each of the above user goals. As an example, the output update `tout` of the `SelectBalance` action is the following relation:

```
λ(in, out0, mem) : λ(out) : out = Def WITH .BalanceSelected := TRUE
```

where `Def` is a record with all its fields set to false thus asserting that nothing else is done. The memory and environment updates are simply default relations. Finally, the timing of this action (field `time`) is discussed below.

*Task goals.* So far we have introduced all the basic goals and actions of a cash machine user. Now we explain how tasks that can be performed with this cash machine are specified as a suitable `TopGoal`. Here we consider essentially flat goal structures, thus a top goal directly includes all the atomic goals as its subgoals. For the task “check balance and withdraw cash,” `TopGoal` is specified as the following record:

```

grd := True
choice := NotYetDischarged
aims := {(# state := CashAim,
         ignore := {SelectBalanceGoal, ReadBalanceGoal} #),
        (# state := BalanceAim,
         ignore := {SelectAmountGoal} #)}
subgoals := comp({InsertCardGoal, EnterPinGoal, ...})

```

The interesting part of this specification is the attribute `aims`. It specifies that, while performing this task, the user model will have two aims (partial goals) defined by the predicates `CashAim` and `BalanceAim`. Furthermore, when the aim is to check the balance, the user model will ignore the options for selecting the amount as irrelevant to this aim (similarly the balance option and reading balance will be ignored when the aim is to withdraw cash). Of course, this is not the only task that can be performed with this machine. A simpler task, “check balance” (or “withdraw cash”) alone, is also possible. For such a task, the specification of `TopGoal` is the same as above, except that the set `aims` now only includes the first (or second) record.

Note that in this way we have developed an essentially generic user model for our cash machine. Three (or more) different tasks can be specified just by providing appropriate attributes (parameters) `aims`.

### 3.3 KLM Timing

In this paper, we use KLM timings to illustrate our approach. For the cash machine example, we consider three types of the original KLM operators: **K** to press a key or button, **H** to home hands on the keyboard, and **M** to mentally prepare for an action or a series of closely related primitive actions. The duration associated with these types of operators is denoted, respectively, by the constants  $K$ ,  $H$  and  $M$ . The duration values we use are taken from Hudson *et al* [19]. These can be easily altered, if research suggests more accurate times as they are just constants defined in the model.

Since our user model is more abstract, the user actions are actually sequences of the **K** and **H** operators, preceded by the **M** operator. As a consequence, the timing of actions is an appropriate accumulation of  $K$ ,  $H$  and  $M$  operators. For example, `InsertCard` involves moving a hand (**H** operator) and inserting a card (we consider this as a **K** operator), preceded by mental preparation (**M** operator). The time attribute for this action is thus specified as  $M+H+K$ . We also use the same timing for the actions `RemoveCard`, `TakeReceipt` and `TakeCash`. On the other hand, `SelectBalance` involves only pressing a button, since the hand is already on the keyboard. Thus its timing is  $M+K$  (similarly for `SelectWithdraw`, `SelectAmount` and `SelectExit`). `EnterPin` involves pressing a key four times (four digits of PIN), thus its timing is  $M+H+4*K$ . Finally, `ReadBalance` is a purely mental action, giving the timing  $M$ .

In addition to the operators discussed, original KLM also includes an operator, **R**, to represent the system response time during which the user has to wait. Since an explicit device specification is included into our verification approach, there is no need to introduce into the user model time values corresponding to the duration of **R**. System delays are explicitly specified as a part of a device model. For example, in our ATM specification, we assumed that system delays occur after a user selects the desired amount of cash and before the device prints a receipt (the `WAIT` state in Fig. 2).

## 4 Verification and Timing Analysis

So far we have formally developed both a machine specification and a (parametric) model of its user. Our approach also requires two additional models: those of user interpretation of interface signals and effect of user actions on the machine (see [5]), connecting the state spaces of the user model and the machine specification. In this example, these connectors are trivial – they simply rename appropriate variables. Finally, the environment specification simply initialises variables that define user possessions as well as the time variable. Thus, the whole system to analyse is the parallel composition of these five SAL modules. Next we discuss what properties of this system can be verified and analysed, and show how this is done. First we consider the verification of correctness properties.

## 4.1 Error Analysis

In our previous work [4, 5], we mainly dealt with two kinds of correctness properties. The first one (functional correctness) aimed to ensure that, in any possible system behaviour, the user's main goal of interaction (as they perceive it) is eventually achieved. Given our model's state space, this is written in SAL as the following LTL assertion:

$$F(\text{Perceived}(\text{in}, \text{mem})) \quad (1)$$

Here  $F$  means “eventually,” and  $\text{Perceived}$  is the conjunction of all the predicates from the set  $\text{Goals}[\text{TopGoal}]$ . aims as explained earlier.

The second property aimed to catch post-completion errors – a situation when subsidiary tasks are left unfinished once the main task goal has been achieved. In SAL, this condition is written as follows:

$$G(\text{Perceived}(\text{in}, \text{mem}) \Rightarrow F(\text{Secondary}(\text{in}, \text{mem}, \text{env}))) \quad (2)$$

Here  $G$  means “always,” and  $\text{Secondary}$  represents the subsidiary tasks. In our example,  $\text{Secondary}$  is a predicate stating that the total value of user possessions (account balance plus withdrawn cash) in a state is no less than that in the initial state.

Both these properties can be verified by SAL model checkers. With the cash machine design from Fig. 2, the verification of both succeeds for each of the three tasks we specified. Note, however, that both properties only guarantee that the main and subsidiary tasks are eventually finished at some unspecified point in the future. In many situations, especially in the case of various critical systems, designs can be judged as “incorrect” on the grounds of poor performance. Next we show how efficiency analysis is supported by our approach by considering execution times.

## 4.2 Timing Analysis

Model checkers give binary results – a property is either true or false. Because of this, they are not naturally suited for a detailed GOMS-like analysis of execution times. Still, if one is content with timing analysis that produces upper and/or lower limits, model checking is a good option. For example, if it suffices to know that both the main and the subsidiary tasks are finished between times  $T_{\text{low}}$  and  $T_{\text{high}}$ , one can verify the condition

$$G(\text{Perceived}(\text{in}, \text{mem}) \Rightarrow F(\text{Secondary}(\text{in}, \text{mem}, \text{env}) \wedge T_{\text{low}} < \text{time} \wedge \text{time} < T_{\text{high}})) \quad (3)$$

The validity of both (1) and (3) predicts that  $T_{\text{high}}$  is an upper limit for the user model, and thus for any person behaving according to the cognitive principles specified, to properly finish a task. If expert knowledge is needed for such performance, SAL would produce a counter-example (a specific sequence of actions and intermediate states) for property (3). This can be used to determine design features requiring expert knowledge.

As an example, consider the task “check balance and withdraw cash.” Let the threshold for slow execution times be 17 seconds (i.e. 17 000 milliseconds). The verification of property (3) with  $T_{\text{high}}$  equal to 17000 fails. The counter-example shows that

the execution time is slow since the user model goes through the whole interaction cycle (inserting a card, entering a PIN, etc.) twice. A design allowing the task to be performed in a single cycle would improve the execution times. In the next section, we consider such a design.

By verifying property (3) for different  $T_{high}$  and  $T_{low}$  values, the estimates of the upper and lower time limits for a task execution can be determined. However, execution times given by counter-examples provide no clue as to how likely they are, in other words, whether there are many methods of task execution yielding these particular times. Neither do they give the duration of other execution methods. To gather precise timing information for possible execution methods, we use an interactive tool provided by the SAL environment, a simulator. It is possible to instruct the latter to run an interactive system so that the system states defined by some predicate (for example, `Perceived`) are reached. In general, different system states are reached by different execution methods. Thus, one can determine the precise timing of a particular method simply by checking the variable `time` in the corresponding state. A more sophisticated analysis and comparison of timing information can be automated, since the SAL simulator is a Lisp-like environment that allows programming functions for suitable filtering of required information. We will explore this in future work.

## 5 Modified Design

An obvious “improvement” on the previous design is to free users from an early selection of a task. Instead, while in the `WITHDRAW` state, the machine now displays the balance in addition to the amount choices (see Fig. 3). The user can read it and then choose an amount option as needed, thus achieving both task goals in one run. To check whether our expectations are valid, we run the simulator to reach system states where both predicates `Perceived` and `Secondary` are true. Checking execution time in these states indicates an improvement. To find out whether execution times improved for all possible paths reaching the above goal states, we model check property (3) for the same  $T_{high}$ . However, this verification fails again. SAL produces a counter-example where the user model chooses an amount option without first reading the displayed balance and, to achieve both aims, is forced to restart interaction.

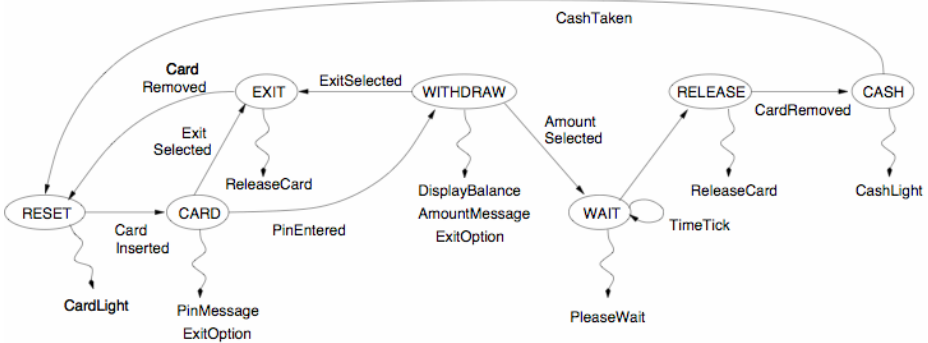


Fig. 3. A specification of the modified design

Furthermore, while the new design is potentially more efficient, it can also lead to systematic user errors, as indicated by a failed verification of property (2). The SAL counter-example shows that the user model, after reading the displayed balance, chooses the exit option, thus forgetting the card. This failure illustrates the close interdependence between correctness and timing properties and the usefulness of our combined approach to the analysis of interactive systems.

In a traditional GOMS analysis this new design is apparently fine as expert non-erroneous behaviour is assumed. However the HUM-GOMS analysis highlights two potentially systematic problems: an attention error and a post-completion error. The expert assumption is thus in a sense required here. Whilst it might be argued that an expert who has chosen that method for obtaining balance and cash would not make the mistake of failing to notice the balance when it was displayed, experimental data suggests that even experts find it hard to eliminate post-completion error in similar situations. Amongst non-expert users both errors are liable to be systematic. The HUM-GOMS analysis has thus identified two design flaws that if fixed would be significant improvements on the design.

A simple fix for both detected flaws is a cash machine similar to our second design, but which, instead of displaying the balance, prints this information and releases the receipt in the same slot and at the same time as the banknotes.

## 6 Conclusion

We have added support for timing analysis into our usability verification approach based on the analysis of correctness properties. This allows both timing analysis and human error analysis to be performed in a single verification environment from a single set of specifications. For this, our cognitive architecture was extended with timing information, as in GOMS models. Our approach uses the existing SAL tools, both the automatic model checkers and the interactive simulator environment, to explore the efficiency of an interactive system based on the models provided. As in our earlier work the cognitive architecture is generic: principles of cognition are specified once and instantiated for a particular design under consideration. This differs from other approaches where a tailored user model has to be created from scratch for each device to be analysed. The generic nature of our architecture is naturally represented using higher-order formalisms. SAL's support for higher-order specifications is the primary reason for developing our verification approach within the SAL environment.

The example we presented aimed to illustrate how our approach can be used for a KLM style prediction of execution times (our SAL specifications are available at <http://www.dcs.qmul.ac.uk/~rimvydas/usermodel/dsvis07.zip>). A difference in our approach is that, if the goal is achieved, the user model may terminate early. Also, if several rules are enabled, the choice between them is non-deterministic. The actual execution time is then potentially a range, depending on the order – there is a maximum and a minimum prediction. These are not real max/min in the sense of saying this is the longest or shortest time it will take, however, just a range of GOMS-like predictions for the different possible paths. In effect, it corresponds to a series of KLM analyses using different procedural rules, but incorporated in HUM-GOMS into a single automated analysis.

Similarly as CCT models [20] and unlike pure GOMS, we have an explicit device specification that has its own timings for each machine response. It is likely that most are essentially instantaneous (below the millisecond timing level) and so approximated to zero time. However, where there are explicit **R** operators in KLM, the corresponding times can be assigned to the device specification.

Even though we illustrated our approach by doing a KLM style analysis, our extension of the cognitive architecture is also capable of supporting CMN-GOMS and NGOMSL approaches to timing predictions. We intend to explore this topic in future work, developing at the same time a hierarchical goal structure.

Another topic of further investigation is timing-related usability errors. We have already demonstrated the capability of our approach to detect potential user errors resulting from the device delays or indirect interface changes without any sort of feedback [4]. The presented extension opens a way to deal with real-time issues (e.g., when system time-outs are too short, or system delays are too long). We also intend to investigate “race condition” errors when two closely fired intentions to action come out in the wrong order [21]. We expect that the inherent non-determinism of our cognitive architecture can generate such erroneous behaviour in appropriate circumstances. Finally, since tool support allows experimentation be done more easily, we believe that our approach can address the scale-up issue and facilitate the analysis of trade-offs between the efficiency of multiple tasks.

**Acknowledgements.** This research is funded by EPSRC grants GR/S67494/01 and GR/S67500/01.

## References

1. Reason, J.: *Human Error*. Cambridge University Press, Cambridge (1990)
2. Gray, W.: The nature and processing of errors in interactive behavior. *Cognitive Science* 24(2), 205–248 (2000)
3. Byrne, M.D., Bovair, S.: A working memory model of a common procedural error. *Cognitive Science* 21(1), 31–61 (1997)
4. Curzon, P., Blandford, A.E.: Detecting multiple classes of user errors. In: Nigay, L., Little, M.R. (eds.) *EHCI 2001*. LNCS, vol. 2254, pp. 57–71. Springer, Heidelberg (2001)
5. Rukšėnas, R., Curzon, P., Back, J., Blandford, A.: Formal modelling of cognitive interpretation. In: Doherty, G., Blandford, A. (eds.) *DSVIS 2006*. LNCS, vol. 4323, pp. 123–136. Springer, Heidelberg (2007)
6. Rukšėnas, R., Curzon, P., Blandford, A.: Detecting cognitive causes of confidentiality leaks. In: *Proc. 1st Int. Workshop on Formal Methods for Interactive Systems (FMIS 2006)*. UNU-IIST Report No. 347, pp. 19–37 (2006)
7. John, B.E., Kieras, D.E.: The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Trans. CHI* 3(4), 320–351 (1996)
8. de Moura, L., Owre, S., Ruess, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: *SAL 2*. In: Alur, R., Peled, D.A. (eds.) *CAV 2004*. LNCS, vol. 3114, pp. 496–500. Springer, Heidelberg (2004)
9. John, B.E., Kieras, D.E.: Using GOMS for user interface design and evaluation: which technique? *ACM Trans. CHI* 3(4), 287–319 (1996)



10. Beckert, B., Beuster, G.: A method for formalizing, analyzing, and verifying secure user interfaces. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 55–73. Springer, Heidelberg (2006)
11. Bowman, H., Faconti, G.: Analysing cognitive behaviour using LOTOS and Mexitl. *Formal Aspects of Computing* 11, 132–159 (1999)
12. Rushby, J.: Analyzing cockpit interfaces using formal methods. *Electronic Notes in Theoretical Computer Science* 43 (2001)
13. Fields, R.E.: Analysis of erroneous actions in the design of critical systems. Tech. Rep. YCST 20001/09, Univ. of York, Dept. of Comp. Science, D. Phil Thesis (2001)
14. Fields, B., Wright, P., Harrison, M.: Time, tasks and errors. *ACM SIGCHI Bull.* 28(2), 53–56 (1996)
15. Lacaze, X., Palanque, P., Navarre, D., Bastide, R.: Performance evaluation as a tool for quantitative assessment of complexity of interactive systems. In: Forbrig, P., Limbourg, Q., Urban, B., Vanderdonckt, J. (eds.) DSV-IS 2002. LNCS, vol. 2545, pp. 208–222. Springer, Heidelberg (2002)
16. Thimbleby, H.: Analysis and simulation of user interfaces. In: *Proc. BCS HCI*, vol. XIV, pp. 221–237 (2000)
17. Butterworth, R.J., Blandford, A.E., Duke, D.J.: Demonstrating the cognitive plausibility of interactive systems. *Formal Aspects of Computing* 12, 237–259 (2000)
18. Newell, A.: *Unified Theories of Cognition*. Harvard University Press (1990)
19. Hudson, S.E., John, B.E., Knudsen, K., Byrne, M.D.: A tool for creating predictive performance models from user interface demonstrations. In: *Proc. 12th Ann. ACM Symp. on User Interface Software and Technology*, pp. 93–102. ACM Press, New York (1999)
20. Kieras, D.E., Polson, P.G.: An approach to the formal analysis of user complexity. *Int. J. Man-Mach. Stud.* 22, 365–394 (1985)
21. Dix, A., Brewster, S.: Causing trouble with buttons. In: *Auxiliary. Proc. HCI 1994* (1994)

## Questions

### **Helmut Stiegler:**

*Question: From where is your human-error model derived which you consider in your specification? Usually, one comes across error processes only during practical use.*

Answer: We are not interested in all kinds of errors, but in errors which are systematic due to design decisions and can be eliminated by modifying them.

### **Paula Kotzé:**

*Question: Can you define the term “cognitive overload” which you defined but set to a value of zero?*

Answer: None recorded.

# Formal Testing of Multimodal Interactive Systems

Jullien Bouchet, Laya Madani, Laurence Nigay, Catherine Oriat, and Ioannis Parissis

Laboratoire d'Informatique de Grenoble (LIG)  
BP 53 38041 Grenoble Cedex 9, France  
Forename.Name@imag.fr

**Abstract.** This paper presents a method for automatically testing interactive multimodal systems. The method is based on the Lutess testing environment, originally dedicated to synchronous software specified using the Lustre language. The behaviour of synchronous systems, consisting of cycles starting by reading an external input and ending by issuing an output, is to a certain extent similar to the one of interactive systems. Under this hypothesis, the paper presents our method for automatically testing interactive multimodal systems using the Lutess environment. In particular, we show that automatic test data generation based on different strategies can be carried out. Furthermore, we show how multimodality-related properties can be specified in Lustre and integrated in test oracles.

## 1 Introduction

A multimodal system supports communication with the user through different modalities such as voice and gesture. Multimodal systems have been developed for a wide range of domains (medical, military, ...) [5]. In such systems, modalities may be used sequentially or concurrently, and independently or combined synergistically. The seminal "Put that there" demonstrator [4] that combines speech and gesture illustrates a case of a synergistic usage of two modalities. The design space described in [25], based on the five Allen relationships, capture this variety of possible usages of several modalities. Moreover, the versatility of multimodal systems is further exacerbated by the huge variety of innovative input modalities, such as the phicons (physical icons) [14]. This versatility results in an increased complexity of the design, development and verification of multimodal systems.

Approaches based on formal specifications automating the development and the validation activities can help in dealing with this complexity. Several approaches have been proposed. As a rule, they consist of adapting existing formalisms in the particular context of interactive systems. Examples of such approaches are the Formal System Modelling (FSM) analysis [10], the Lotos Interactor Model (LIM) [23] or the Interactive Cooperative Objects (ICO), based on Petri Nets [21]. The synchronous approach has also been proposed as an alternative to modelling and verifying by model-checking of some properties of interactive systems [8]. Similarly to the previous approaches, the latter requires formal description of the interactive systems such as Lustre [13] programs on which properties, also described as Lustre programs, are

checked. However, its applicability is limited to small pieces of software, since it seems very hard to fully specify systems in this language.

As opposed to the above approaches used for the design and verification, this paper proposes to use the synchronous approach as a framework for testing interactive multimodal systems. The described method therefore focuses on testing a partial or complete implementation. It consists of automatically generating test data from enhanced Lustre formal specifications. Unlike the above presented methods, it does not require the entire system to be formally specified. In particular, the actual implementation is not supposed to be made in a specific formal language. Only a partial specification of the system environment and of the desired properties is needed.

The described testing method is based on Lutess [9, 22], a testing environment handling specifications written in the Lustre language [13]. Lutess has been designed to deal with synchronous specifications and has been successfully used to test specifications of telecommunication services [12]. Lutess requires a non-deterministic Lustre specification of the user behaviour. It then automatically builds a test data generator that will feed with inputs the software under test (i.e., the multimodal user interface). The test generation may be purely random but can also take into account additional specifications such as operational profiles or behavioural patterns. Operational profiles make it possible to test the system under realistic usage conditions. Moreover, they could be a means of assessing usability as has been shown in [24] where Markov models are used to represent various user behaviours. Behavioural patterns express classes of execution scenarios that should be executed during testing.

A major interest of synchronous programming is that modelling, and hence verifying, software is simpler [13] than in asynchronous formalisms. The objective of this work is to establish that automated testing based on such an approach can be performed in an efficient and meaningful way for interactive and multimodal systems. To do so, it is assumed, according to theoretical results [1], that interactive systems can, to some extent, be assimilated with synchronous programs. On the other hand, multimodality is taken into account through the type of properties to be checked: we especially focus on the CARE (Complementarity, Assignment, Redundancy, Equivalence) [7, 18] properties as well as on temporal properties related to the use over time of multiple modalities.

The structure of the paper is as follows: first, we present the CARE and temporal properties that are specific to multimodal interaction. We then explain the testing approach based on the Lutess testing environment and finally illustrate the application of the approach on a multimodal system developed in our laboratory, Memo.

## 2 Multimodal Interaction: The CARE Properties

Each modality can be used independently within a multimodal system, but the availability of several modalities naturally raises the issue of their combined usage. Combining modalities opens a vastly augmented world of possibilities in multimodal user interface design, studied in light of the four CARE properties in [7, 18]. These properties characterize input and output multimodal interaction. In this paper we focus on input multimodality only. In addition to the combined usage of input modalities, multimodal interaction is characterized by the use over time of a set of modalities.

The CARE properties (Equivalence, Assignment, Redundancy, and Complementarity of modalities) form an interesting set of relations relevant to characterization of multimodal systems. As shown in Fig. 1, while Equivalence and Assignment express the availability and respective absence of choice between multiple modalities for a given task, Complementarity and Redundancy describe relationships between modalities.

- Assignment implies that the user has no choice in performing a task: a modality is then assigned to a given task. For example, the user must click on a dedicated button using the mouse (modality = direct manipulation) for closing a window.
- Equivalence of modalities implies that the user can perform a task using a modality chosen amongst a set of modalities. These modalities are then equivalent for performing a given task. For example, to empty the desktop trash, the user can choose between direct manipulation (e.g. shift-click on the trash) and speech (e.g. the voice command "empty trash"). Equivalence augments flexibility and also enhances robustness. For example, in a noisy environment, a mobile user can switch from speech to direct manipulation using the stylus on a PDA. In critical systems, equivalence of modalities may also be required to overcome device breakdowns.
- Complementarity denotes several modalities that convey complementary chunks of information. Deictic expressions, characterised by cross-modality references, are examples of complementarity. For example, the user issues the voice command "delete this file" while clicking on an icon. In order to specify the complete command (i.e. elementary task) the user must use the two modalities in a complementary way. Complementarity may increase the naturalness and efficiency of interaction but may also provoke cognitive overload and extra articulatory synchronization problems.
- Redundancy indicates that the same piece of information is conveyed by several modalities. For example, in order to reformat a disk (a critical task) the user must use two modalities in a redundant way such as speech and direct manipulation. Redundancy augments robustness but as in complementary usage may imply cognitive overload and synchronization problems.

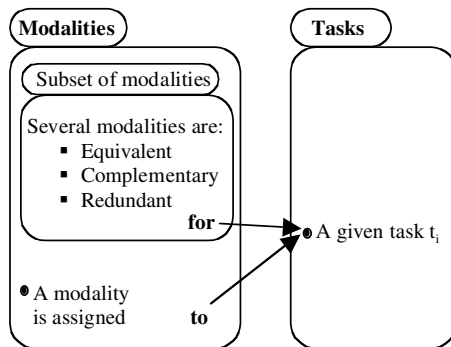


Fig. 1. The CARE relationships between modalities and tasks

Orthogonal to the CARE relationships, a temporal relationship characterises the use over time of a set of modalities. The use of these modalities may occur simultaneously or in sequence within a temporal window  $T_w$ , that is, a time interval. Parallel and sequential usages of modalities within a temporal window are formally defined in [7]. The key point is that the corresponding events from different modalities occur within a temporal window to be interpreted as temporally related: the temporal window thus expresses a constraint on the pace of the interaction. Temporal relationships are often used by fusion software mechanisms [18] to detect complementarity and redundancy cases assuming that users' events that are close in time are related. Nevertheless, distinct events produced within the same temporal window through different modalities are not necessarily complementary or redundant. This is the case for example when the user is performing several independent tasks in parallel, also called concurrent usage of modalities [18]. This is another source of complexity for the software.

The CARE and temporal relationships characterise the use of a set of modalities. They highlight all the diversity of possible input event sequences specified by the user and therefore the complexity of the software responsible for defining the tasks from the captured users' actions. Facing this complexity, we propose a formal approach for testing the software of a multimodal system that handles the input event sequences. In [7], we study the compatibility between what we call system-CARE as defined above and user-CARE properties for usability assessment based on cognitive models such as PUM [3] or ICS [2]. In our formal approach for testing, we focus on system-CARE properties.

### 3 Formal Approach for Testing Multimodal Systems

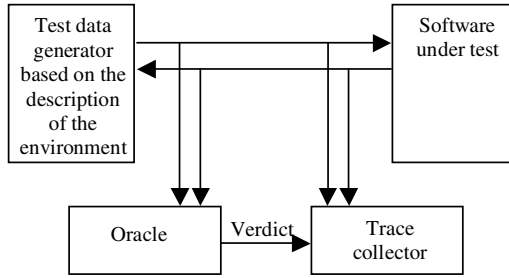
Our approach is based on the Lutess testing environment. In this section, we first present Lutess and then explain how it can be used for testing multimodal systems. In [16] we presented a preliminary study showing the feasibility of our approach and a first definition of the CARE properties that we simplify here. Moreover in [17], we presented in the context of a case study, one way to generate test data, namely the operational profile strategy. In this section, we present the complete approach with three different ways of generating test data.

#### 3.1 Lutess: A Testing Environment for Synchronous Programs

Lutess [9, 22] is a testing environment initially designed for functional testing of synchronous software with boolean inputs and outputs. Lutess supports the automatic generation of input sequences for a program with respect to environment constraints. The latter are assumptions on the possible behaviours of the program environment. Input data are dynamically computed (i.e. while the software under test is executed) to take into account the inputs and outputs that have already been produced.

Lutess automatically transforms the environment constraints into a test data generator and a test harness. The latter:

- links the generator, the software under test and the properties to be checked (i.e. the oracle), and
- coordinates the test execution and records the sequences of input/output values and the associated oracle verdicts (see Fig. 2).



**Fig. 2.** The Lutess environment

The test is operated on a single action-reaction cycle. The generator randomly selects an input vector and sends it to the software under test. The latter reacts with an output vector and feeds back the generator with it. The generator proceeds by producing a new input vector and the cycle is repeated.

In addition to the random generation, several strategies, explained in Section 3.2.4, are supported by Lutess for guiding the generation of test data. In particular, operational profiles can be specified as well as behavioural patterns. The test oracle observes the inputs and the outputs of the software under examination, and determines whether the software properties are violated. Finally the collector stores the input, output and oracle values that are all boolean values.

The software under examination is assumed to be synchronous, and the environment constraints must be written in Lustre [13], a language designed for programming reactive synchronous systems. A synchronous program, at instant  $t$ , reads inputs  $i_t$ , computes and issues outputs  $o_t$ , assuming the time is divided in discrete instants defined by a global clock. The synchrony hypothesis states that the computation of  $o_t$  is made instantaneously at instant  $t$ . In practice, this hypothesis holds if the program computes the outputs within a time interval that is short enough to take into account every evolution of the program environment.

A Lustre program is structured into nodes. A Lustre node consists of a set of equations defining outputs as functions of inputs and local variables. A Lustre expression is made up of constants, variables as well as logical, arithmetic and Lustre-specific operators. There are two Lustre-specific temporal operators: "pre" and "->". "pre" makes it possible to use the last value an expression has taken (at the last tick of the clock). "->", also called "followed by", is used to assign initial values (at  $t = 0$ ) to expressions. For instance, the following program returns a "true" value everytime its input variable passes from "false" to "true" (rising edge).

```

node RisingEdge(in:bool;) returns(risingEdge:bool);
let
    risingEdge = false -> in and not pre in;
tel
  
```

An interesting feature of the Lustre language is that it can be used as a temporal logic (of the past). Indeed, basic logical and/or temporal operators expressing invariants or properties can be implemented in Lustre. For example, `OnceFromTo(A, B, C)` specifies that property A must hold at least once between the instants where events B and C occur. Hence, Lustre can be used as both a programming and a specification language.

## 3.2 Using Lutess for Testing Multimodal Systems

### 3.2.1 Hypotheses and Motivations

The main hypothesis of this work is that, although Lutess is dedicated to synchronous software, it can be used for testing interactive systems. Indeed, as explained above, the synchrony hypothesis states that outputs are computed instantaneously but, in practice, this hypothesis holds when the software is able to take into account any evolution of its external environment (the theoretical foundations of the transformation of asynchronous to synchronous programs are provided in [1]). Hence, a multimodal interactive system can be viewed as a synchronous program as long as all the users' actions and external stimuli are caught. In a different domain than Human-Computer Interaction, Lutess has been already successfully used under the same assumption of testing telephony services specifications [12].

To define a method for testing multimodal input interaction we focus on the part of the interactive system that handles input events along multiple modalities. Considering the multimodal system as the software under test, the aim of the test is therefore to check that a sequence of input events along multiple modalities represented are correctly processed to obtain appropriate outputs such as a complete task. To do so with Lutess, one must provide:

1. *The interactive system as an executable program*: no hypothesis is made on the software implementation. Nevertheless, in order to identify levels of abstraction for connecting Lutess with the interactive system, we will assume that the software architecture of the interactive system is along the PAC-Amodeus software architecture [18]. Communication between Lutess and the interactive system also requires an event translator, translating input and output events to boolean vectors that Lutess can handle. We have recently shown [15] that this translator can be semi-automatically built assuming that the software architecture of the interactive system is along PAC-Amodeus [18] and developed using the ICARE component-based environment [5, 6]. In this study [15], we showed that the translator between Lutess and an interactive system can be built semi-automatically having some knowledge about the executable program and in our case the ICARE events exchanged between the ICARE components. Such a study can be done in the context of another development environment: our approach for testing multimodal input interaction is not dependent on a particular development environment (black box testing), as opposed to the formal approach for testing that we described in [11], where we relied on the internal ICARE component structure (white box testing). Indeed in [11], our goal was to test the ICARE components corresponding to the fusion mechanism.
2. *The Lustre specification of the test oracle*: this specification describes the properties to be checked. Properties may be related to functional or multimodal interaction requirements. Functional requirements are expressed as properties independent of the modalities. Multimodal interaction requirements are expressed as properties on event sequences considering various modalities. We focus on the CARE and temporal properties described in Section 2. For instance, a major issue is the fusion mechanism [18], which combines input events along various modalities to determine the associated command. This mechanism relies on a temporal window (see

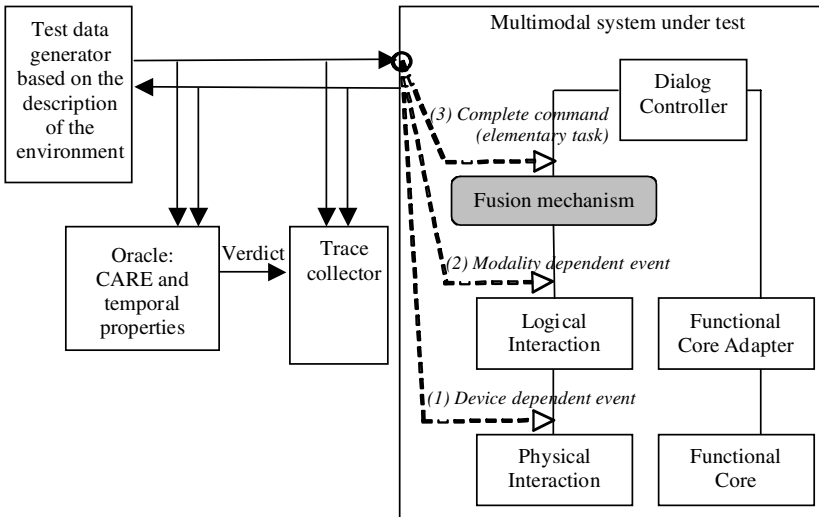
Section 2) within which the users' events occur. For example, when two modalities are used in a complementary or redundant way, the resulting events are combined if they occur in the same temporal window; otherwise, the events are processed independently.

3. *The Lustre specification of the behaviour of the external environment of the system:* from this specification, test data as sequences of users' events are randomly generated. In the case of context-aware systems, in addition to a non-deterministic specification of the users' behaviour, elements specifying the variable physical context can be included. Moreover, additional specifications (operational profiles, behavioural patterns) make it possible to use different generation strategies.

In the following three sections, we further detail each of these three points, respectively, the connection, the oracle and the test data generation based on the specification of the environment.

### 3.2.2 Connection between Lutess and the Interactive Multimodal System

Testing a multimodal system requires connecting it to Lutess, as shown in Fig. 3. To do so, the level of abstraction of the events exchanged between Lutess and the multimodal system must be defined. This level will depend on the application properties that have to be checked and will determine which components of the multimodal system will be connected to Lutess.



**Fig. 3.** Connection between Lutess and a multimodal system organized along the PAC-Amodeus model: three solutions

In order to identify the levels of abstraction of the events exchanged between Lutess and the multimodal system, we must make assumptions on the architecture of the multimodal system being tested. We suppose that the latter is organized along the PAC-Amodeus software architectural model. This model has been applied to the software design of multimodal systems [18]. According to the PAC-Amodeus model,



the structure of a multimodal system is made of five main components (see Fig. 3) and a fusion mechanism performing the fusion of events from multiple modalities. The Functional Core implements domain specific concepts. The Functional Core Adapter serves as a mediator between the Dialog Controller and the domain-specific concepts implemented in the Functional Core. The Dialog Controller, the keystone of the model, has the responsibility for task-level sequencing. At the other end of the spectrum, the Logical Interaction Component acts as a mediator between the fusion mechanism and the Physical Interaction Component. The latter supports the physical interaction with the user and is then dependent on the physical devices. Since our method focuses on testing multimodal input interaction, three PAC-Amodeus components are concerned: the Physical and Logical Interaction Components as well as the fusion mechanism. By considering the PAC-Amodeus components candidates to receive input events from Lutess, we identify three levels of abstraction of the generated events:

1. Simulating the Physical Interaction Component: generated events should be sent to the Logical Interaction Component. In this case, Lutess should send low-level device dependent event sequences to the multimodal system like selections of buttons using the mouse or character strings for recognized spoken utterances.
2. Simulating the Physical and Logical Interaction Components: generated events sent to the fusion mechanism should be modality dependent. Examples include <mouse, empty trash> or <speech, empty trash>.
3. Simulating the fusion mechanism: generated events should correspond to complete commands, independent of the modalities used to specify them, for instance <empty trash>.

Since we aim at checking the CARE and temporal properties of multimodal interaction and the associated fusion mechanism, as explained in Section 2, the second solution has been chosen: the test data generated by the Lutess test generator are modality dependent event sequences.

### 3.2.3 Specification of the Test Oracles

The test oracles consist of properties that must be checked. Properties may be related to functional and multimodal interaction requirements. Examples of properties related to functional requirements are provided in Section 4. In this section we focus on multimodality-related requirements and consider the CARE and temporal properties defined in Section 2: we show that they can be expressed as Lustre expressions and then can be included in an automatic test oracle (see [16] for a preliminary study on this point).

#### *Equivalence:*

Two modalities  $M_1$  and  $M_2$  are equivalent w.r.t. a set  $T$  of tasks, if every task  $t \in T$  can be activated by an expression along  $M_1$  or  $M_2$ . Let  $E_{AM_1}$  be an expression along modality  $M_1$  and let  $E_{AM_2}$  be an expression along  $M_2$ .  $E_{AM_1}$  or  $E_{AM_2}$  can activate the task  $t_i \in T$ . Therefore, equivalence can be expressed as follows:

$$\text{OnceFromTo} (E_{AM_1} \text{ or } E_{AM_2}, \text{ not } t_i, t_i)$$

We recall (see Section 3.1) that  $\text{OnceFromTo}(A, B, C)$  specifies that property  $A$  must hold at least once between the instants where events  $B$  and  $C$  occur. Therefore, the above generic property holds if at least one of the expressions  $E_{AM1}$  or  $E_{AM2}$  has been set before the action  $t_i$  occurs.

*Redundancy and Complementarity:*

In order to define the two properties Redundancy and Complementarity that describe combined usages of modalities, we need to consider the use over time of a set of modalities. For both Redundancy and Complementary, the use of the modalities may occur within a temporal window  $Tw$ , that is, a time interval. As Lustre does not provide any notion of physical time, to specify the temporal window, we consider  $C$  to be the duration of an execution cycle (time between reading an input and writing an output). The temporal window is then specified as the number of discrete execution cycles:

$$N = Tw \text{ div } C.$$

Two modalities  $M_1$  and  $M_2$  are redundant w.r.t. a set  $T$  of tasks, if every task  $t \in T$  is activated by an expression  $E_{AM1}$  along  $M_1$  and an expression  $E_{AM2}$  along  $M_2$ . The two expressions must occur in the same temporal window  $Tw$ :  $\text{abs}(\text{time}(E_{AM1}) - \text{time}(E_{AM2})) < Tw$ . Considering  $N = Tw \text{ div } C$ , and the task  $t_i \in T$ , the Lustre expression of the redundancy property is the following one.

```
Implies (t_i,
         abs(lastOccurrence(E_AM1) - lastOccurrence(E_AM2)) <= N
         and atMostOneSince(t_i, E_AM1) and atMostOneSince(t_i, E_AM2))
```

where:

- $\text{Implies}(A, B)$  is the usual logic implication (not  $A$  or  $B$ ).
- $\text{lastOccurrence}(A)$  returns the latest instant that  $A$  occurred.
- $\text{atMostOneSince}(A, B)$  is true when at most one occurrence of  $A$  has been observed since the last time that  $B$  has been true.

Two modalities are used in a complementary way w.r.t. a set  $T$  of tasks, if every task  $t \in T$  is activated by an expression  $E_{AM1}$  along  $M_1$  and an expression  $E_{AM2}$  along  $M_2$ . The two expressions must occur in the same temporal window  $Tw$ . We therefore get the same Lustre expression as for redundancy. Indeed Complementarity and Redundancy correspond to the same use over time of modalities and the difference relies on the semantic of the expressions along the modalities. While complementarity implies expressions with complementary meaning for the task considered (e.g. speech command "open" while clicking on an icon using the mouse), redundancy involves expressions conveying the same meaning (e.g., speech command "open paper.doc" while double-clicking on the icon of the file named paper.doc using the mouse). The meaning of the conveyed expressions is defined by the Lutes user (i.e. tester). Consequently, the same oracle is defined for redundancy and complementarity.

### 3.2.4 Strategies for Generating Test Data

The automatic test input generation is a key issue in software testing. In the particular case of interactive systems, such a generation relies on the ability to model various users' behaviours and to automatically derive test data compliant with the models. Lutes provides several generation facilities and underlying models.

*Constrained Random Generation:*

The user is represented by a set of invariants specifying all its possible behaviours. The latter are randomly generated on an equal probability basis. More precisely, at every execution step, one of the input vectors satisfying the invariants will be fairly chosen among all the possible vectors.

*Operational profiles:*

Although the random generation is operated in a fair way, the resulting behaviour is seldom realistic. To cope with this problem, operational profiles can be defined by means of occurrence probabilities associated with user actions [19]. Occurrence probabilities can be conditional (that is, they will be taken into account during the test data generation only when a user-specified condition holds) or unconditional. Random generation is performed w.r.t. these probabilities.

An interesting feature of this generation mode is that it makes possible to issue events in the same temporal window and, hence, to check the fusion capabilities of a multimodal system. As we have shown in [19], one has to associate with the input events a probability computed from the temporal window duration to ensure that events will occur in the same temporal window. Let  $N$  be the number of discrete execution cycles corresponding to the full duration of the temporal window (computed as in Section 3.2.3). For an input event to occur within the temporal window, its occurrence probability must be greater or equal to  $1/N$ . For example, to specify that  $A$  and  $B$  will both be issued in that order in the same temporal window, we can write:

```
proba(A, 1/N, after(B) and pre always_since(not A, B));
```

Indeed, this formula means that if at least a  $B$  event has occurred in the past and if no  $A$  event occurred since the last  $B$  occurrence, then the  $A$  occurrence probability is equal to  $1/N$ . Since the temporal window starts at the last occurrence of  $B$  and lasts  $N$  ticks,  $A$  will very probably occur at least once before the end of the window.

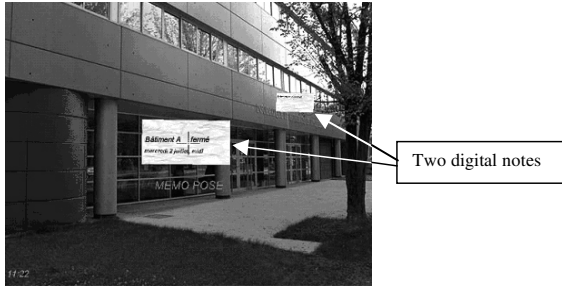
*Behavioural patterns:*

Behavioural patterns make possible to partially specify a sequence of user actions. As opposed to the above operational profile-based generation mode, a behavioural pattern involves several execution instants. Behavioural patterns enable the description of executions that may not be easy to attained randomly and are hard to specify with occurrence probabilities. The random test input generation takes into account this partial specification of user actions.

## 4 Illustration: The Memo Multimodal System

Memo [4] is an input multimodal system aiming at annotating physical locations with digital post it-like notes. Users can drop a note to a physical location. The note can then be read/carried/removed by other mobile users.

A Memo user is equipped with a GPS and a magnetometer enabling the system to compute her/his location and orientation. The memo user is also wearing a head mounted display (HMD). Its semi-transparency enables the fusion of computer data (the digital notes) with the real environment as shown in Fig. 4.



**Fig. 4.** A sketched view through the HMD: The Memo mobile user is in front of the computer science teaching building at the University of Grenoble and can see two digital notes

In [17], we fully illustrate our testing method by considering the test of Memo using an operational profile-based approach for generating the test data. In order to illustrate all the strategies for generating test data, we consider here three tasks, namely "get a post-it", "set a post-it" and "remove a post-it" with Memo. For the manipulation of Memo notes, the mobile user can get a note that will then be carried by her/him while moving and be no longer visible in the physical environment. The user can carry one note at a time. As a consequence if s/he tries to get a note while already carrying one note, the action will have no effect. S/he can set a carried note to appear at a specific place. Issuing the set command without carrying a note has no effect. To perform the three tasks "get", "set" and "remove", the user has the choice between three equivalent modalities: issuing voice commands, pressing keys on the keyboard or clicking on mouse buttons. A command "get" or "remove" specified using speech, keyboard and mouse is applied to the notes that the user is looking at (i.e., the notes close to her/him). Memo can also be set to support redundant usage of modalities. Using Memo, speech, keyboard and mouse commands can be issued in a redundant way. For example, the user can use two redundant modalities, voice and mouse commands, for removing a note: the user issues the voice command "remove" while pressing the mouse button. Because the corresponding expressions are redundant and the two actions (speaking and pressing) produced nearly in parallel or close in time, the command will be executed and as a result the corresponding note will be deleted. If the two "remove" actions were not produced close in time, there is no redundancy detected and the remove command will therefore not be executed.

In the following sections and considering the three tasks "get", "set" and "remove", we illustrate our method by first explaining the connection between Lutess and Memo. We then define the test oracle for Memo and finally explain how we automatically generate test data using different strategies.

#### 4.1 Connection between Lutess and Memo

The connection between Memo and Lutess is made by a Java class, MemoLutess, in charge of translating Lutess outputs into Memo inputs and vice-versa. As explained in Section 3.2.1, we developed a method for semi-automatically generating this translator that we describe in [15] as an extension of the ICARE platform. For Memo, the code has been written manually without the ICARE platform. So the class MemoLutess has been

written by hand. This class includes a constructor, creating a new instance of a Memo system. A main method creates a new instance of MemoLutess and links it to Lutess.

```
/* Main method */
static public main(String[] args) {
    MemoLutess m = new MemoLutess();
    m.connectLutess();
}
```

The connectLutess method is made of an infinite loop which (1) reads a sequence of inputs issued by the Lutess test data generator and (2) sends the corresponding events to the Memo system; then, it (3) waits for Memo to execute the resultant commands, (4) obtains the new Memo state (5) and sends the computed output vector to the Lutess generator.

```
/* Main interaction loop */
void connectLutess() {
    while (true) {
        readInputs(); // Read test inputs
        memoApp.sendEvents() ; // Send corresponding events to Memo
        wait(N); // wait N ms for Memo to react
        memoApp.getState() ; // Get the new state of Memo
        writeOutputs();} // Write outputs
```

As explained in Section 3.2.2, the level of abstraction is set at the modality level. Generated events are hence received by the fusion component of Memo. For the "get" "set" and "remove" tasks, the following events are involved in the interaction:

- *Localization* is a boolean vector which indicates the user's movements along the x, y and z axes. For instance, Localization[xplus]=true means that the user's x-coordinate increases. Similarly *Orientation* is a boolean vector, which indicates the changes in the user's orientation. For instance, Orientation[pitchplus] indicates that the user is bending one's head.
- *Mouse*, *Keyboard* and *Speech* are boolean vectors corresponding to a "get", "set" or "remove" command specified using speech, keyboard or mouse. For instance, Mouse[get] indicates that the user has pressed the mouse button corresponding to a "get" command.

The state of the Memo system is observed through four boolean outputs:

- *memoSeen*, which is true when at least one note is visible and close enough to the user to be manipulated,
- *memoCarried*, which is true when the user is carrying a note,
- *memoTaken*, which is true if the user has get a note during the previous action-reaction cycle,
- *memoSet*, which is true if the user has set a carried note to appear at a specific place during the previous cycle,
- *memoRemoved*, which is true if the user has removed a note during the previous cycle.

## 4.2 Memo Test Oracle

The test oracle consists of the required Memo properties. First we consider functional properties. For example the state of Memo cannot change except by means of suitable

input events: between the instant the user is seeing a note and the instant there is no note in her/his visual field, the user has moved or specified a "get" command.

```
once_from_to((move or cmdget) and pre memoSeen, memoSeen, not memoSeen)
```

Moreover we specify that notes are taken or set only with appropriate commands. For example, after a note has been seen and before it has been taken, a "get" command has to occur at an instant when the note is seen.

```
once_from_to(cmdget and pre memoSeen, memoSeen, memoTaken)
```

Furthermore if a note is carried, then a "get" command has previously occurred.

```
once_from_to(cmdget and pre memoSeen, not memoCarried, memoCarried)
```

In addition to functional properties, multimodality-related properties are specified in the test oracle, as explained in Section 3.2.3. For instance, to check that the task `memoTaken` takes place only after the occurrence of the redundant expressions `Mouse[get]` and `Speech[get]`, we should write the following test oracle:

```
node MemoOracle(-- application inputs and outputs
)
  returns (propertyOK:bool);
let
  propertyOK =
    Implies (memoTaken,
      abs (lastOccurrence(Mouse[get]) -
        lastOccurrence(Speech[get])) <= N
      and
      atMostOneSince(memoTaken , Mouse[get]) and
      atMostOneSince(memoTaken , Speech[get]));
tel
```

The above node states that (1) `memoTaken` occurs only when (1) `Mouse[get]` and `Speech[get]` occur in the same temporal window (of duration `N`) and that (2) in that case `memoTaken` occurs only once.

### 4.3 Memo Test Input Generation

#### 4.3.1 Modelling the Environment and the Users' Behaviour

Input data are generated by Lutess according to formulas defining assumptions about the external environment of Memo, i.e. the users' behaviour. We here describe actions that the user cannot perform. For example the user cannot move along an axis in both directions at the same time. The corresponding formulas are:

```
not (Localization[xminus] and Localization[xplus])
not (Localization[yminus] and Localization[yplus])
not (Localization[zminus] and Localization[zplus])
```

Similarly, we also specify by three formulas that the user cannot turn around an axis in both directions at the same time.

Moreover, Lutess sends data to Memo at the modality level. Since there is one abstraction process per modality, only one data along a given modality can therefore be sent at a given time. The commands "get", "set" and "remove" can be performed using speech, keyboard or mouse. We therefore get the following formulas<sup>1</sup>:

```
AtMostOne(3, Mouse) ; AtMostOne(3, Keyboard) ; AtMostOne(3, Speech)
```

<sup>1</sup> Mouse is a boolean table of three elements indexed by "get", "set" and "remove": `AtMostOne(3, Mouse)` means that at most one of the elements of the table is true.

### 4.3.2 Guiding the Test Data Generation

#### *Random generation and operational profiles:*

A random simulation of the users' actions results in sequences in which every input event has the same probability to occur. This means, for instance, that Localization[xminus] will occur as many times as Localization[xplus]. As a result, the users' position will hardly change. To test Memo in a more realistic way, the data generation can be guided by means of operational profiles (set of conditional or unconditional probabilities definition). Unconditional probabilities are used to force the simulation to correspond to a particular case, for example that the user is turning one's head to the right:

```
proba( (Orientation[yawminus], 0.80), (Orientation[yawplus], 0.01),
      (Orientation[pitchminus], 0.01), (Orientation[pitchplus], 0.01),
      (Orientation[rollminus], 0.01), (Orientation[rollplus], 0.01)).
```

Conditional probabilities are used, for instance, to specify that a "get" command has a high probability to occur when the user has a note in her/his visual field (close enough to be manipulated):

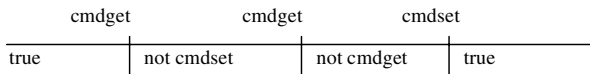
```
proba( (Mouse[get], 0.8, pre memoSeen),
      (Keyboard[get], 0.8, pre memoSeen), (Speech[get], 0.8, pre memoSeen))
```

The following expression states that, when there is no note visible, the user will very probably move:

```
proba( (Orientation[yawminus], 0.9, not pre memoSeen), ... ).
```

#### *Behavioural patterns:*

A pattern is a sequence of actions and conditions that should hold between two successive actions. During the random test data generation, inputs matching the scenario have a higher occurrence probability. Let us consider the scenario corresponding to the sequence of commands presented in Fig. 5: the user performs twice the "get" command, then a "set" command. The scenario also specifies that in between the first two "get" commands, the user does not perform a "set" command and similarly between the two "get" and "set" commands, no "get" command.



**Fig. 5.** An example of a scenario for guiding the generation of test data

This scenario can be described in Lutess as follows:

```
cond(
    (Mouse[get] or Keyboard[get] or Speech[get]),
    (Mouse[get] or Keyboard[get] or Speech[get]),
    (Mouse[set] or Keyboard[set] or Speech[set]));
intercond(
    true,
    not(Mouse[set] or Keyboard[set] or Speech[set]),
    not(Mouse[get] or Keyboard[get] or Speech[get]),
    true);
```

Let us consider a second scenario. It describes a redundant usage of two modalities: mouse and speech. The scenario starts in a state where notes are visible (`pre memoSeen`). The user first takes one note in a redundant way, with mouse and speech at the same instant. The user then removes a second note by using again mouse and speech in a redundant way but at two different instants belonging to the same temporal window. The scenario is expressed as follows:

```
cond(
    pre memoSeen and (Speech[get] and Mouse[get]) and
    not (Speech[remove] or Mouse[remove]),
    Mouse[remove] and not Speech[remove],
    Speech[remove] and not Mouse[remove]);
intercond(
    true,
    not Speech[remove],
    not Mouse[remove]);
```

[line 1]	-	-	-	-	Se	-	-	-
[line 2]	mG	-	sG	-	Se	Car	Tak	-
[line 3]	-	mR	-	-	Se	Car	-	-
[line 4]	-	-	-	sR	Se	Car	-	-
[line 5]	-	-	-	-	-	Car	-	Rem

**Fig. 6.** An excerpt from a Memo trace

Fig. 6<sup>2</sup> shows an extract of trace which matches this second scenario. In this trace, the first line contains the event `memoSeen` (*Se*), implying that one or several notes are close to the user. In the second line, the two simultaneous events `Mouse[Get]` and `Speech[Get]` (*mG* and *sG*) cause one note to be taken (event *Tak* line 2). `memoSeen` is still set, which means that another note is visible. Lines 3 and 4 contain the events `Mouse[remove]` and `Speech[remove]` (*mR* and *sR*), which cause the visible note to be removed (event *Rem* line 5) since the two events (*mR* and *sR*) belong to the same temporal window.

## 5 Conclusion and Future Work

In this article, we have presented a method for automatically testing multimodal systems based on Lutess, a testing environment originally designed for synchronous software. Multimodality is addressed through the software properties that are checked: the CARE and temporal properties. Testing the satisfaction of the CARE and temporal properties with Lutess requires (1) expressing the properties in Lustre to build a test oracle and (2) generating adequate test input data. We have shown that the expression of the CARE and temporal properties in Lustre is possible, since the language is a temporal logic of the past and makes it possible to specify constraints on event sequences. The test data generation relies on a users' model including invariants and guiding directives (i.e. operational profiles, behavioural patterns). We have shown that by specifying operational profiles it is possible to generate test data corresponding to the combined usage of modalities, and that scenarios are also useful for the expression of functional properties.

<sup>2</sup> mG, mR, sG, SR stand for `Mouse[get]`, `Mouse[remove]`, `Speech[get]` and `Speech[remove]`. Se, Car, Tak, Rem stand for `memoSeen`, `memoCarried`, `memoTaken`, `memoRemoved`.



In future work, we will explore further the guide-types for generating the test data, and in particular behavioural patterns that correspond to usability scenarios. To do so, we plan to use information from the task analysis in order to define the behavioural patterns. This work will be done in the context of our platform ICARE-Lutes that supports a semi-automatic generation of the translators between Lutes and the multimodal system developed using ICARE. Since an ICARE diagram is defined for a given task, we will first link our ICARE platform with a task analysis tool such as CTTE [20]. We will then exploit the task tree for defining behavioural patterns used for guiding the test. Extending our ICARE-Lutes platform in order to be connected to a task analysis tool will lead us to define an integrated platform from task to concrete multimodal interaction for designing, developing and testing multimodal systems.

## Acknowledgments

Many thanks to G. Serghiou for reviewing the paper. This work is partly funded by the French National Research Agency project VERBATIM (RNRT) and by the Open-Interface European FP6 STREP focusing on an open source platform for multimodality (FP6-035182).

## References

1. Benveniste, A., Caillaud, B., Le Guernic, P.: From synchrony to asynchrony. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 162–177. Springer, Heidelberg (1999)
2. Barnard, P., May, J.: Cognitive Modelling for User Requirements. *Computers, Communication and Usability: Design issues, research and methods for integrated services*, pp. 101–146. Elsevier, Amsterdam (1993)
3. Blandford, A., Young, R.: Developing runnable user models: Separating the problem solving techniques from the domain knowledge. In: Proc. of HCI 1993, People and Computers VIII, pp. 111–122. Cambridge University Press, Cambridge (1993)
4. Bolt, R.: Put That There: Voice and Gesture at the Graphics Interface. In: Proc. of SIGGRAPH 1980, pp. 262–270. ACM Press, New York (1980)
5. Bouchet, J., Nigay, L., Ganille, T.: ICARE Software Components for Rapidly Developing Multimodal Interfaces. In: Proc. of ICMI 2004, pp. 251–258. ACM Press, New York (2004)
6. Bouchet, J., Nigay, L.: ICARE: A Component-Based Approach for the Design and Development of Multimodal Interfaces. In: Proc. of CHI 2004 extended abstract, pp. 1325–1328. ACM Press, New York (2004)
7. Coutaz, J., Nigay, L., Salber, D., Blandford, A., May, J., Young, R.: Four Easy Pieces for Assessing the Usability of Multimodal Interaction: The CARE properties. In: Proc. Of INTERACT 1995, pp. 115–120. Chapman et Hall, Boca Raton (1995)
8. d'Ausbourg, B.: Using Model Checking for the Automatic Validation of User Interfaces Systems. In: Proc. of DSV-IS 1998, pp. 242–260. Springer, Heidelberg (1998)
9. du Bousquet, L., Ouabdesselam, F., Richier, J.-L., Zuanon, N.: Lutes: a Specification Driven Testing Environment for Synchronous Software. In: Proc. of ICSE 1999, pp. 267–276. ACM Press, New York (1999)

10. Duke, D., Harrison, M.: Abstract Interaction Objects. In: Proc. of Eurographics 1993, pp. 25–36. North Holland, Amsterdam (1993)
11. Dupuy-Chessa, S., du Bousquet, L., Bouchet, J., Ledru, Y.: Test of the ICARE platform fusion mechanism. In: Gilroy, S.W., Harrison, M.D. (eds.) DSV-IS 2005. LNCS, vol. 3941, pp. 102–113. Springer, Heidelberg (2006)
12. Griffeth, N., Blumenthal, R., Gregoire, J.-C., Ohta, T.: Feature Interaction Detection Contest. In: Proc. of Feature Interactions in Telecommunications Systems V, pp. 327–359. IOS Press, Amsterdam (1998)
13. Halbwachs, N.: Synchronous programming of reactive systems, a tutorial and commented bibliography. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 1–16. Springer, Heidelberg (1998)
14. Ishii, H., Ullmer, B.: Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. In: Proc. of CHI 1997, pp. 234–241. ACM Press, New York (1997)
15. Jourde, F., Nigay, L., Parissis, I.: Test formel de systèmes interactifs multimodaux: couplage ICARE – Lutess. In: Proc. of 19èmes Journées Internationales du génie logiciel (in french)
16. Madani, L., Parissis, I., Nigay, L.: Testing the CARE properties of multimodal applications by means of a synchronous approach. In: IASTED Int'l Conference on Software Engineering, Innsbruck, Austria (February 2005)
17. Madani, L., Oriat, C., Parissis, I., Bouchet, J., Nigay, L.: Synchronous Testing of Multimodal Systems: An Operational Profile-Based Approach. In: Proc. of Int'l Symposium on Software Reliability Engineering (ISSRE 2005), pp. 325–334. IEEE Computer Society, Los Alamitos (2005)
18. Nigay, L., Coutaz, J.: A Generic Platform for Addressing the Multimodal Challenge. In: Proc. of CHI 1995, pp. 98–105. ACM Press, New York (1995)
19. Ouabdesselam, F., Parissis, I.: Constructing Operational Profiles for Synchronous Critical Software. In: Proc. of Int'l Symposium on Software Reliability Engineering (ISSRE 1995), pp. 286–293. IEEE Computer Society, Los Alamitos (1995)
20. Mori, G., Paterno, F., Santoro, C.: CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. In: IEEE Transactions on Software Engineering, pp. 797–813 (August 2002)
21. Palanque, P., Bastide, R.: Verification of Interactive Software by Analysis of its Formal Specification. In: Proc. of INTERACT 1995, pp. 191–197. Chapman et Hall, Boca Raton (1995)
22. Parissis, I., Ouabdesselam, F.: Specification-based Testing of Synchronous Software. In: Proc. of ACM SIGSOFT, pp. 127–134. ACM Press, New York (1996)
23. Paterno, F., Faconti, G.: On the Use of LOTOS to Describe Graphical Interaction. In: Proc. of HCI 1992, pp. 155–173. Cambridge University Press, Cambridge (1992)
24. Thimbleby, H., Cairns, P., Jones, M.: Usability Analysis with Markov Models. ACM Transactions on Computer Human Interaction 8(2), 99–132 (2001)
25. Vernier, F., Nigay, L.: A Framework for the Combination and Characterization of Output Modalities. In: Palanque, P., Paternó, F. (eds.) DSV-IS 2000. LNCS, vol. 1946, pp. 32–48. Springer, Heidelberg (2001)

# Knowledge Representation Environments: An Investigation of the CASSMs between Creators, Composers and Consumers

Ann Blandford<sup>1</sup>, Thomas R.G. Green<sup>2</sup>, Iain Connell<sup>1</sup>, and Tony Rose<sup>3</sup>

<sup>1</sup> UCL Interaction Centre, University College London, Remax House,  
31-32 Alfred Place London WC1E 7DP, U.K.

A.Blandford@ucl.ac.uk

<http://www.ucl-icc.ucl.ac.uk/annb/>

<sup>2</sup> University of Leeds, U.K.

<sup>3</sup> System Concepts Ltd., U.K.

**Abstract.** Many systems form ‘chains’ whereby developers use one system (or ‘tool’) to create another system, for use by other people. For example, a web development tool is created by one development team then used by others to compose web pages for use by yet other people. Little work within Human–Computer Interaction (HCI) has considered how usability considerations propagate through such chains. In this paper, we discuss three-link chains involving people that we term Creators (commonly referred to as designers), Composers (users of the tool who compose artefacts for other users) and Consumers (end users of artefacts). We focus on usability considerations and how Creators can develop systems that are both usable themselves and also support Composers in producing further systems that Consumers can work with easily. We show how CASSM, an analytic evaluation method that focuses attention on conceptual structures for interactive systems, supports reasoning about the propagation of concepts through Creator–Composer–Consumer chains. We use as our example a knowledge representation system called Tallis, which includes specific implementations of these different perspectives. Tallis is promoting a development culture within which individuals are empowered to take on different roles in order to strengthen the ‘chain of comprehension’ between different user types.

**Keywords:** Usability evaluation methods, CASSM, design chains.

## 1 Introduction

It is widely recognised that there are many stakeholder groups in any design project, typically including managers, purchasers, end users and developers. Approaches such as Soft Systems Methodology [5] encourage an explicit consideration of these different stakeholder groups in design. However, when it comes to considering usability, the focus narrows immediately to the ‘end users’ of the system under consideration. For example, most classic evaluation techniques, such as Heuristic Evaluation [13] and Cognitive Walkthrough [16] focus on what the user will experience in terms of their tasks and the feedback received from the system. Norman [15] discusses the

relationship between designer and user in terms of the ‘designer’s conceptual model’ – an understanding of the system that has to be communicated from designer to end user via the interface. In all these cases, the focus remains on a single ‘system’ that is being designed. No work that we are aware of extends this perspective. This is perhaps not surprising, since established usability-oriented analysis techniques focus attention on user tasks and the procedures users need to follow to complete those tasks. The users of different interfaces experience interactions with different properties that are not readily related to each other. In this paper, we explicitly consider different systems within a development chain, focusing in particular on three groups of stakeholders that we term Creators, Composer and Consumers ( $C^3$ ) – namely the Creators of tools that can be used by Composers to construct products for Consumers to use.

As an early example, consider the design of web pages using a web composition tool such as Dreamweaver<sup>®</sup>. It is recognised good practice for all pictures on web pages to be supplemented by “ALT” text that describes the content of the picture to improve ease of use for users with limited vision. If the web composition tool makes it easy for Composers to include ALT information, and makes it obvious at the time of including a picture that ALT text should be added, then the resulting web page will be more usable. The Creator of the web development tool can improve the likely usability of web pages produced by a Composer if the Creator is aware of the potential needs of the Consumer.

We argue that a declarative approach to evaluation can yield a more insightful evaluation of such chains than a procedural one, for reasons presented below. The declarative approach we have adopted is CASSM [4], a technique for usability evaluation that is based on identifying the concepts with which users are working and those implemented within a system. This approach has helped to draw out relationships between different systems within a ‘chain’ of products that have (typically) different users and different interfaces. The approach is exemplified with a system, Tallis, for representing clinical guidelines that makes explicit the fact that it has different classes of users who experience different interfaces.

## 1.1 Creators, Composers and Consumers

We do not consider ourselves to have invented  $C^3$  chains; indeed, they are a widespread phenomenon. Nevertheless, we are not aware of prior work that has discussed such chains within the HCI literature, or considered usability in terms of chains. Therefore, we start by briefly discussing some examples of  $C^3$  chains – namely website creation tools, programming development environments and online library building applications.

Website creation tools such as Dreamweaver<sup>®</sup> allow Composers to create, edit and manipulate html and other mark-up language code prior to uploading finished website code to a server. The role of Creators is not only to make the programming and editing environment easy to use but also to facilitate the creation of usable, acceptable web sites (as illustrated with the ALT text example above). The role of Composers is to create and test web pages or sites that are easy and pleasant for the end user to work with. The role of Consumers is to browse, search or otherwise work with the resulting web sites. Thus, the Creators have to understand not only what Composers

will experience, and consider the usability of the composition environment, but also make it easy for Composers to deliver web pages that are well laid out and easy for Consumers to interact with.

The same roles are to be seen in the design and use of program development environments such as NetBeans, an interactive Java environment [13]. The Creators of NetBeans and similar environments provide a tool that will meet the needs of Composers, i.e. Java programmers. Composers write Java programs that should be usable by the Consumers — the people who work with those programs to get a job done. In some cases the roles may become blurred: the same person may create the environment, use it to write programs, and then make use of those programs; nevertheless, there are separate roles depending on which system is the current focus of use.

Chains may stretch further in both directions: a Java programming environment may be written in another programming language, say C++, for which a compiler may be written in some other language—stretching back through assembly code to the instruction set recognised by the hardware. In the other direction, Java programs created in NetBeans, etc, may be used as tools by people who are building other tools. Chains may also branch; for example, web applications are viewed in browsers, and there are often interactions between the application and the browser itself so that the design of both influences the users' experience. This is a factor in the design of Tallis as discussed below, but we do not consider this branching further in this paper.

An example of a tool that extends the development chain is a digital library system, where developers work with software development environments to create a further layer of tools, such as Greenstone [17], with which librarians can create collections of documents to be made available to end users. In a study evaluating the Greenstone digital library software [2], one of the developers commented as follows:

“[There is a] difficulty with the way Greenstone is perceived by different parties. [The developers] see Greenstone very much as a toolset which other folks should 'finish off' to make a good collection. Their conception is that it would be very hard to take Greenstone to a level where a librarian could make a few choices on GUI [Graphical User Interface] and have a reasonable (not to say actively excellent) interface for the library.”

In other words: in the view of the respondent, the Creators of the Greenstone toolset were not recognising their potential role in making it easy for Composers (who typically have little HCI expertise) to construct usable digital libraries for Consumers.

The possibility that a development environment such as NetBeans might be used to construct a digital library tool set like Greenstone, which would in turn be used to develop digital libraries, illustrates the idea that the overall chain might involve more than three groups of designers/users. Here, we only consider  $C^3$  chains. Within a longer chain, the decision as to which people fill the roles of Creator, Composer, and Consumer would depend on where the focus of interest is. In the case of NetBeans, it would be on the development environment and resulting systems, whereas for Greenstone it would be on the development tools and resulting library collections.

Table 1 tabulates the distinction between Creators, Composers and Consumers for these and other systems. In all these cases, there will typically be a development team who create the tool; they may or may not have direct access to their immediate users, the Composers of products. The end users (Consumers) of the product typically have a role where interaction is relatively constrained, with limited scope for changing structures within the product.

**Table 1.** Distinction between Creators, Composers and Consumers for different types of interactive system

Creator of tool	Composer of product	Consumer of product
User Interface Development Environment	Develop interfaces	Use interfaces
Online library tool set (e.g. Greenstone)	Create and manage library collections	Retrieve and display search items
Drawing tool	Create and edit drawings	View and interpret drawings
Website creation tool (e.g. Dreamweaver)	Create and manipulate html and other code, run web pages in a browser	Run website in a browser
Programming development system (e.g. NetBeans)	Create and manipulate code, test programs, run programs	Run programs
Music composition system	Create and edit musical representations	Read, interpret and play music
Word processing system	Create and edit text	Read and interpret text
Game engine	Create new game software	Play game

## 1.2 CASSM and Misfit Analysis

With this understanding of  $C^3$  systems, we turn to consider evaluation of these different systems. Approaches to the evaluation of any interactive system, whether analytic or empirical, based on prototype or working artefact, require from evaluators an insight into the assumptions and expectations held by the intended users of that system. CASSM (Concept based Analysis of Surface and Structural Misfits) is an analytic method which aids the identification of designer-user misfits. Prior to this study, it had only been used in the traditional way, of considering a single interactive system and its users. This study extends the use of CASSM to consider  $C^3$  chains.

In contrast to most evaluation approaches, CASSM does not focus on tasks, but on entities and attributes, and the differences between the system and user models of how entities and their attributes are represented and manipulated at the interface. Previously, we have described how CASSM can identify misfits in systems as diverse as drawing tools and online music libraries [7] and ambulance dispatch [3]. In this paper, we extend the application of CASSM to Tallis [8], a knowledge representation system that exhibits an unusual degree of overlap between the  $C^3$  roles. The CASSM analysis of Tallis allows us to distinguish between the useful and less useful manifestations of this overlap.

In a CASSM analysis, we make an explicit distinction between the representation embodied within an interactive system and that understood by the users of that system. Earlier papers [1,6] show how we characterise this distinction in terms of a taxonomy of User, Interface and System properties, where the various concepts (entities and attributes) which result from the CASSM analysis are depicted as Present or Absent from the System models, and Present, Absent or Difficult to apprehend for the User or via the Interface. (See [4] for tutorial and worked examples).

In its emphasis on objects rather than tasks, CASSM is distinct from other analytic approaches which aim to illuminate the differences between system and user models. Connell *et al* [6] have contrasted CASSM with Cognitive Walkthrough, whose focus

on goal support at each stage of a task has some similarities with Norman's [15] theory of action (which depicts system-user misfits in terms of the gulfs of execution and evaluation). CASSM can be viewed as focusing more on the conceptual gulfs that Norman [15] discusses between the designer and the user.

## 2 Tallis Composer and Enactor

As noted above, knowledge representation tools also exemplify the  $C^3$  chain. Composers create, manipulate and edit a rule-based set of choices and actions, presented to Consumers via an interface. Tallis is a knowledge representation tool that is being developed with a view to producing and disseminating guidelines for clinical practice. It is typically used for modelling clinical diagnosis and treatment processes in the domain of Oncology (the branch of medicine that deals with cancer).

Tallis comprises three interrelated systems: Composer, Tester and Engine. Tester supports debugging, and is not considered in this study. Tallis Composer is a Graphical User interface (GUI) environment which supports the composition of guidelines to aid clinicians in diagnosis and treatment. Guidelines, the output from the Composer, are held in PROforma code [9]. Tallis Engine is the environment in which guidelines are run (or enacted). Enactment takes place in a web browser via a Java virtual machine. In this section, we describe Tallis using an illustrative (non-clinical) guideline for use of the London Underground ticket vending machines. Later sections present the results of a CASSM analysis of Tallis.

This ticket vending machines domain was chosen for two reasons. First, in order to gain experience of using Tallis, it was easier to create and test a guideline in a familiar domain. Second, for the purposes of eliciting Consumer feedback on use of a Tallis guideline, it was easier to recruit a user group who were familiar with the ticketing domain than it would have been to recruit oncology specialists.

### 2.1 Tallis and PROforma

Tallis is a Java implementation of a knowledge representation language called *PROforma*, which is designed to support the publication of clinical expertise [11]. Support takes the form of an expert system which assists patient care through active decision support and workflow management. Fox *et al* [10] describe *PROforma* as an "intelligent agent" language and technology, where agent specification is done by composing tasks into collections of prepared plans. Plans can be enacted sequentially, in parallel, or in response to events.

The *PROforma* decision and plan model offers four classes of task, namely Plans, Decisions, Actions and Enquiries. The root class of this structure is the Keystone, an empty 'placeholder' task. Decisions, Actions and Enquiries may be combined to make up Plans, which themselves consist of other tasks, including other Plans. A combination of tasks so formed represents a *PROforma* guideline, encapsulating one piece of clinical expertise, which may be published on a world wide web repository such as the Open Clinical Knowledge Publishing Collaboratory [12].

Figure 1 shows an extract from the Tallis Composer representation of a sample guideline to support use of London Underground ticket vending machines (TVMs).

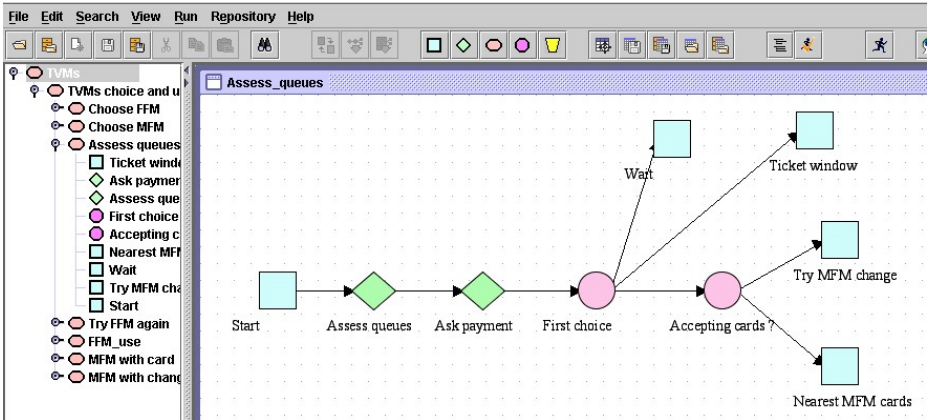


Fig. 1. Extract from the Tallis Composer tool






The left-hand panel of Figure 1 shows part of the TVMs guideline task hierarchy, and the large panel the structure of the task named *Assess\_queues*. The middle part of the toolbar above the panel offers the five *PROforma* tasks (Action , Enquiry , Plan , Decision  and Keystone ), any of which can be inserted into *Assess\_queues* (by drag-and-drop from the toolbar to the task window). Other panels (not shown) allow configuration of the attributes of each task component.

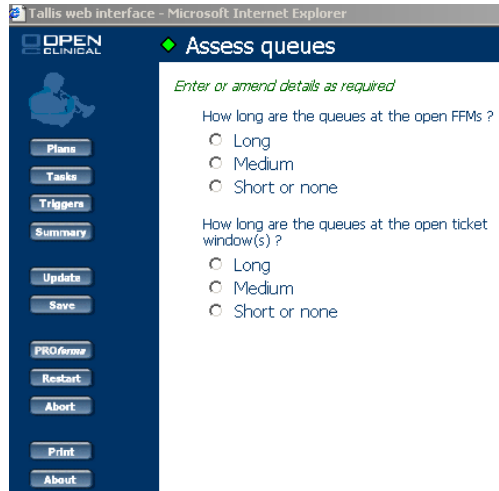
Figure 2 shows the initial result of enacting the above guideline in a web browser using Tallis Engine. The left-hand panel allows the guideline user to inspect certain components of the guideline, including the *PROforma* itself, and to summarise the enactment trail thus far. The guideline may also be restarted or aborted.

## 2.2 Tallis Users

In the Tallis context, *Creators* produce and design the Tallis Composer interface and also the default Tallis Engine interface (Figures 1 and 2; sophisticated Tallis users can tailor the Engine interface to suit the needs of their application). *Creators* also prescribe how the *PROforma* code which results from a Composer session is to be enacted. *Composers* make use of Tallis Composer to produce guidelines (or self-contained guideline fragments) which are encoded in *PROforma* and run via the Engine. *Composers* may also publish guidelines in a Repository. *Consumers* download published guidelines and run them in a web browser using the Engine.

Knowledge representation systems such as Tallis are interesting examples of  $C^3$  systems because the assumptions made by the guideline Composer about Consumer expectations and knowledge are critical to guideline usage, and it is the task of the Creator to make it easy for the Composer to easily generate usable guidelines that match the understanding of Consumers. As noted above, the Engine interface is tailorable, so the challenge might be more appropriately stated as that of producing a good general default that can be readily tailored to particular user groups.





**Fig. 2.** Initial result of enacting a PROforma guideline (created using Tallis Composer) in a web browser (using Tallis Engine). The task being run is Access\_queues.

### 3 CASSM Analysis of Tallis Composer and Engine

This Section describes the result of applying the CASSM approach to Tallis Composer and Engine, and setting out the results using a dedicated tool named Cassata (available from [4]).

An important part of a CASSM analysis is the elicitation of user data. In the case of Tallis, this took complementary forms as described below. In practice, the current culture of working with Tallis meant that some participants spoke from more than one perspective; thus, most of the clinical interviewees feature below in multiple sections.

#### 3.1 Data Collection

One source of data was a detailed diary, kept by the lead researcher, of insights into the experiences of learning Tallis over a period of several weeks. As discussed above, the guideline that was developed represented knowledge about underground ticket purchase. During this time, the researcher worked closely with Tallis Creators to improve their awareness of novice user difficulties and to improve his understanding of the system design.

One of the Creators (i.e. a core member of the Tallis development team) was interviewed about his perceptions of the system. Another of the Creators was recorded while Composing the first part of a guideline for the ticket vending machines. The video protocol so obtained represents an expert view of guideline composition and Tallis Composer use, as well as giving insights into the design philosophy for Tallis. The comparison between his version of the TVMs guideline and the larger but less efficient version initially produced by the researcher was used to probe the differences between expert and novice Composers. Following this comparison, the TVMs guideline intended for Consumer use was re-composed.

Three further Tallis users were interviewed. One was an Oncology clinician who worked closely with the Tallis developers and used Tallis to create and upload sample guidelines to the CRUK repository; he was able to present the views of Creator, as well as Composer and Consumer. The second was a professor of medical informatics who had also made use of Tallis in teaching. The third was a lecturer in Health Informatics who based some of his teaching and student course work around Tallis. All three interviewees were asked about their views of using Tallis Composer and Engine to produce guidelines; with two of them, it was possible to run through sample repository guidelines. In one case the participant demonstrated how he had used Tallis to compose guidelines. In the other, the participant acted as Consumer while running the TVMs guideline, and then inspected the guideline components as Composer. The second and third participants were asked about the wider context of decision support systems, and specifically how Tallis compares with similar systems.

To obtain views of Consumers that were independent of the Composer perspective, five postgraduate HCI students used the ticket vending machine guideline to complete sample ticket-buying scenarios. They were asked about their perceptions of Tallis Engine. Interviews were audio recorded and relevant issues extracted.

## 3.2 Analysis and Results

We present the results according to role; as outlined above, several of the study participants discussed Tallis from multiple perspectives, which we have separated out here. Because several of the interviewees had a clinical or medical informatics background, they were able to talk about their views as Consumers of Tallis enacted guidelines as well as their views as Creators or Composers; therefore, we consider two separate groups of Consumers: those of enacted clinical guidelines and those of the enacted ticket machine guideline.

We have constructed CASSM descriptions of the Tallis Composer and Tallis Engine to highlight user–system misfits. These have been constructed by working through interview transcripts and system descriptions to identify the core user and system concepts. On the user side, contextual information from transcripts has been used to determine whether those concepts are present in the user’s conceptual model of the system, whether they pose user difficulties or whether they are absent from the user’s conceptualisation. The user’s conceptualisation will typically include both system (device) concepts and ones pertaining specifically to the domain in which they are working. On the interface and system side, system descriptions have been used to determine whether concepts are represented at the interface and in the underlying system model. For every concept, where possible, further data has been used to determine how easily actions can be performed to change the state of the concept (e.g. creating new entities or changing attributes). Where this data has not been available, we have entered ‘not sure’ in the CASSM table. More details of conducting a CASSM analysis are available from [4].

### 3.2.1 Creators

One of the interviewees from Cancer Research UK described the system as follows:

“what we are looking [at] is how to provide decision support, which will be a core of this project, over the treatment of the patient, from diagnosis until follow-up treatment and everything, so basically this is ... there are a lot of

other things apart from decision support, like, erm, automatic enactment of other tasks, and lots of other things, but the core part of it is decision support for clinicians, and it will also record all data, data entry.” [taken from transcript of interview]

Another compared Tallis to a flow-chart representation of clinical care pathways: the flow chart representation “has some of the same high-level goals and ‘spin’, and that is an approach that is very common. It’s an importantly different approach, because the guidelines are not enactable. They cannot be created by clinicians and then enacted by others. There is no active decision support.” [taken from handwritten notes of interview]

Thus, from a Creator perspective, Tallis is a system that supports the development and use of clinical guidelines, with important features such as active support for decision making and integration with other clinical tasks within the overall patient care pathway. One important feature, highlighted in the first of the above extracts, is that Tallis provides the facility to generate an audit trail of clinical decisions in case of any queries about the clinical decision making for a particular patient. Although the developers think of the system used within a clinical context, it is also possible to implement guidelines for other decision making tasks – such as the ticket machine example used within this study.

### 3.2.2 Composers

The following extracts from interviews highlight Composer perspectives on Tallis Composer. These perspectives have formed the basis for the CASSM description presented below. Key ideas built into the CASSM model are highlighted in yellow (or greyscale) within the transcript.

“there are multiple **plans** and **tasks**, and each **plan involves another task**”  
[1<sup>st</sup> interviewee]

“[the guideline] will support **investigations** ... actually this is not the latest version [of Tallis], what have added is **clinical evidence**,” [1<sup>st</sup> interviewee]

“as an editing tool it’s very difficult to keep track of because you don’t have **global view**.” [2<sup>nd</sup> interviewee]

The second interviewee also discussed the challenge of teaching students to work with Tallis. In particular, he highlighted the idea that there are some standard ‘patterns’ of structure within a knowledge representation (typical patterns of components that represent common ways of reasoning) that can be reused when constructing large guidelines, but that students have to construct them from first principles every time:

“Students are asked to consider how they might put a **pathway** through a **set of Tallis components [Plans, Actions, Enquiries, Decisions]**. Getting more than simple ‘asking for information and using that in next decision’ combinations is difficult - we use a **pattern for a Plan that is a query and choice** depending on the answer to that query” [2<sup>nd</sup> interviewee]

The third interviewee talked about what Tallis is not as well what it is, but then repeated many of the concepts enumerated by the first interviewee:

“We don’t get support in Tallis for knowledge representation - Tallis doesn’t have (modelling) tools with which we can build a model (of e.g. a patient) from which statements can be taken. Tallis doesn’t allow you to represent the underlying model of (e.g. a patient). Tallis is not object or entity oriented (but is process or ‘task’ oriented) - you [the guideline creator] have to map decision criteria onto the ‘objects’ provided by Tallis (which are **plans, enquiries, decisions, actions** etc.).” [handwritten notes of 3<sup>rd</sup> interview]

To construct the full CASSM description, we can also take information from a simple system description (extracted from [9]), as follows:

“*PROforma* is a formal knowledge representation language capable of capturing the **structure and content of a clinical guideline** in a form that can be interpreted by a computer. The language forms the basis of a method and a technology for developing and publishing executable clinical guidelines. Applications built using *PROforma* software are designed to support the management of medical procedures and clinical decision making at the point of care.

In *PROforma*, a guideline application is modelled as a set of **tasks** and **data items**. The notion of a task is central - the *PROforma* task model [...] divides from the keystone (**generic task**) into four types: **plans, decisions, actions and enquiries**.

**Plans** are the basic building blocks of a guideline and **may contain any number of tasks** of any type, including other plans. **Decisions** are taken at points where **options** are presented, e.g. whether to treat a patient or carry out further investigations. **Actions** are typically clinical procedures (such as the administration of an injection) which need to be carried out. **Enquiries** are typically requests for further information or data, required before the guideline can proceed.

[...] networks of tasks can be composed that represent plans or procedures carried out over time. In the editor, logical and temporal relationships between tasks are captured naturally by linking them as required with arrows. Any procedural and medical knowledge required by the guideline as a whole or by an individual task is entered using **templates** attached to each **task**.”

These extracts do not define a full model, but are sufficient for an illustrative, sketchy CASSM model, as shown in Table 2.

This CASSM description includes notes of superficial difficulties as highlighted in the interviews: that it is difficult to get an overview of a guideline at the interface, that components (and their linkages) are hard to change once created, and that the idea of a ‘pattern’ of structure is important to some Composers, but is absent from the Tallis Composer environment. This sketchy description does not account for difficulties users might experience in constructing clinical guidelines using the *PROforma* language – that would require a more thorough analysis than is appropriate for the present purpose.

**Table 2.** Entities and attributes for Tallis Composer as extracted from user data of Composers

	Concept	User	Interface	System	Set / create	Change / delete	Notes
E	guideline	present	difficult	present	easy	easy	difficult to get an overview of the guideline
A	evidence	present	present	present	easy	easy	easy for composer, harder for engine
A	investigation	present	present	present	easy	easy	
E	task	difficult	present	present	easy	hard	Also called 'components' and 'objects'.
E	a decision pathway	present	difficult	notSure	notSure	notSure	
E	data item	present	present	present	easy	easy	
E	pattern	present	absent	absent	cant	cant	
E	plan	notSure	present	present	easy	notSure	
A	attributes	notSure	present	present	easy	notSure	
E	action	notSure	present	present	easy	notSure	
A	attributes	notSure	present	present	easy	notSure	
E	enquiry	notSure	present	present	easy	notSure	
A	attributes	notSure	present	present	easy	notSure	
E	decision	notSure	present	present	easy	notSure	
A	attributes	notSure	present	present	easy	notSure	Includes options

### 3.2.3 Consumers: Clinicians

As noted above, most of the clinical interviewees discussed their experiences of Tallis Engine (i.e. the Consumer interface). Their descriptions of Engine included the following from the first interviewee:

“this is - from a **patient’s history**, [...] of breast cancer, and this is **examination of imaging**, of mammogram or ultrasound” [1<sup>st</sup> interviewee]

“this is the first screen which are some information about the **demographics about the patient**. There is a, some more information, and whether the patient has got a previous medical past, if you say yes, then [another part of the dialogue becomes ungreyed out], otherwise it is greyed out; here we can see that the patient is not **pregnant** and the patient has got some **family history** [...] patient has got a **lump which is 30 mm and which is not fixed**” [1<sup>st</sup> interviewee]

“**Interventions**’ [in enacted guidelines] don’t mean anything to clinicians - change to **‘candidates’**, but names of decisions should be captions, not technical names.” [1<sup>st</sup> interviewee]

The same interviewee commented on the experience of working with Engine:

“this process [...] forces the clinician to do a particular **sequence of the task**, which in actual practice is not the case always. [...] But otherwise, for different clinicians, if you take a novice candidate or a clinician who is very junior, this probably is better because it guides the clinician [in] the normal steps. But for a senior clinician, say for a consultant, it’s sometimes irritating, like, he don’t want to go all the stages he already know, so he might go to a particular task” [transcript of 1<sup>st</sup> interviewee]

“sometimes it might inhibit a clinician - the other thing is we cannot go back, like if I enter some details here, and the patient came up with some other details at a later stage, [or] if I forgot to enter the details [earlier], I can’t go back” [transcript of 1<sup>st</sup> interviewee]

The second interviewee commented explicitly about the relationship between the Composer and Engine environments; the following refers to the Engine window:

“Top level presentation is fine - the next level down needs to be ... if the things aren’t boolean statements, and they are just pieces of evidence for and against then it’s not too bad, [but] if they’re things like this, which is a long expression [looking at the Interventions page, after the first pair of enquiry windows] that’s not something I’d want my users - my end users - to see. I’m perfectly happy for my knowledge engineer to see that, as part of the debugging process ... but it doesn’t display boolean combinations well at this point.” [2<sup>nd</sup> interviewee]

**Table 3.** Entities and attributes for Tallis Engine as extracted from user data of clinical users

	Concept	User	Interface	System	Set / Create	Change/ Delete	Notes
E	guideline	present	difficult	present	fixed	fixed	difficult to get an overview of the whole guideline
A	clinical evidence	present	present	present	easy	hard	
A	investigation	present	present	present	easy	hard	
A	intervention	difficult	present	present	easy	hard	"should be called 'candidates'"
E	patient	present	notSure	notSure	easy	easy	
A	"model"	present	absent	absent	cant	cant	
A	details	present	present	notSure	easy	hard	
A	history	present	present	present	easy	hard	
A	demographics	present	present	present	easy	hard	
A	symptoms	present	present	present	easy	hard	
E	treatment	present	present	present	fixed	fixed	
E	care pathway	present	notSure	notSure	notSure	notSure	
E	plan	difficult	present	present	fixed	fixed	
E	task	difficult	present	present	fixed	fixed	
E	trigger	difficult	present	present	fixed	fixed	
E	PROforma	difficult	present	present	fixed	fixed	
E	evidence	present	present	present	easy	notSure	
A	representation	difficult	present	present	fixed	fixed	
E	decision process	present	present	present	notSure	notSure	
E	decision outcome	present	present	present	notSure	indirect	
E	decision /data record	notSure	difficult	present	indirect	cant	

The third interviewee compared Tallis to flowchart descriptions of clinical guidelines:

“Flow-chart representations [of guidelines] might be better than a Tallis representation (you just have to use your eyes to follow it). However,

representations involving timelines (e.g. **care pathways**) might need the additional complexity of systems such as Tallis.” [notes from 3<sup>rd</sup> interview]

These extracts, together with reference to the Engine environment (as illustrated in Figure 2), have been used to construct the CASSM description shown in Table 3. This is not instantiated to a particular clinical problem (e.g. the diagnosis and treatment of breast cancer), but is a general model of clinical guideline use.

This shows more substantial likely user difficulties than the Composer environment; users are expected to work with concepts (such as ‘intervention’, ‘plan’ or ‘PROforma’) that are unfamiliar, and of minimal obvious relevance to them in their (clinical) decision making. In addition, while much information is easy to enter, it is difficult to change later, due to the linear model of decision making implemented within Tallis.

### 3.2.4 Consumers: TVMs

Many of the same issues emerge in the findings from the study of ticket machine decision making. The data for the ticket machine Consumers is taken from the implementation and user comments on the Engine guideline produced as part of this study. Extracts from user comments are as follows. In all these cases the extracts are taken from questionnaires completed after the interaction or from the analyst’s notes, and numbers at the beginning indicate which user made the comment. The first set of comments refer, as with the clinical users, to Tallis concepts that are independent of the domain of ticket purchasing:

[1] “Don’t need the **Intervention** screen - it gives information that I already know.”

[5] “Interventions screens look like programming language - had to understand boolean logic to use it - seems like decision-making screen”

[1] “Don’t feel in control - have to follow **path**, can’t make choices that are not offered.”

[5] “Summary [at end] are titles of tracks, not **what I did**. Does not remind you of **overall goal**, nor the tracks you have done. Summary is textual way of showing the **process**, not the overall goal.”

[1] “Can’t use the summary [trail of previously used Tallis entities] to go back to previous stages [in order to do alternative forward routes without having to restart]”

[5] “Not sure if Print will reproduce the complete **decision process** [ie the results of clicking on the + symbols under each decision, or just the **decision itself**].”

Because these users were working with an implemented guideline instantiated to a particular domain, they also referred to domain concepts including the following.

[3] “Adult/Child screen confusing, since 'multiple choice' option comes later”

[3] “Can’t see Family Ticket in **ticket type** selection”

[1] “Can’t do **tickets in advance** [e.g. for specified day which is not today]”

[3] “Machines [or the simulation] don’t tell you the **cheaper route or choices**, etc. ( the one offered may not be the most economical)”

[2] “Need clearer information on **ticket prices** on the [real] machines”

[3] “Tube map [on FFM] does not show where **zones** are, and zones [the concept and the boundaries] are confusing until you learn”

The set of domain concepts users worked with also included several from the task instruction sheet, and which any ticket purchaser works with (such as a ticket!), so these are also included in the CASSM model shown in Figure 4.

**Table 4.** Entities and attributes for Tallis Engine (TVM users)

	Concept	User	Interface	System	Set / Create	Change / Delete	Notes
E	guideline	present	difficult	present	fixed	fixed	difficult to get an overview of the guideline
A	evidence	present	present	present	easy	hard	
A	getting information	present	present	present	easy	hard	
A	intervention	difficult	present	present	easy	hard	"should be called 'candidates'"
E	ticket buying situation	present	notSure	notSure	fixed	fixed	
A	details	present	present	notSure	easy	hard	
E	plan	difficult	present	present	fixed	fixed	
E	action	difficult	present	present	fixed	fixed	
E	trigger	difficult	present	present	fixed	fixed	
E	decision process	difficult	difficult	present	fixed	fixed	
E	decision outcome	present	present	present	bySys	hard	
E	decision /data record	notSure	difficult	present	indirect	cant	
E	ticket	present	difficult	absent	fixed	fixed	
A	type	present	difficult	present	hard	notSure	
A	price	present	difficult	present	easy	notSure	finding cheapest ticket is hard
A	validity date	present	present	present	cant	cant	can only buy for today
E	train	present	absent	absent	fixed	fixed	
E	queue	present	present	notSure	fixed	fixed	
E	payment / money	present	present	notSure	fixed	fixed	
E	zone	present	difficult	present	hard	hard	

As in the case of the clinical Consumer, from the point of view of the end-user (Consumer) of the enacted TVMs guideline, much of what is made available is absent from the Consumer’s model (and cannot be switched off by the Consumer). In the view of the Composers and Consumers who were interviewed, these are Composer’s and not Consumer’s tools – a point made very explicitly by interviewee 2: “that’s not something I’d want my users - my end users - to see”.



### 3.3 Comparing the CASSM Models

Tables 2, 3 and 4 can be compared against each other to establish the differences in models. A comparison of tables 3 and 4 supports understanding of how Tallis Engine can be used in different domains (in this case, clinical decision making and TVM use). More centrally to the theme of this paper, a comparison of tables 2 and 3 / 4 focuses attention on the C<sup>3</sup> chain, highlighting which concepts are transferred through the chain and which are not.

First, we briefly consider the differences between Tables 3 and 4. Essentially, the only difference between these tables is in the domain model. So patient information (presented sketchily in Table 3) is replaced by ticket-buying information in Table 4. The only other difference between these tables is the inclusion of the representation of evidence in Table 3 – included there because it was mentioned by one of the clinical interviewees but it did not emerge in any of the TVMs sessions.

More interesting is the difference between Table 2 and Tables 3/4. In this case, the important features are as follows:

1. Both Composers and Consumers reported difficulty in getting an overview of the guideline (although the role of an overview is different for the two user groups).
2. Consumers found it difficult to backtrack while running guidelines. This point did not emerge from the Composers' perspectives.
3. Domain information is absent from Table 2, because this is a generic decision support environment (albeit motivated by the requirements of clinical decision making). This has negative consequences for Consumers, who think in domain terms (e.g. "patient models") rather than decision processes.
4. Plans, Tasks, Triggers and PROforma are included in Tables 3 and 4, although this information is difficult for most Consumers to work with. Similarly, the representation of evidence is noted in Table 3 as being difficult for Consumers.
5. The decision outcome was noted by Consumers as being important; from a Composer perspective, this emerges through the interaction, and is therefore not an explicit concept.

The CASSM analysis of Tallis has highlighted both important differences and inappropriate overlaps between the Composer and Consumer models. Probably the two most important themes are the inappropriate emphasis on inspection of guideline components in the Engine (item 4 in the list above), and the focus on process rather than patient models (item 3).

The inclusion of Composer-relevant information in the Consumer system (item 4) suggests a conflation of the roles of Composers and Consumers, in that what is appropriate for the former has been assumed to also be of concern to the latter.

Conversely, the differences between the two user models is reflected in the differences of emphasis in the corresponding Cassata tables. In particular, a 'patient model' was found to be important for Consumers, and several Consumers expressed an interest in being able to backtrack through the decision process. A better understanding of Consumers' requirements might lead the developers to consider how to improve backtracking in the Engine environment, and whether to incorporate an explicit patient model within the Composer environment. Explicit inclusion of a patient model would make it more difficult to develop non-clinical guidelines, but could improve the 'fit' between the tool and the target context of use.

This illustrates how, for Tallis as for other composition tools, Creators need to be aware of both Composer and Consumer roles, while keeping them apart. In this particular case, in order to encourage clinicians who use guidelines to also create them, there may be a need for specific add-ons or enabling features which ‘upgrade’ from Consumer-level to Composer-level. However, this needs to be considered separately from the basic challenge of making such guidelines usable by and useful to clinicians in their every day work, without any expectation that all users will become guideline Composers.

## 4 Discussion

We have shown how CASSM can be used to illuminate multiple classes of user model which form part of the ‘chain’ from designer to end user, and that tabulating results in the form demonstrated by Cassata enables the analyst to focus on the essential differences between these models. As discussed in the Introduction, the  $C^3$  chain is not specific to decision support or knowledge representation systems.

One role for CASSM in the development cycle is in pre-empting any conflation of Composer and Consumer models. CASSM does not explicitly differentiate between appropriate and inappropriate overlaps between models; a reasonable heuristic appears to be that Creators need to be more aware of the Consumer’s perspective, but that Consumers should not generally be expected to assimilate non-essential information about the Composer environment.

Elsewhere, we have compared the findings of CASSM analyses with those of procedurally based approaches such as Cognitive Walkthrough [6]. We have not conducted such a comparative analysis in the work with Tallis because, as should be evident from Figures 1 and 2, the procedures for working with the two interfaces are completely different. The Composer interface demands complex planning by users and an interaction based on a graphical drag-and-drop paradigm, whereas the Engine interface requires users to engage in a sequence of selections that leads them carefully through the decision process. The sequence embodied within the Engine interface is defined by the ordering of elements within the corresponding Composer knowledge representation, but is not reflected in the process that the Composer has to go through to construct the knowledge representation. These differences make it impossible to conduct a meaningful procedural comparison between the Composer and Engine interfaces; this contrasts with the conceptual comparison that CASSM has supported (section 3.3).

Tallis is an interesting example of the  $C^3$  model because decisions made by a Consumer at the early stages of an interaction session determine those aspects of the interface which will be available later on. Even website development tools may not expect this much premature commitment in the end product: at least with web sites one can backtrack and go down some different path, whereas Tallis does not offer such flexibility. However, Tallis may be unusual in having a ‘back-channel’ between Consumers and Composers, in that the same clinicians who make use of guidelines are also encouraged to compose them, and to upload them to the repository for others to consume. In that sense, there may be a special benefit in the Consumer having a view of the Creator’s world, in order to understand how the system has come to be.

Of course, programming support environments also expect Composers to act as Consumers when running, testing and debugging code, but it may be the special and detailed support for the interrogation of user outcomes (Table 4) that makes Tallis so prone to this kind of conflation. It is evident from the Consumer reports (Sections 3.2.3 and 3.2.4) that so much emphasis on intervention and diagnosis, rather than user control, can hinder rather than illuminate the support for outcomes.

CASSM can help to identify where in the ‘chain’ a particular tool is best used, because both Creators and Composers need comprehension of the other user models. In particular, Creators need to know about Composers and Consumers, and Composers need to know about Consumers. To what extent it is helpful for understanding to also flow the other way – that Consumers should understand the perspectives of Creators and Composers – remains an open question. Arguably, a ready-to-hand tool should not impose on its user the requirement to understand how it was made, or why it is the way it is. However, this is not the culture within which the Tallis development is taking place. In the current development context, the communications between the Creators, Composers and Consumers are perceived as being essential to the development of a shared culture of guideline development and use. However, the very culture that supports collaboration may also alienate potential Consumers who have no interest in being Composers. Such socio-political considerations are outside the scope of CASSM; nevertheless, the use of CASSM within this development culture has highlighted important questions about how information is presented to and used by different user populations.

## Acknowledgements

We are very grateful to all the participants in this study, both experts and novices, without whom this analysis would not have been possible, and to Paul Cairns for constructive criticism of a draft of this paper. The work on CASSM was funded by EPSRC (GR/R39108).

## References

1. Blandford, A., Green, T., Connell, I.: Formalising an understanding of user–system misfits. In: Bastide, R., Palanque, P., Roth, J. (eds.) DSV-IS 2004 and EHCI 2004. LNCS, vol. 3425, pp. 253–270. Springer, Heidelberg (2005)
2. Blandford, A., Keith, S., Butterworth, R., Fields, B., Furniss, D.: Disrupting Digital Library Development with Scenario Informed Design. In: *Interacting with Computers* (in press) (to appear)
3. Blandford, A.E., Wong, B.L.W., Connell, I.W., Green, T.R.G.: Multiple viewpoints on computer supported team work: a case study on ambulance dispatch. In: Faulkner, X., Finlay, J., D tienne, F. (eds.) *People and Computers XVI*. Proceedings of HCI 2002, September 2002, pp. 139–156. Springer, London (2002)
4. CASSM, Shrinkwrapped tutorial, Cassata tool and worked examples (2004), <http://www.ucl.ac.uk/annb/CASSM/>
5. Checkland, P.B.: *Systems Theory, Systems Practice*. John Wiley, Chichester (1981)

6. Connell, I.W., Blandford, A.E., Green, T.R.G.: CASSM and Cognitive Walkthrough: usability issues with ticket vending machines. *Behaviour & Information Technology* 23(5), 307–320 (2004)
7. Connell, I.W., Green, T.R.G., Blandford, A.E.: Ontological Sketch Models: highlighting user-system misfits. In: O'Neill, E., Palanque, P., Johnson, P. (eds.) *People and Computers XVII - Designing for Society*. Proceedings of HCI 2003, Bath, September 2003, pp. 163–178. Springer, London (2003)
8. CRUK, Tallis system (2006a) Viewed 30/11/06, <http://www.acl.icnet.uk/lab/tallis/>
9. CRUK, Tallis PROforma (2006b) Viewed 30/11/06, [http://www.openclinical.org/gmm\\_proforma.html](http://www.openclinical.org/gmm_proforma.html)
10. Fox, J., Beveridge, M., Glasspool, D.: Understanding intelligent agents: analysis and synthesis. *AI Communications* 16, 139–152 (2003)
11. Fox, J., Johns, N., Rahmanzadeh, A.: Disseminating medical knowledge: the PROforma approach. *Artificial Intelligence in Medicine* 14, 157–181 (1998)
12. KPC, Open Clinical Knowledge Publishing Collaboratory (2006) Viewed 30/11/06, <http://www.openclinical.org/kpc/Introduction.page>
13. NetBeans (no date) Viewed 12/02/07, <http://www.netbeans.org/>
14. Nielsen, J.: Heuristic Evaluation. In: Nielsen, J., Mack, R. (eds.) *Usability Inspection Methods*, pp. 25–62. John Wiley, New York (1994)
15. Norman, D.A.: Cognitive engineering. In: Norman, D.A., Draper, S.W. (eds.) *User Centred System Design*, pp. 31–62. Lawrence Erlbaum Associates, Hillsdale (1986)
16. Wharton, C., Rieman, J., Lewis, C., Polson, P.: The cognitive walkthrough method: A practitioner's guide. In: Nielsen, J., Mack, R. (eds.) *Usability inspection methods*, pp. 105–140. John Wiley, New York (1994)
17. Witten, I.H., Bainbridge, D., Boddie, S.J.: Greenstone: Open-source digital library software with end-user collection building. *Online Information Review* 25(5), 288–298 (2001)

# Consistency between Task Models and Use Cases

Daniel Sinnig<sup>1</sup>, Patrice Chalin<sup>1</sup>, and Ferhat Khendek<sup>2</sup>

<sup>1</sup> Department of Software Engineering and Computer Science,  
Concordia University, Montreal, Quebec, Canada  
{d\_sinnig, chalin}@encs.concordia.ca

<sup>2</sup> Department of Electrical and Computer Engineering,  
Concordia University, Montreal, Quebec, Canada  
khendek@ece.concordia.ca

**Abstract.** Use cases are the notation of choice for functional requirements documentation, whereas task models are used as a starting point for user interface design. In this paper, we motivate the need for an integrated development methodology in order to narrow the conceptual gap between software engineering and user interface design. This methodology rests upon a common semantic framework for developing and handling use cases and task models. Based on the intrinsic characteristic of both models we define a common formal semantics and provide a formal definition of consistency between task models and use cases. The semantic mapping and the application of the proposed consistency definition are supported by an illustrative example.

**Keywords:** Use cases, task models, finite state machines, formal semantics, consistency.

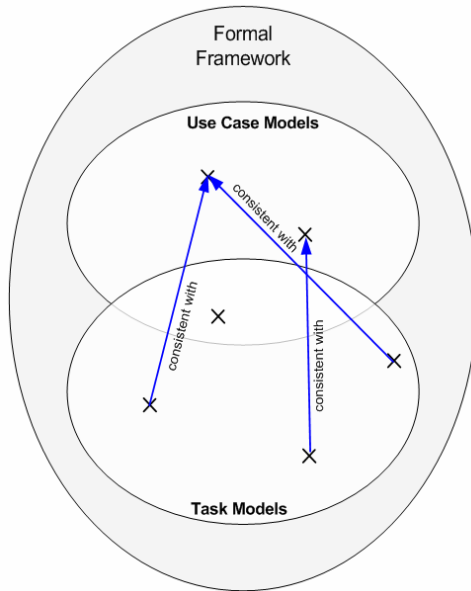
## 1 Introduction

Current methodologies and processes for functional requirements specification and UI design are poorly integrated. The respective artifacts are created independently of each other. A unique process allowing for UI design to follow as a logical progression from functional requirements specification does not exist. Moreover, it has been noted that most UI design methods are not well integrated with standard software engineering practices. In fact, UI design and the engineering of functional requirements are often carried out by different teams using different processes [1].

There is a relatively large conceptual gap between software engineering and UI development. Both disciplines have and manipulate their own models and theories, and use different lifecycles. The following issues result directly from this lack of integration:

- Developing UI-related models and software engineering models independently neglects existing overlaps, which may lead to redundancies and increase the maintenance overhead.
- Deriving the implementation from UI-related models and software engineering models towards the end of the lifecycle is problematic as both processes do not have the same reference specification and thus may result in inconsistent designs.

Use cases are the artifacts of choice for the purpose of functional requirements documentation [2] while UI design typically starts with the identification of user tasks, and context requirements [3]. Our primary research goal is to define an integrated methodology for the development of use case and task model specifications, where the latter follows as a logical progression from the former. Figure 1 illustrates the main component of this initiative, which is the definition of a formal framework for handling use cases and task models at the requirements and design levels. The cornerstone for such a formal framework is a common semantic model for both notations. This semantic model will serve as a reference for tool support and will be the basis for the definition of a consistency relation between a use case specification and a task model specification. The latter is the focus of this paper.



**Fig. 1.** Relating Use Cases and Task Models within a Formal Framework

The structure of this paper is as follows. Section 2 reviews and compares key characteristics of use cases and task models. Section 3 presents a formal mapping from use cases and task models to (nondeterministic) state machines. Based on the intrinsic characteristics of use cases and task models, we provide a formal definition of consistency. Our definition is illustrated with an example as well as with a counterexample. Finally in Section 4, we draw the conclusion and provide an outlook to future research.

## 2 Background

In this section we remind the reader of the key characteristics of use cases and task models. For each notation we provide definitions, an illustrative example as well as a

formal representation. Finally, both notations are compared and the main commonalities and differences are contrasted.

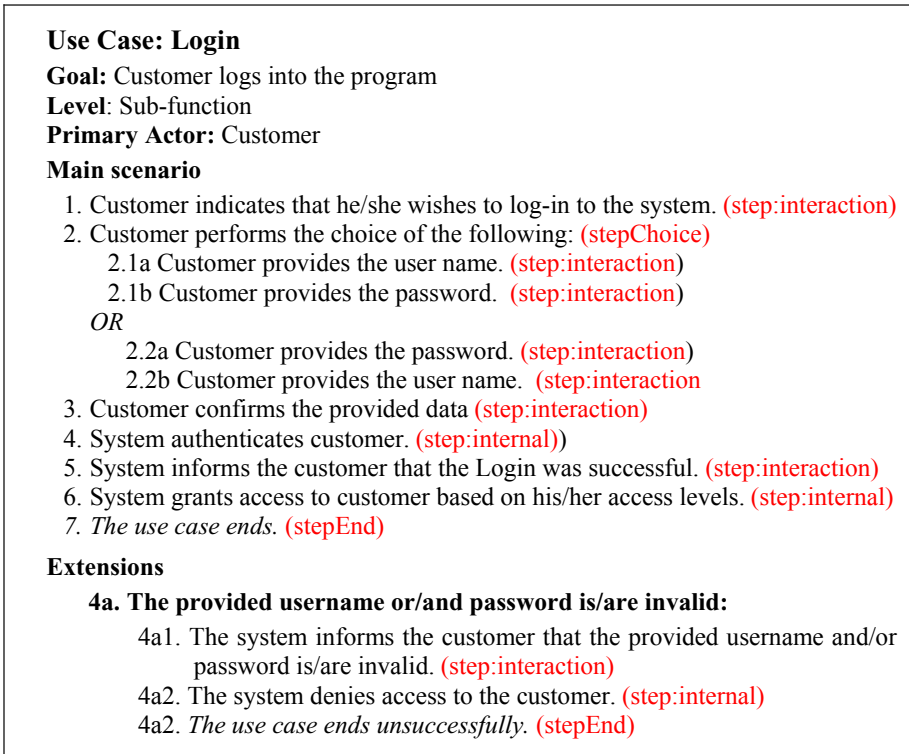
## 2.1 Use Cases

A use case captures the interaction between actors and the system under development. It is organized as a collection of related success and failure scenarios that are all bound to the same goal of the primary actor [4]. Use cases are typically employed as a specification technique for capturing functional requirements. They document the majority of software and system requirements and as such, serve as a contract (of the envisioned system behavior) between stakeholders [2]. In current practice, use cases are promoted as structured textual constructs written in prose language. While the use of narrative languages makes use case modeling an attractive tool to facilitate communication among stakeholders, prose language is well known to be prone to ambiguities and leaves little room for advanced tool support.

As a concrete example, Figure 2 presents a sub-function level use case for a “Login” function. We will be using the same example throughout this paper, and for the sake of simplicity, have kept the complexity of the use case to a minimum. A use case starts with a header section containing various properties of the use case. The core part of a use case is its main success scenario, which follows immediately after the header. It indicates the most common ways in which the primary actor can reach his/her goal by using the system. The main success scenario consists of a set of steps as well as (optional) control constructs such as choice points. We note that technically and counter-intuitively to its name, the main success scenario does not specify a single scenario but a set of scenarios. However, current practice in use case writing suggests the annotation of the main success scenario with such control constructs [2]. Within our approach we acknowledge this “custom” by allowing control structures to be included in the main success scenario.

A use case is completed by specifying the use case extensions. These extensions constitute alternative scenarios which may or may not lead to the fulfillment of the use case goal. They represent exceptional and alternative behavior (relative to the main success scenario) and are indispensable to capturing full system behavior. Each extension starts with a condition (relative to one or more steps of the main success scenario), which makes the extension relevant and causes the main scenario to “branch” to the alternative scenario. The condition is followed by a sequence of action steps, which may lead to the fulfillment or the abandonment of the use case goal and/or further extensions. From a requirements point of view, exhaustive modeling of use case extensions is an effective requirements elicitation device.

As mentioned before use cases are typically presented as narrative, informal constructs. A formal mapping from their informal presentation syntax to a semantic model is not possible. Hence, as a prerequisite, for the definition of formal semantics and consistency, we require use cases to have a formal structure, which is independent of any presentation. We have developed a XML Schema (depicted in Figure 3) which acts as a meta model for use cases. As such, it identifies the most important use case elements, defines associated mark-up and specifies existing containment relationships among elements. We use XSLT stylesheets [5] to automatically generate a “readable” use case representation (Figure 2) from the corresponding XML model.



**Fig. 2.** Textual Presentation of the “Login” Use Case

Most relevant for this paper is the definition of the *stepGroup* element as it captures the behavioral information of the use case. As depicted, the *stepGroup* element consists of a sequence of one of the following sub elements:

- The *step* element denotes a use case step capturing the primary actor’s interactions or system activities. It contains a textual description and may recursively nest another *stepGroup* element. As implied by the annotations in Figure 2, we distinguish between interaction steps and internal steps. The former are performed or are observable by the primary actor and require a user interface, whereas the latter are unobservable by the primary actor.
- The *stepEnd* element denotes an empty use case step which has neither a successor nor an extension.
- The *stepChoice* element denotes the alternative composition of two *stepGroup* elements.
- The *stepGoto* element denotes an arbitrary branching to another *step*.



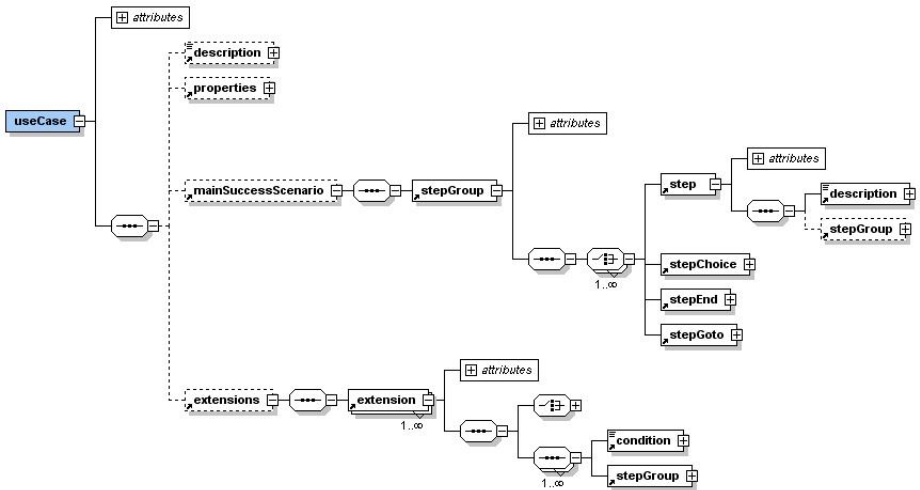


Fig. 3. Use Case Meta Model

We note that the *stepGroup* element is part of the *mainSuccessScenario* as well as the *extension* element. The latter additionally contains a condition and a reference to one or many steps stating *why* and *when* the extension may occur.

## 2.2 Task Models

User task modeling is by now a well understood technique supporting user-centered UI design [6]. In most UI development approaches, the task set is the primary input to the UI design stage. Task models describe the tasks that users perform using the application, as well as how the tasks are related to each other. Like use cases, task models describe the user's interaction with the system. The primary purpose of task models is to systematically capture the way users achieve a goal when interacting with the system [7]. Different presentations of task models exist, ranging from narrative task descriptions, work flow diagrams, to formal hierarchical task descriptions.

Figure 4 shows a ConcurTaskTreesEnvironment (CTTE) [8] visualization of the "Login" task model. CTTE is a tool for graphical modeling and analyzing of ConcurTaskTrees (CTT) models [9]. The figure illustrates the hierarchical break down and the temporal relationships between tasks involved in the "Login" functionality (depicted in the use case of Section 2.1). More precisely, the task model specifies how the user makes use of the system to achieve his/her goal but also indicates how the system supports the user tasks. An indication of task types is given by the used symbol to represent tasks. Task models distinguish between externally visible system tasks and interaction tasks. Internal system tasks (as they are captured in use cases) are omitted in task models.

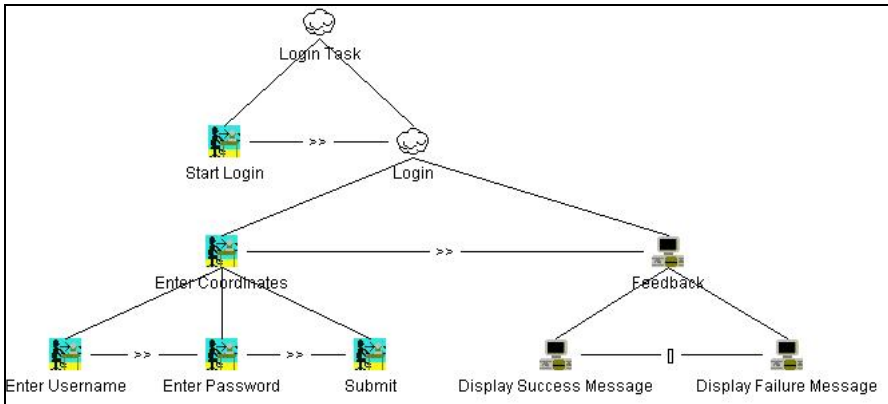


Fig. 4. “Login” Task Model

Formally a task model is organized as a directed graph. Tasks are hierarchically decomposed into sub-tasks until an atomic level has been reached. Atomic tasks are also called actions, since they are the tasks that are actually carried out by the user or the system. The execution order of tasks is determined by temporal operators that are defined between peer tasks. In CTT various temporal operators exist; examples include: enabling ( $\gg$ ), choice ( $[]$ ), iteration ( $*$ ), and disabling ( $[>$ ). A complete list of the CTT operators together with a definition of their interpretation can be found in [9].

### 2.3 Use Cases vs. Task Models

In the previous two sections, the main characteristics of use cases and task models were discussed. In this section, we compare both specifications and outline noteworthy differences and commonalities. In Section 3 the results of this comparison will be used as guides for the definition of a proper consistency relation that fits the particularities of both specifications.

Both use cases and task models belong to the family of scenario-based notations, and as such capture sets of usage scenarios of the system. In theory, both notations can be used to describe the same information. In practice however, use cases are mainly employed to document functional requirements whereas task models are used to describe UI requirements/design details. Based on this assumption we identify three main differences which are pertinent to their purpose of application:

1. Use cases capture requirements at a higher level of abstraction whereas task models are more detailed. Hence, the atomic actions of the task model are often lower level UI details that are irrelevant (actually contraindicated [2]) in the context of a use case. We note that due to its simplicity, within our example, this difference in the level of abstraction is not explicitly visible.
2. Task models concentrate on aspects that are relevant for UI design and as such, their usage scenarios are strictly depicted as input-output relations between the user and the system. Internal system interactions (i.e. involvement of secondary actors or internal computations) as specified in use cases are not captured.

3. If given the choice, a task model may only implement a subset of the scenarios specified in the use case. Task models are geared to a particular user interface and as such must obey to its limitations. E.g. a voice user interface will most likely support less functionality than a fully-fledged graphical user interface. In the next section we will address the question of which use case scenarios the task model may specify and which scenarios the task model *must* specify.

### 3 Formal Definition of Consistency

In this section we first review related work and mathematical preliminaries. Next we define the mapping from use cases and task models to the proposed semantic domain of finite state machines. Finally we provide a formal notion of consistency between use cases and task models.

#### 3.1 Related Work

Consistency verification between two specifications has been investigated for decades and definitions have been proposed for various models [10-14]. But to our knowledge a formal notion of consistency has never been defined for use cases and task model specification.

Brinksma points out that the central question to be addressed is “*what is the class of valid implementations for a given specification?*” [15] To this effect various pre-orders for labeled transition systems have been defined. Among others the most popular ones are *trace inclusion* [16], *reduction* [15], and *extension* [12, 15, 17]. The former merely requires that every trace of the implementation is also a valid trace according to the specification. The *reduction* preorder defines an implementation as a proper reduction of a specification if it results from the latter by resolving choices that were left open in the specification [15]. In this case, the implementation may have less traces. In the case of the *extension* preorder two specifications are compared for consistency by taking into account that one specification may contain behavioral information which is not present in the other specification. In the subsequent section we adopt (with a few modifications) the *extension* preorder as the consistency relation between uses cases and task models. A prerequisite for a formal comparison (in terms of consistency) of use cases and task models is a common semantics.

In [18] Sinnig et al. propose a common formal semantics for use cases and task models based on sets of partial order sets. Structural operational semantics for CTT task models are defined in [19]. In particular Paternò defines a set of inference rules to map CTT terms into labeled transition systems. In [20] Xu et al. suggest process algebraic semantics for use case models, with the overall goal of formalizing use case refactoring.

In [21, 22, 23] use case graphs have been proposed to formally represent the control flow within use cases. For example Koesters et al. define a use case graph as a single rooted directed graph, where the nodes represent use case steps and the edges represent the step ordering. Leaf nodes indicate the termination of the use case [21].

In our approach we define common semantics for use cases and task model based on finite state machines. In the next section we lay the path for the subsequent sections by providing the reader with the necessary mathematical preliminaries.

### 3.2 Mathematical Preliminaries

We start by reiterating the definition of (non-deterministic) finite state machines (FSM) which is followed by the definitions of auxiliary functions needed by our consistency definition.

**Definition 1.** A (**nondeterministic**) **finite state machine** is defined as the following tuple:  $M = (Q, \Sigma, \delta, q_0, F)$ , where

- $Q$  is a finite set of states.
- $\Sigma$  is a finite set of symbols (the input alphabet), where each symbol represents an event.
- $q_0$  is the initial state with  $q_0 \in Q$
- $F$  is the set of final (accepting) states with  $F \subseteq Q$
- $\delta: Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$  is the transition function<sup>1</sup>, which returns for a given state and a given input symbol the set of (possible) states that can be reached.

In what follows we define a set of auxiliary functions which will be used later on for the definition of consistency between two FSMs.

**Definition 2.** The **extended transition function**.  $\delta^*: Q \times \Sigma^* \rightarrow 2^Q$  is defined in a standard way as:

$$\delta^*(q_i, w) = Q_j$$

where  $Q_j$  is the set of possible states the Non-deterministic FSM may be in, having started in state  $q_i$  and after the sequence of inputs  $w$ . A formal recursive definition of the extended transition function can be found in [24].

**Definition 3.** The function **accept**:  $Q \rightarrow 2^\Sigma$  denotes the set of possible symbols which may be accepted in a given state.

$$\text{accept}(q) = \{a \mid \delta^*(q, a)\}$$

Note that ‘ $a$ ’ ambiguously denotes either a symbol or the corresponding string of one element.

**Definition 4.** The function **failure**:  $Q \rightarrow 2^\Sigma$  denotes the set of possible symbols which may not be accepted (refused) in a given state.  $\text{failure}(p)$  is defined as the complement of  $\text{accept}(p)$ .

$$\text{failure}(p) = \Sigma \setminus \text{accept}(p)$$

**Definition 5.** The **language L accepted** by a FSM  $M = (Q, \Sigma, \delta, q_0, F)$  is the set of all strings of event symbols for which the extended transition function yields at least one final state (after having started in the initial state  $q_0$ ). Each element of  $L$  represents one possible scenario of the FSM.

$$L(M) = \{w \mid \delta^*(q_0, w) \cap F \neq \emptyset\}$$

---

<sup>1</sup>  $\lambda$  Represents the empty string.  $\Sigma^0 = \{\lambda\}$ .

**Definition 6.** The **set of all traces** generated by the NFSM  $M = (Q, \Sigma, \delta, q_0, F)$  is the set of all strings or sequences of events accepted by the extended transition function in the initial state.

$$\text{Traces}(M) = \{w \mid \delta^*(q_0, w)\}$$

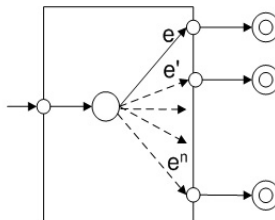
### 3.3 Mapping Use Cases to Finite State Machines

In this section we define a mapping from use cases to the domain of finite state machines. It is assumed that the use case specification complies with the structure outlined in Section 2.1.

The building blocks of a use case are the various use case steps. According to the control information entailed in the use case, the various steps are gradually composed into more complex steps until the composition eventually results in the entire use case. We distinguish between sequential composition and choice composition. The former is denoted by the relative ordering of steps within the use case specification or the *stepGoto* construct, whereas the latter is denoted by the *stepChoice* element.

A use case step may have several outcomes (depending on the number of associated extensions). This has an implication on the composition of use case steps. In particular the sequential composition of two use case steps is to be defined *relative* to a given outcome of the preceding step. For example the steps of the main success scenario are sequentially composed relative to their successful (and most common) outcome. In contrast to this, the steps entailed in use case extensions are sequentially composed relative to an alternative outcome of the corresponding “extended” steps.

Following this paradigm, we propose representing each use case step as a **finite state machine**. Figure 5 depicts a blueprint of such a state machine representing an atomic use case step. The FSM only consists of an initial state and multiple final states. The transitions from the initial state to the final states are triggered by events. Each event represents a different outcome of the step. In what follows we illustrate how the sequential composition and choice composition of use case steps are semantically mapped into the sequential composition and deterministic choice composition of FSMs.



**Fig. 5.** FSM Blueprint for Atomic Use Case Steps

Figure 6 schematically depicts the sequential composition of two FSMs  $M_1$  and  $M_2$  relative to state  $q_n$ . The resulting FSM is composed by adding a transition from  $q_n$  (which is a final state in  $M_1$ ) and the initial state ( $s_0$ ) of  $M_2$ . As a result of the composition, both  $q_n$  and  $s_0$  lose their status as final or initial states, respectively. The choice

composition of use case steps is semantically mapped into the deterministic choice composition of the corresponding FSMs. As depicted on the left hand side of Table 1 (in Section 3.4) the main idea is to merge the initial states of the involved FSMs into one common initial state of the resulting FSM.

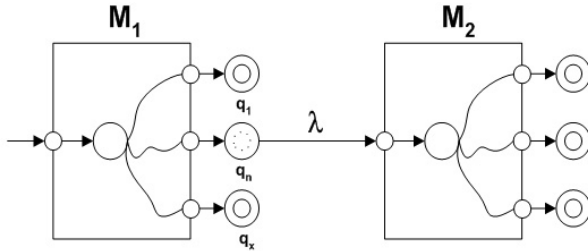


Fig. 6. Sequential Composition of Two FSMs

Figure 7 depicts the FSM representing the “Login” use case from Section 2.1. It can be easily seen how the FSM has been constructed from various FSMs representing the use case steps. Identical to the textual use case specification, the FSM specifies the entry of the login coordinates (denoted by the events  $e_{21}$  and  $e_{22}$ ) in any order. Due to the associated extension, step 4 is specified as having different outcomes. One outcome (denoted by event  $e_4$ ) will lead to a successful end of the use case whereas the other outcome (denoted by event  $e_{4a}$ ) will lead to login failure.

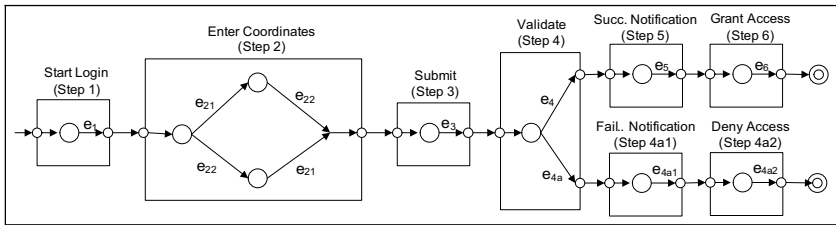


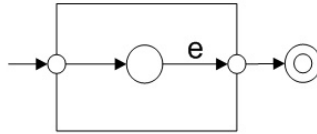
Fig. 7. FSM Representation of the “Login” Use Case

### 3.4 Mapping CTT Task Models to Finite State Machines

After we have demonstrated how use cases are mapped to FSM specifications, we now demonstrate the mapping from CTT task models to the same semantic domain. The building blocks of task models are the action tasks (i.e. tasks that are not further decomposed into subtasks). In CTT, action tasks are composed to complex tasks using a variety of temporal operators. In what follows we will demonstrate how actions tasks are mapped into FSMs and how CTT temporal operators are mapped into compositions of FSMs.

In contrast to use case steps, tasks do not have an alternative outcome and the execution of a task has only one result. Figure 8 depicts the FSM denoting an action task. It consists of only one initial and one final state. The transition between the two states is triggered by an event denoting the completion of task execution.

In what follows we demonstrate how CTT temporal operators (using the example of *enabling* ( $\gg$ ) and *choice* ( $[]$ )) are semantically mapped into compositions of FSMs. The sequential execution of two tasks (denoted by the *enabling* operator) is semantically mapped into the sequential composition of the corresponding state machines. As each FSM representing a task has only one final state, the sequential composition of two FSMs  $M_1$  and  $M_2$  is performed by simply defining a new lambda transition from the final state of  $M_1$  to the initial state of  $M_2$ .



**Fig. 8.** FSM Representing an Action Task

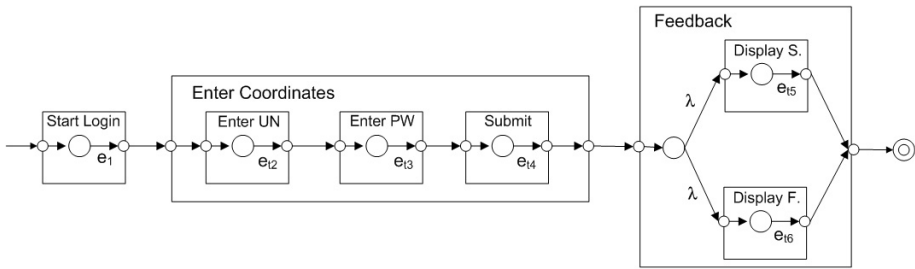
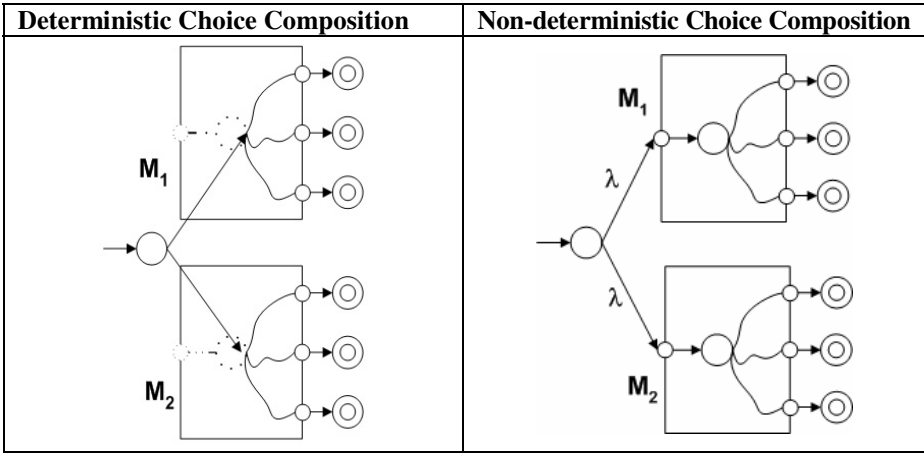
The mapping of the CTT *choice* operator is less trivial. At this point it is important to recall our assumption (see Section 2.3) that task models specify system behavior as an input-output relation, where internal system events are omitted. Moreover the execution of a task can result only in one state. The specification of alternative outcomes is not possible. Both observations have implications on the semantic mapping of the *choice* operator. Depending on the task types of the operands we propose distinguishing between deterministic choices and non-deterministic choices. If the enabled tasks of both operands are application tasks (e.g. “Display Success Message”, “Display Failure Message”, etc.) then (a) the non-deterministic choice is used to compose the corresponding FSMs, otherwise (b) the deterministic choice composition is employed.

The former (a) is justified by the fact that each application works in a deterministic manner. Hence, the reason why the system performs either one task or the other is because the internal states of the system are not the same. Depending on its internal state, the system either performs the task specified by the first operand or the task specified by the second operand. However, task models do not capture internal system operations. As a result, from the task model specification, we do not know why the system is in one state or the other and the choice between the states becomes non-deterministic.

As for the latter case (b), the choice (e.g. between two interaction tasks) is interpreted as follows. *In a given state of the system, the user has the exclusive choice between carrying one or the other task.* Clearly the system may only be in one possible state when the choice is made. Hence, the deterministic choice composition is applicable.

Table 1 schematically depicts the difference between deterministic choice composition and non-deterministic choice composition of two FSMs. In contrast to deterministic choice composition (discussed in the previous section) non-deterministic choice composition does not merge the initial states of the involved FSMs, but introduces a new initial state.

**Table 1.** Choice Compositions of FSMs



**Fig. 9.** FSM Representation of the “Login” Task Model

Figure 9 portrays the corresponding FSM for the “Login” task model. We note that the non-deterministic choice composition has been employed to denote the CTT choice between the system tasks “Display Success Message” and “Display Failure Message”. After the execution of the “Submit” task the system non-deterministically results in two different states. Depending on the state either the Failure or the Success Message is displayed.

For the sake of completeness we now briefly sketch out how the remaining CTT operators (besides *enabling* and *choice*) can be mapped into FSM compositions: In CTT it is possible to declare tasks as *iterative* or *optional*. Iterative behavior can be implemented by adding a transition from the final state to the initial state of the FSM representing the task, whereas optional behavior may be implemented by adding a lambda transition from the initial state to the final state. The remaining CTT operators are more or less a short hand notation for more complex operations. As such they can be rewritten using the standard operators. For example the *order independency* ( $t_1 \parallel t_2$ ) operator can be rewritten as the choice of either executing  $t_1$  followed by  $t_2$  or executing  $t_2$  followed by  $t_1$ . Another example is the *concurrency* ( $t_1 \parallel\parallel t_2$ ) operator,



which can be rewritten as the choice between all possible interleavings of action tasks entailed in  $t_1$  and  $t_2$ . Similar rewritings can be established for the operators *disabling* and *suspend/resume*. Further details can be found in [18].

### 3.5 A Formal Definition of Consistency

In Section 2.3 we made the assumption and viewed task models as UI specific implementations of a use case specification. In this section we will tackle the question of what is the class of valid task model implementations for a given use case specification. To this effect we propose the following two consistency principles:

1. Every scenario in the task model is also a valid scenario in the use case specification. That is, *what the implementation (task model) does is allowed by the specification (use case)*.
2. Task models do not capture internal operations, which are however specified in the corresponding use case specification. In order to compensate for this allowed degree of under-specification we require the task model to *cater for all possibilities that happen non-deterministically* from the user's perspective.

For example as specified by the "Login" use case the system notifies the primary actor of the success or failure of his login request based on the outcome of the *internal* validation step. According to the second consistency principle we require every task model that implements the "Login" use case specification to specify the choice between a task representing the success notification and a task representing the failure notification.

We note that the first consistency principle can be seen as a safety requirement, as it enforces that *nothing bad can happen* (the task model must not specify an invalid scenario with respect to the use case specification). The second consistency principle can be seen as a liveness requirement as it ensures that the task model specification does not "deadlock" due to an unforeseen system response.

In order to formalize the two consistency principles we adopt Brinksma's extension relation [15], which tackles a related conformance problem for labeled transition systems. Informally, a use case specification and a task model specification are consistent, if and only if the later is an extension of the former. Our definition of consistency between task models and use cases is as follows:

**Definition 7 Consistency.** Let  $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$  be the FSM representing the use case  $U$  and  $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$  be the FSM representing the task model  $T$ . Then  $T$  is consistent to the use case  $U$  iff the following two properties hold.

- (1) **Language inclusion** (safety property)

$$L(M_2) \subseteq L(M_1)$$

- (2) **Sufficient coverage:** (liveness property)

$$\forall t \in T \text{ with } T = \{\text{Traces}(M_2) \setminus L(M_2)\}$$

- a. Let  $Q_{M_1} = \{p_1, p_2, \dots, p_n\}$  be  $\delta^*(q_{01}, t)$ . That is, the  $p_i$ 's are all and only the states that can be reached from the initial state of  $M_1$  after having accepted  $t$ .

- b. Let  $Q_{M_2} = \{q_1, q_2, \dots, q_m\}$  be  $\delta^*(q_{02}, t)$ . That is, the  $q_i$ 's are all and only the states that can be reached from the initial state of  $M_2$  after having accepted  $t$ .
- c. **We require that:**  $\forall p \in Q_{M_1} \exists q \in Q_{M_2}. \text{failure}(p) \subseteq \text{failure}(q)$ .

The liveness property states that the task model FSM must refuse to accept an event in a situation where the use case FSM may also refuse. If we translate this condition back to the domain of use cases and task models, we demand the task model to provide a task for every situation where the use case must execute a corresponding step. The main difference to Brinksma's original definition is that our definition is defined over finite state machines instead of labeled transition systems. As a consequence, we require that the language accepted by the task model FSM is included in the language accepted by the use case FSM (safety property). Task models that only implement partial scenarios of the use case specification are deemed inconsistent.

One precondition for the application of the definition is that both state machines operate over the same alphabet. The mappings described in the previous sections do not guarantee this property. Hence, in order to make the FSMs comparable, a set of preliminary steps have to be performed and are described in the following:

1. **Abstraction from internal events:** Task models do not implement internal system events. Hence, we require the alphabet of the use case FSM to be free of symbols denoting internal events. This can be achieved by substituting every symbol denoting an internal event by lambda ( $\lambda$ )<sup>2</sup>.
2. **Adaptation of abstraction level:** Task model specifications are (typically) at a lower level of abstraction than their use case counterparts. As such a use case step may be refined by several tasks in the task model. Events representing the execution of these refining tasks will hence not be present in the use case FSM. We therefore require that for every event 'e' of the task model FSM there exists a bijection that relates 'e' to one corresponding event in the use case FSM. This can be achieved by replacing intermediate lower level events in the task model FSM with lambda events. Events denoting the completion of a refining task group are kept.
3. **Symbol mapping:** Finally, the alphabets of the two FSMs are unified by renaming the events of the task model FSM to their corresponding counterparts in the use case FSM.

In what follows we will apply our consistency definition to verify that the "Login" task model is a valid implementation of the "Login" use case. Table 2 depicts the FSMs for the "Login" use case ( $M_U$ ) and the "Login" task model ( $M_T$ ), after the unification of their input alphabets. We start with the verification of the safety property (language inclusion). With

$$L(M_U) = \{ \langle e_1, e_{21}, e_{22}, e_3, e_5 \rangle, \langle e_1, e_{22}, e_{21}, e_3, e_5 \rangle, \langle e_1, e_{21}, e_{22}, e_3, e_{4a1} \rangle, \langle e_1, e_{22}, e_{21}, e_3, e_{4a1} \rangle \}$$

$$L(M_T) = \{ \langle e_1, e_{21}, e_{22}, e_3, e_5 \rangle, \langle e_1, e_{21}, e_{22}, e_3, e_{4a1} \rangle \}$$

we can easily see the  $L(M_T) \subseteq L(M_U)$ . Hence the first property is fulfilled.

<sup>2</sup> Lambda denotes the empty string and as such is not part of the language accepted by an FSM.

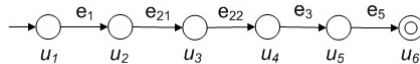
**Table 2.** Use Case FSM and Task Model FSM After the Unification of Their Alphabets

Unified Use Case FSM ( $M_U$ )	Unified Task Model FSM ( $M_T$ )

We continue with the verification of the second property (liveness). The set T of all partial runs of  $M_T$  is as follows:

$$T = \{ \langle e_1 \rangle, \langle e_1, e_{21} \rangle, \langle e_1, e_{21}, e_{22} \rangle, \langle e_1, e_{21}, e_{22}, e_3 \rangle \}$$

We verify for each trace  $t$  in T that the liveness property holds. Starting with  $t = \langle e_1 \rangle$  we obtain  $Q_{MU} = \{q_2\}$ ;  $Q_{MT} = \{u_2\}$  as the set of reachable states in  $M_U$  and  $M_T$  after having accepted  $t$ . Next we verify that for every state in  $Q_{MU}$  there exists a state in  $Q_{MT}$  with an encompassing failure set. Since  $Q_{MU}$  and  $Q_{MT}$  only contain one element we require that  $\text{failure}(q_2) \subseteq \text{failure}(u_2)$ . With  $\text{failure}(q_2) = \{e_1, e_3, e_5, a_{4a1}\}$  and  $\text{failure}(u_2) = \{e_1, e_{22}, e_3, e_5, a_{4a1}\}$  this property is clearly fulfilled. In a similar fashion we prove that the liveness property holds for the traces:  $\langle e_1, e_{21} \rangle, \langle e_1, e_{21}, e_{22} \rangle$ . More interesting is the case where  $t = \langle e_1, e_{21}, e_{22}, e_3 \rangle$ . We obtain  $Q_{MU} = \{q_6, q_7, q_{10}\}$ ;  $Q_{MT} = \{u_5, u_6, u_8\}$  as the set of reachable states in  $M_U$  and  $M_T$  after having accepted  $t$ . Next we have to find for each state in  $Q_{MU}$  a state in  $Q_{MT}$  with an “encompassing” failure set. For  $q_6$  ( $\text{failure}(q_6) = \{e_1, e_{21}, e_{22}, e_3\}$ ) we identify  $u_5$  ( $\text{failure}(u_5) = \{e_1, e_{21}, e_{22}, e_3\}$ ). For  $q_7$  ( $\text{failure}(q_7) = \{e_1, e_{21}, e_{22}, e_3, e_{4a1}\}$ ) we identify  $u_6$  ( $\text{failure}(u_6) = \{e_1, e_{21}, e_{22}, e_3, e_{4a1}\}$ ) and for  $q_{10}$  ( $\text{failure}(q_{10}) = \{e_1, e_{21}, e_{22}, e_3, e_5\}$ ) we identify  $u_8$  ( $\text{failure}(u_8) = \{e_1, e_{21}, e_{22}, e_3, e_5\}$ ). For each identified pair of  $p_i$  and  $q_i$  it can be easily seen that  $\text{failure}(p_i) \subseteq \text{failure}(q_i)$ , hence we conclude that the “Login” task model represented by  $M_T$  is consistent to the “Login” use case represented by  $M_U$  q.e.d.



**Fig. 10.** FSM Representation of an Inconsistent “Login” Task Model

We conclude this chapter with a counter example, by presenting a “Login” task model which is not a valid implementation of the “Login” use case. The FSM ( $M_{T2}$ ) portrayed by Figure 10 represents a task model which does not contain the choice between “Display Failure Message” and “Display Success Message”. Instead, after the “Submit” task ( $e_3$ ), “Success Message” ( $e_5$ ) is always displayed. It can be easily seen that the safety property holds with  $L(M_{T2}) \subseteq L(M_U)$ . The verification of the liveness property however will lead to a contradiction. For this purpose, let us consider the following trace of  $M_{T2}$ :  $t = \langle e_1, e_{21}, e_{22}, e_3 \rangle$ . We obtain  $Q_{MU} = \{q_6, q_7, q_{10}\}$  and  $Q_{MT2} = \{u_5\}$  as the set of all reachable states in  $M_U$  and  $M_T$  after having accepted  $t$ .

In this case however, for  $q_{10}$  we cannot find a corresponding state in  $Q_{MT2}$  (which in this case consists of a single element only) such that the failure set inclusion holds. We obtain  $failure(q_{10})=\{e_1, e_{21}, e_{22}, e_3, e_5\}$  and  $failure(u_5)=\{e_1, e_{21}, e_{22}, e_3, e_{4a1}\}$ . Clearly  $failure(q_{10})$  is not a subset of  $failure(u_5)$ . Hence the task model is not consistent to the “Login” use case.

## 4 Conclusion

In this paper we proposed a formal definition of consistency between use cases and task models based on a common formal semantics. The main motivation for our research is the need for an integrated development methodology where task models are developed as logical progressions from use case specifications. This methodology rests upon a common semantic framework where we can formally validate whether a task model is consistent with a given use case specification. With respect to the definition of the semantic framework, we reviewed and contrasted key characteristics of use cases and task models. As a result we established that task model specifications are at a lower level of abstraction than their use case counterparts. We also noted that task models omit the specification of internal system behavior, which is present in use cases.

These observations have been used as guides for both the mapping to finite state machines and for the formal definition of consistency. The mapping is defined in a compositional manner over the structure of use cases and task models. As for the definition of consistency, we used an adaptation of Brinksmas’s extension pre-order. We found the extension relation appropriate because it acknowledges the fact that under certain conditions two specifications remain consistent, even if one entails additional behavioral information which is omitted in the second. Both the mapping and the application of the proposed definition of consistency have been supported by an illustrative example.

As future work, we will be tackling the question of how relationships defined among use cases (i.e. *extends* and *includes*) can be semantically mapped into finite state machines. This will allow us to apply the definition of consistency in a broader context, which is not restricted to a single use case. Another issue deals with the definition of consistency among two use case specifications and in this vein also among two task model specifications. For example, if a user-goal level use case is further refined by a set of sub-function use cases it is important to verify that the sub-function use cases do not contradict the specification of the user goal use case. Finally we note that for the simple “Login” example consistency can be verified manually. However, as the specifications become more complex, efficient consistency verification requires supporting tools. We are currently investigating how our approach can be translated into the specification languages of existing model checkers and theorem provers.

## Acknowledgements

This work was funded in part by the National Sciences and Engineering Research Council of Canada. We are grateful to Homa Javahery who meticulously reviewed and revised our work.

## References

1. Seffah, A., Desmarais, M.C., Metzger, M.: Software and Usability Engineering: Prevalent Myths, Obstacles and Integration Avenues. In: *Human-Centered Software Engineering - Integrating Usability in the Software Development Lifecycle*. Springer, Heidelberg
2. Cockburn, A.: *Writing effective use cases*. Addison-Wesley, Boston (2001)
3. Pressman, R.S.: *Software engineering: a practitioner's approach*. McGraw-Hill, Boston (2005)
4. Larman, C.: *Applying UML and patterns: an introduction to object-oriented analysis and design and the unified process*. Prentice Hall PTR, Upper Saddle River (2002)
5. XSLT, XSL Transformations Version 2.0 [Internet] (Accessed: December 2006) (Last Update: November 2006), <http://www.w3.org/TR/xslt20/>
6. Paternò, F.: Towards a UML for Interactive Systems. In: Nigay, L., Little, M.R. (eds.) *EHCI 2001*. LNCS, vol. 2254, pp. 7–18. Springer, Heidelberg (2001)
7. Souchon, N., Limbourg, Q., Vanderdonckt, J.: Task Modelling in Multiple contexts of Use. In: *Proceedings of Design, Specification and Verification of Interactive Systems*, Rostock, Germany, pp. 59–73 (2002)
8. Mori, G., Paternò, F., Santoro, C.: CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Transactions on Software Engineering*, 797–813 (August 2002)
9. Paternò, F.: *Model-Based Design and Evaluation of Interactive Applications*. Springer, Heidelberg (2000)
10. Bowman, H., Steen, M.W.A., Boiten, E.A., Derrick, J.: A Formal Framework for Viewpoint Consistency. *Formal Methods in System Design*, 111–166 (September 2002)
11. Ichikawa, H., Yamanaka, K., Kato, J.: Incremental specification in LOTOS. In: *Proc. of Protocol Specification, Testing and Verification X*, Ottawa, Canada, pp. 183–196 (1990)
12. De Nicola, R.: Extensional Equivalences for Transition Systems. *Acta Informatica* 24, 211–237 (1987)
13. Butler, M.J.: *A CSP Approach to Action Systems*, PhD Thesis in Computing Laboratory. Oxford University, Oxford (1992)
14. Khendek, F., Bourduas, S., Vincent, D.: Stepwise Design with Message Sequence Charts. In: *Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE)*, Cheju Island, Korea, August 28-31 (2001)
15. Brinksma, E., Scollo, G., Steenbergen, C.: LOTOS specifications, their implementations, and their tests. In: *Proceedings of IFIP Workshop Protocol Specification, Testing, and Verification VI*, pp. 349–360 (1987)
16. Bergstra, J.A.: *Handbook of Process Algebra*. Elsevier Science Inc., Amsterdam (2001)
17. Brookes, S.D., Hoare, C.A.R., Roscoe, A.D.: A Theory of Communicating Sequential Processes. *Journal of ACM* 31(3), 560–599 (1984)
18. Sinnig, D., Chalin, P., Khendek, F.: Towards a Common Semantic Foundation for Use Cases and Task Models. *Electronic Notes in Theoretical Computer Science (ENTCS)* (December 2006) (to appear)
19. Paternò, F., Santoro, C.: The ConcurTaskTrees Notation for Task Modelling, Technical Report at CNUCE-C.N.R. (May 2001)
20. Xu, J., Yu, W., Rui, K., Butler, G.: Use Case Refactoring: A Tool and a Case Study. In: *Proceedings of APSEC 2004*, Busan, Korea, pp. 484–491 (2004)
21. Kusters, G., Pagel, B., Winter, M.: Coupling Use Cases and Class Models. In: *Proceedings of the BCS-FACS/EROS workshop on Making Object Oriented Methods More Rigorous*, Imperial College, London, June 24th, 1997, pp. 27–30 (1997)

22. Mizouni, R., Salah, A., Dssouli, R., Parreaux, B.: Integrating Scenarios with Explicit Loops. In: Proceedings of NOTERE 2004, Essaidia Morocco (2004)
23. Nebut, C., Fleurey, F., Le Traon, Y., Jezequel, J.-M.: Automatic test generation: a use case driven approach. IEEE Transactions on Software Engineering 32(3), 140–155 (2006)
24. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 3rd edn. Addison Wesley, Reading (2006)

## Questions

### **Gerrit van de Veer:**

**Question:** *I agree that UI requirements should be developed separately from functional requirements. Indeed use case models are used to “document functionality”. I would prefer to say “abused to document functionality”. Indeed task models are used to describe the dialog between user and system; I would say that CTT is “abused” to do this.*

*I disagree on continuing to mix concepts. We should not forget that Moran already in 1981, followed by Tauber, Norman (the gulf of execution), Nielsen, van Welie and van der Veer, all state that there are levels in the user’s task needs, through semantics and the functionality needed, to the dialog and syntax of interaction with the system, down to representation and ergonomics.*

*My plea:*

- *Task models describe what the users need; there is a step from task needs to functionality (for example an ATM should provide safety of my account, and I should be able to memorize any codes. This needs an analysis and a design model.)*
- *A use case can be applied as an implementation model, from functionality to dialog. This is engineering. (e.g. for ATM decide to either use a plastic card and 4 digit code, or iris scan to identification)*

Answer: Task models are often used for analysis, so I would not agree about the distinction in practice. Use cases are about requirements so it is necessary to keep them as generic as possible.

### **Michael Harrison:**

**Question:** *Is the expressive power of CTT and use cases to be limited to regular expressions?*

Answer: If we are going to make the analysis decidable then we have to. This limitation is adequate for the types of descriptions that are required

### **Yves Vandriessche:**

**Comment:** *I agreed with Gerrit van de Veer that I would also see CTT used as a first stage followed by device representation using use cases. You mentioned that UI changes at a later stage (adding a button for example) should not change the design specification. I just wanted to mention that you can use CTT at arbitrary granularity; you can keep to a more abstract level instead of going down to a level at which your leaf task represents individual widgets. Two CTTs could be used: a more general one used in the design and specification of the application and a more detailed CTT based on the former for UI design.*

# Task-Based Design and Runtime Support for Multimodal User Interface Distribution

Tim Clerckx, Chris Vandervelpen, and Karin Coninx

Hasselt University, Expertise Centre for Digital Media,  
and transnationale Universiteit Limburg  
Wetenschapspark 2, BE-3590 Diepenbeek, Belgium  
{tim.clerckx, chris.vandervelpen, karin.coninx}@uhasselt.be

**Abstract.** This paper describes an approach that uses task modelling for the development of distributed and multimodal user interfaces. We propose to enrich tasks with possible interaction modalities in order to allow the user to perform these tasks using an appropriate modality. The information of the augmented task model can then be used in a generic runtime architecture we have extended to support runtime decisions for distributing the user interface among several devices based on the specified interaction modalities. The approach was tested in the implementation of several case studies. One of these will be presented in this paper to clarify the approach.

**Keywords:** Task-based development, model-based user interface development, distributed user interfaces, multimodal user interfaces.

## 1 Introduction

In the last decade users are increasingly eager to use mobile devices as an appliance to perform tasks on the road. Together with the increase of wireless network capabilities, connecting these mobile *assistants* to other computing devices becomes easier. As a result we are at the dawn of the era of context aware computing. Context is a fuzzy term without a consent definition. In this work we define context as the collection of factors influencing the user's task in any way, as described by Dey [9]. Factors such as available platforms, sensor-based environmental context, the user's personal preferences, and setup of interaction devices appertain to this set. When we pick out context factors such as available platforms and interaction devices, we are discussing the area of Ubiquitous Computing [19] where users are in contact with several devices in their vicinity.

In previous work we have been concentrating on model-based development of context-aware interactive systems on mobile devices. We created a task-based design process [5] and a runtime architecture [6] enabling the design, prototyping, testing, and deployment of context-aware user interfaces. The focus in our previous work was to create context-aware applications where context factors such as sensor-based context information or information from a user model can be associated with a task model in order to enable the generation of prototypes and to use a generic runtime architecture. However, in our approach the user interface was always centralized on a

mobile device. In this work we describe how we have extended our framework, DynaMo-AID, in order to support the shift towards Ubiquitous Computing. We will discuss how a task model can be enriched with properties that are used (1) at design time to specify how tasks should be presented to the user according to the platform and (2) at runtime to distribute the tasks among the available interaction resources (definition 1). Devices may support several distinct interaction techniques. E.g. on the one hand editing text on a PDA can be accomplished by using a stylus to manipulate a software keyboard. On the other hand speech interaction can be used provided that the PDA is equipped with a microphone. As a result, at runtime has to be decided which interaction resources are at the user's disposal and a usable distribution among interaction resources has to be chosen.

Runtime distribution requires meta data about the tasks in order to realize a usable distributed user interface. This is in particular the case when we are considering ubiquitous environments because at design time it is impossible to know what the environment will look like regarding available interaction resources. E.g. the user walks around with his/her mobile device and comes across a public display that can be communicated with through a wireless connection. When this is the case decisions regarding user interface distribution have to be taken at runtime to anticipate on the current environment. Furthermore, a mapping of abstract information about the user interface to more concrete information is required to construct the final user interface due to the unknown nature of the target device(s).

The remainder of this paper is structured as follows. First we give a brief overview of the DynaMo-AID development process (section 2.1). We focus on the parts relevant for this paper. Next we elaborate on the changes we have applied to the process to enable the support for modelling multimodal and distributed user interfaces (section 2.2). Afterwards the ontology constructed to support the modelling of the modalities and devices is discussed (section 2.3). Section 3 discusses the runtime architecture: first an overview is presented (section 3.1), then we focus on the rendering engine (section 3.2), finally we discuss the approach used to decide how to distribute the tasks among the available devices. In the following section we will discuss related work and compare it to our approach. Finally conclusions are drawn and future work is discussed.

## 2 Overview of the Extended DynaMo-AID Development Process

In this section we first introduce the DynaMo-AID development process for context-aware user interfaces. We emphasize the changes we have made to support the development of multimodal and distributed user interfaces. We focus on the part of the design process where a task specification is enriched with interaction constraints. Finally we elaborate on the environment ontology used for defining the interaction constraints.

### 2.1 Developing Context-Aware User Interfaces

The DynaMo-AID development process (Fig. 1) is prototype-driven with the aim to obtain a context-aware user interface. The process consists of the design of several



abstract and concrete models. After the specification of these models, the supporting tool generates a prototype taking into account the models. The prototype can then be evaluated to seek for flaws in the models. Afterwards the models can be updated accordingly and a new prototype is generated. These steps in the process can be performed iteratively until the designer is satisfied with the resulting user interface. Next the user interface can be deployed on the target platform.

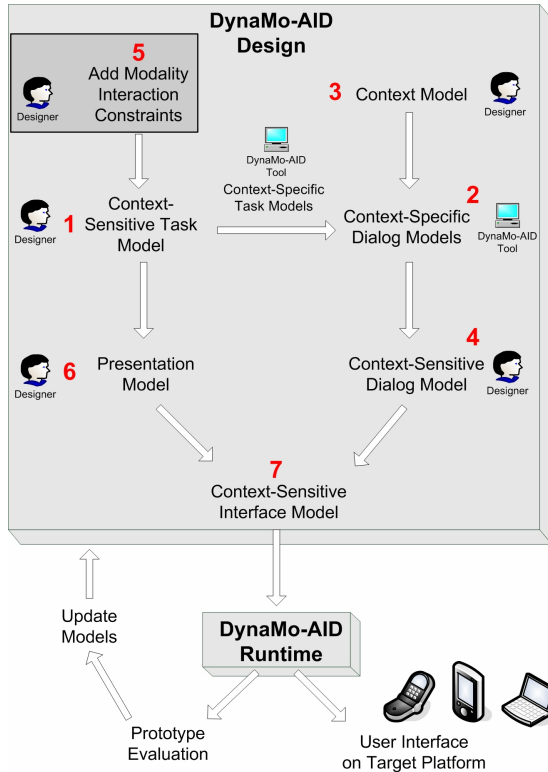


Fig. 1. Overview of the DynaMo-AID development process

The upper part of Fig.1 reveals an overview of the design process. First the designer has to construct a context-sensitive task model (1). To accomplish this, the designer makes use of the ConcurTaskTree notation [12] augmented with extra tasks to introduce context-awareness at the task level [4]. Taking into account this information, the tool extracts a set of dialog models (2) where each dialog model is relevant for a particular context of use. Afterwards these dialog models are connected at those points relevant to apply a context change (4), i.e. a switch from a dialog model relevant in a certain context of use to another dialog model relevant in another context of use. Furthermore the designer specifies the kind of context information implying the context change (3). The fifth step (5) is an extension and will be discussed in section 2.2. Next the concrete tasks are annotated with Abstract Interaction Objects (AIOs) [17] providing an abstract description about the way the

task will have to be presented to the user (6). The aggregate of the models are collected in the interface model (7) which is the input for the runtime architecture in order to either generate a prototype or deploy the user interface on the target platform.

Important for the remainder of the paper is the fact that the dialog model is a State Transition Network (STN). Each state in the STN is an enabled task set, a collection of tasks enabled during the same period of time [12]. This means the tasks should be presented to the user simultaneously, i.e. in the same dialog. The transitions of the STN are labelled with the task(s) initiating the transition to another state. Using this information, a dialog controller can keep track of the current state of the user interface and invoke a switch to another dialog if appropriate (section 3).

Accordingly the dialog model provides the information necessary to decide *which* tasks have to be deployed at a certain moment in time. When several devices and/or interaction modalities are available to the user, the question arises *where* these tasks have to be deployed.

Previous research already tackled the problem of deploying task sets on different devices. Paternò and Santoro [13] for instance described that tasks or domain objects related to a task can be assigned to a selection of platforms in order to decide at runtime whether or not to deploy a task according to the current platform. Our approach also supports this possibility at the task level where it is possible to assign different tasks to different contexts of use (platform is one kind of *context of use*).

However, we argue the approach of enabling tasks for a certain platform and disabling these same tasks for another platform might constrain the user in accomplishing his/her goals. On the one side this can be desirable when the domain objects supporting the performance of this task are constrained by the platform but on the other side the user will not be able to perform all the tasks in the path to accomplish his/her goals. This problem can be tackled by distributing the tasks among different devices in the user's vicinity in a way that all the necessary tasks can be presented to the user. In the next section we propose a method to assign interaction constraints to the tasks in order to make the distribution of tasks among distinct devices and/or interaction modalities possible at runtime.

## 2.2 Supporting the Design of Distributed and Multimodal User Interfaces

As we have explained in the previous section each state in the dialog model consists of a set of tasks. When the user interface is deployed in a highly dynamic environment with different interaction devices and/or modalities the system has to decide which tasks are deployed on which interaction device supporting the appropriate modalities. Some additional abstract information regarding task deployment is necessary to make these decisions. Therefore we define the following terms based on the definitions in [18]:

**Definition 1.** An *Interaction Resource (IR)* is an atomic input or output channel available to a user to interact with the system.

**Definition 2.** An *Interaction Resource Class (IRC)* is a class of Interaction Resources sharing the same interaction modalities.

**Definition 3.** An *Interaction Device (ID)* is a computing device that aggregates Interaction Resources associated with that particular computing device.

**Definition 4.** An Interaction Group (IG) is a set of joined Interaction Devices necessary for a user to perform his/her tasks.

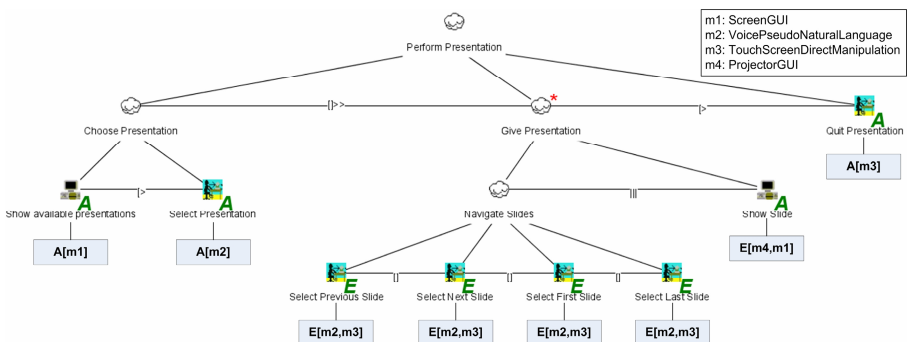
An example of an ID is a traditional desktop computer that aggregates the IRS keyboard, mouse and display screen. The QWERTY-keyboard attached to the desktop computer is an Interaction Resource belonging to the Interaction Resource Class of keyboards. An example of an IG is the collection of TV ID with a set-top box and a PDA ID, where the PDA is used as a remote control to interact with the TV system.

The goal of the research described in this paper is to find a way to distributed tasks among the available Interaction Resources, given the setup of an Interaction Group. To accomplish this the designer will have to provide additional information for each task about the types of Interaction Resources that can be used to perform the task. Because a task might be performed by several distinct Interaction Resource Classes (e.g. editing text can be done with a keyboard and/or speech) the designer will have to specify how these IRCs relate to each other. This can be expressed using the CARE properties introduced by Coutaz et al. [8]. The CARE properties express how a set of modalities relate to each other:

- **Complementarity:** all the modalities have to be used to perform the task;
- **Assignment:** a single modality is assigned to the task in order to perform the task;
- **Redundancy:** all the modalities have the same expressive power meaning the use of a second modality to perform the task will not contribute anything to the interaction;
- **Equivalence:** the task can be performed by using any one of the modalities.

The CARE properties are an instrument to reason about multimodal interactive systems. We use the CARE properties in our approach to indicate how the different modalities assigned to the same task relate to each other. Therefore we define:

**Definition 5.** A Modality Interaction Constraint (MIC) is a collection of modalities related to each other through a CARE property.



**Fig. 2.** Example task model with interaction constraints appended to tasks

The information provided by the Modality Interaction Constraint associated with a task can then be used at runtime to find an Interaction Resource belonging to an Interaction Resource Class supporting the appropriate modalities. The relation between modalities and IRCs will be explained in section 2.3.

Fig. 2 shows an example of a task model annotated with Modality Interaction Constraints. The task model describes the task of performing a presentation. First the presentation has to be selected. To accomplish this the available presentations are shown to the user on a device supporting the *ScreenGUI* output modality. The task to select the desired presentation is assigned to the *VoicePseudoNaturalLanguage* modality. This implies the task can only be performed using speech input. Afterwards the presentation can commence. The presenter can navigate through the slides by using a device supporting *VoicePseudoNaturalLanguage*, *TouchScreenDirect-Manipulation* or both in which case the user chooses the modality. Meanwhile the slide is shown on a device using either a *ProjectorGUI* or a *ScreenGUI*.

The presentation can only be switched off using *TouchScreenDirectManipulation* to prevent someone in the audience to end the presentation prematurely.

### 2.3 Interaction Environment Ontology

In order to make it easy for a designer to link modalities to tasks, we have constructed an extensible interaction environment ontology describing different modalities, Interaction Resource, and the way these two concepts are related to each other. The ontology we have constructed is an extension of a general context ontology used in the DynaMo-AID development process [14].

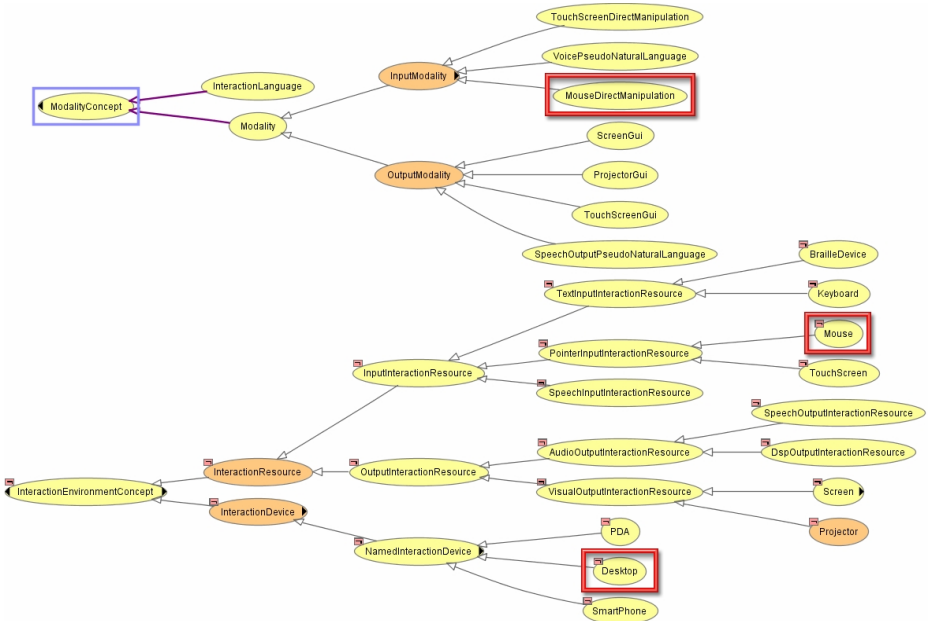


Fig. 3. Structure of the Interaction Environment Ontology

Fig. 3 shows the ontology consisting of two parts. The first part describes the interaction environment using the classes *InteractionDevice* and *InteractionResource*. An Interaction Device aggregates one or more interaction resources using the *hasInteractionResource* property. An interaction resource can either be an *OutputInteractionResource* or an *InputInteractionResource*. Every class in the ontology has particular properties describing the characteristics of individuals of that class. For example, a Desktop individual contains *hasInteractionResource* properties pointing to individuals of the class *CrtScreen*, *Keyboard* and *Mouse*. The Mouse individual on its turn has properties for describing the number of buttons, the used technology...

The second part of the ontology describes the possible modalities based on concepts described in [8]. In this work a modality is defined as the conjunction of an interaction language (direct manipulation, pseudo natural language, gui...) and an interaction device/resource (mouse, keyboard, speech synthesizer...). To model this, we added the classes *InteractionLanguage* and *Modality* to our ontology. A *Modality* individual can be an *InputModality* or an *OutputModality*. A concrete *Modality* individual is defined by two properties. The *usesInteractionLanguage* property points to an *InteractionLanguage* individual. At this time these are *DirectManipulationLanguage*, *GuiLanguage* or *PseudoNaturalLanguage*. It is possible for the designer to add new *InteractionLanguage* individuals to the ontology. The second property of *Modality* individuals is the *usesDevice* property. This property points to an *InteractionResource* individual. In this way we created six predefined modalities: *MouseDirectManipulation*, *KeyboardDirectManipulation*, *VoicePseudoNaturalLanguage*, *SpeechOutputPseudoNaturalLanguage*, *ScreenGui* and *ProjectorGui*. A designer can add new modalities to the ontology as she/he likes. To link a particular *Modality* individual to an *InteractionDevice* individual the property *supportsModality* is used. As shown in fig. 3 using the thick rectangles, an individual *desktopPC1* of the *Desktop* class could be linked to a *MouseDirectManipulation* modality using the *supportsModality* property. The modality on its turn is related to a *Mouse* individual using the *usesDevice* property and to the *DirectManipulation* interaction language using the *usesInteractionLanguage* property. Notice that for this to work, the *Mouse* individual has to be linked to *desktopPC1* using the *hasInteractionResource* property.

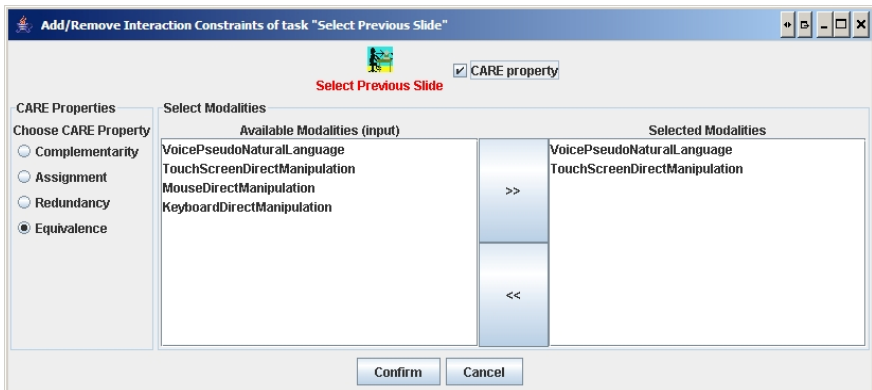


Fig. 4. Screenshot of the dialog box used to annotate a task with an interaction constraint

To enable designers to annotate Modality Interaction Constraints to the tasks, we have extended the DynaMo-AID design tool [7]. Fig. 4 shows the dialog box in the tool which inspects the type of task and queries the ontology in order to present the available modalities to the designer. If the task is an interaction task, input modalities will be shown to the designer, if the task is an application task, output modalities will appear in the *Available Modalities* part of the dialog box.

### 3 Runtime Support: Prototyping and Deployment of the User Interface

In the previous section we have described how designers can add information to a task model to describe which interaction modalities are appropriate to perform the tasks. In this section we discuss how this information can be used at runtime in order to enable runtime distribution of tasks among Interaction Devices.

#### 3.1 Overview of the Runtime Architecture

To support distribution we have extended our existing generic runtime architecture supporting model-based designed user interfaces influenced by context changes.

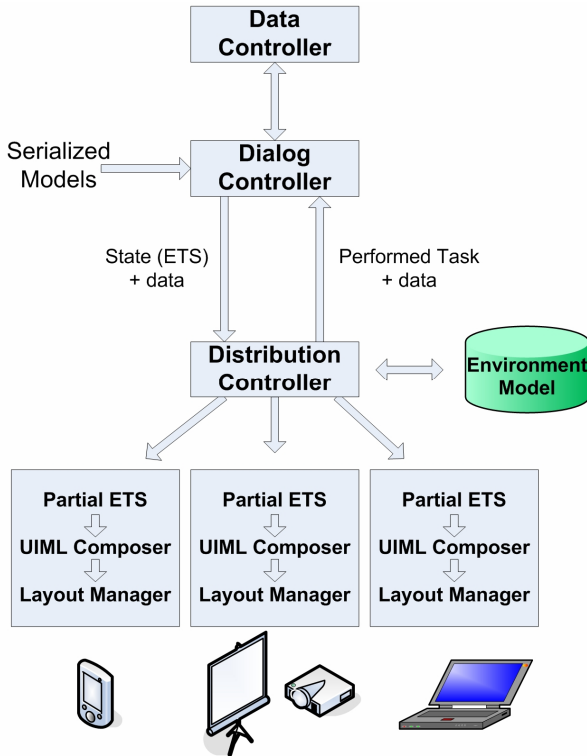


Fig. 5. Overview of the extensions of the runtime architecture

Fig. 5 shows an overview of the extensions we have applied. The models described in the previous section can be constructed in the design tool and can be serialized to an XML-based format. These models are the input of the runtime architecture. The *Dialog Controller* takes into account the dialog model to keep track of the current state of the user interface. The current state implies which tasks are enabled at the current time (section 2.1). The *Data Controller* keeps track of the data presented in the user interface and takes care of the communication with the functional core of the system.

When the user interface is started the environment is scanned for devices. The Universal Plug and Play<sup>1</sup> standard is used to discover devices in the vicinity. Each device broadcasts a device profile mentioning the available Interaction Resources supported by the device. Taking this information into account an *Environment Model* can be constructed. This environment model contains the whereabouts of the Interaction Devices and the available Interaction Resources for each device. When the environment model is constructed, the dialog controller will load the first state in accordance with the starting state of the State Transition Network. The active state thus corresponds to the tasks that are enabled when the system is started. This information is passed on to the *Distribution Controller* along with the data related to these tasks as provided by the data controller. The distribution controller will then seek for each task an appropriate Interaction Device containing an Interaction Resource that supports the interaction modalities related to the tasks. The distribution controller will then group the tasks by Interaction Device, resulting in Partial Enabled Task Sets (groups of tasks enabled during the same period of time and deployed on the same Interaction Device). Afterwards the Abstract Interaction Objects related to the tasks of the Partial Enabled Task Set are grouped and are transformed to a UIML<sup>2</sup> document. Behaviour information is added to the UIML document to be able to communicate with the renderer and an automatic layout manager will add layout constraints that can be interpreted by the rendering engine.

### 3.2 Rendering Engine

Fig. 6 shows an overview of the rendering architecture consisting of three layers: the Distribution Controller, the Presentation Manager (a servlet) and the clients. Whenever the presentation of the user interface needs an update, e.g. when a new state has to be deployed or when a user interface update occurs, the Distribution Controller sends a *notifyClient* message to one or multiple clients (depending on the distribution plan, section 3.3) using the InteractionDevice Proxy that is connected to the Client. As a response to this message, the client browsers are redirected to the URL where the Presentation Manager servlet awaits client requests (A.1 and B.1, HTTP). These requests can be of different types (according to the information in the *notifyClient* message):

---

<sup>1</sup> <http://www.upnp.org>

<sup>2</sup> <http://www.uiml.org>

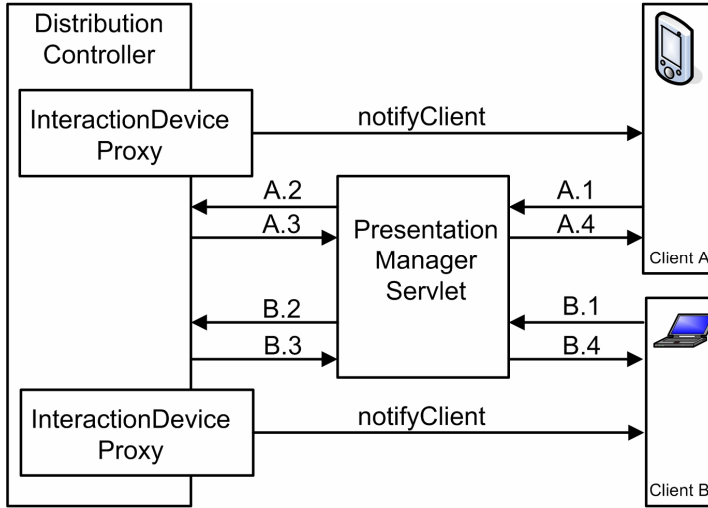


Fig. 6. Overview of the rendering architecture

- **requestUI:** requests a newly available presentation for the interaction device. After receiving the message, the Presentation Manager forwards the message to the Distribution Controller (A.2 and B.2) which responds by sending the UIML representation of the user interface and the data for this client to the Presentation Manager (A.3 and B.3). The Presentation Manager servlet now builds an internal PresentationStructure object for the user interface and stores the object in the current session. Depending on the modalities that should be supported, the presentation manager chooses the appropriate generator servlet, XplusVGeneratorServlet or XHTMLGeneratorServlet, that generates the concrete presentation and sends it as an HTTP response back to the client (A.4 and B.4). The task of the XplusVGenerator is to transform the PresentationStructure object to XHTML + VoiceXml (X+V<sup>3</sup>). X+V supports multimodal (Speech + GUI) interfaces and can be interpreted by multimodal browsers such as the ACCESS Systems' NetFront Multimodal Browser<sup>4</sup> The XHTMLGeneratorServlet transforms the PresentationStructure object to XHTML for interpretation by normal client browsers;
- **requestDataUpdate:** requests a data update for the current presentation. When the Presentation Manager servlet receives this message from a client it is forwarded to the Distribution Controller (A.2 and B.2) which sends the user interface data as a response (A.3 and B.3). Now the Presentation Manager updates the data in the PresentationStructure object available in the current session and chooses the appropriate generator servlet to generate the concrete user interface and to send it to the client browser (A.4 and B.4);

<sup>3</sup> <http://www.voicexml.org/specs/multimodal/x+v/12/>

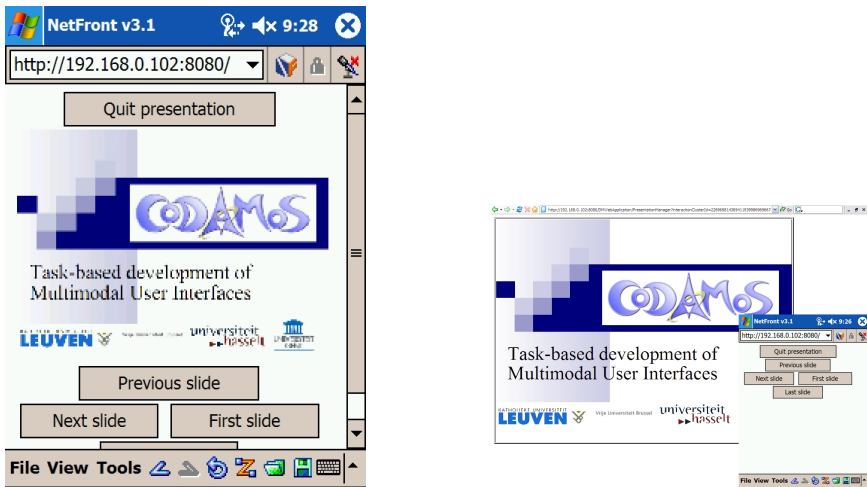
<sup>4</sup> <http://www-306.ibm.com/software/pervasive/multimodal/>



- **taskPerformed:** when a user interacts with the user interface, e.g. by clicking a button, a taskPerformed message together with the request parameters are sent to the Presentation Manager which forwards the message to the Distribution Controller.

Notice that the system follows a Model-View-Controller architecture. The Presentation Manager Servlet is the controller, the generator servlets are the views and the PresentationStructure is the model.

Fig. 7 shows what happens when generating the user interface for the task model in fig. 2. In (a), the user interface was deployed in an environment without an interaction device that supports the modality *ProjectorGUI*. This implies, the *Navigate Slides* and the *Show Slide* task are all deployed on a PDA using the X+V generator and a multimodal browser that supports X+V. This means we can navigate slides using voice by saying for example `Next Slide` or `First Slide`, or we can use the stylus to interact with the buttons. In (b) the user interface is distributed because we added a laptop attached to a projector to the environment. In this case the *Navigate Slides* tasks are still deployed on the PDA using the X+V generator. The *Show Slide* task however is deployed on the laptop screen using the XHTML generator and an XHTML browser.



**Fig. 7.** Example of Fig.2 rendered on a single PDA (a) and in an environment with a PDA and a desktop computer (b)

### 3.3 Constructing a Distribution Plan

In the two previous sections we talked about the structure of the runtime architecture and the rendering engine. However the question how to divide an enabled task set into a usable federation of partial enabled task sets has not yet been discussed. In this section we discuss the first approach we have implemented and some observed problems with this approach. Afterwards we propose a solution asking some extra modelling from the designer.

## Task-Device Mappings Using Query Transformations

In our first approach, we use a query language, SparQL<sup>5</sup>, to query the information in the environment model which is a runtime instantiation of the Interaction Environment Ontology (section 2.3). SparQL is a query language for RDF<sup>6</sup> and can be used to pose queries at ontologies modelled using the OWL<sup>7</sup> language.

```

PREFIX codamos: <http://edm.uhasselt.be/codamos#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?interactiondevice
WHERE { ?interactiondevice codamos:supportsModality ?m1 .
        ?interactiondevice codamos:supportsModality ?m2 .
        ?m1 rdf:type codamos:ProjectorGUI .
        ?m2 rdf:type codamos:TouchScreenGUI
}

```

(a)

```

PREFIX codamos: <http://edm.uhasselt.be/codamos#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?interactiondevice
WHERE { ?interactiondevice codamos:supportsModality ?m1 .
        ?interactiondevice codamos:supportsModality ?m2 .
        ?m1 rdf:type codamos:ProjectorGUI .
        ?m2 rdf:type codamos:TouchScreenGUI
}

```

(b)

```

PREFIX codamos: <http://edm.uhasselt.be/codamos#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?interactiondevice
WHERE { ?interactiondevice codamos:supportsModality ?m1 .
        ?interactiondevice codamos:supportsModality ?m2 .
        ?m1 rdf:type codamos:ProjectorGUI .
        ?m2 rdf:type codamos:TouchScreenGUI
}

```

(c)

**Fig. 8.** Queries deduced from the Modality Interaction Constraint related to the *Show Slide* task of the example in Fig. 2. Query (a) searches for a device supporting all the modalities in the equivalence relation. Queries (b) and (c) are reduced queries that are constructed if query (a) did not return a result.

To map each task of the enabled task set to the appropriate Interaction Device, the Modality Interaction Constraint related to task  $\tau$  will be transformed to a SparQL query. Fig. 8 shows an example of the mapping of the Modality Interaction Constraints attached to the *Show Slide* task of our previous example. This constraint says that modality  $m_4$  (ProjectorGUI) and modality  $m_1$  (ScreenGUI) are equivalent for this task. The more modalities in the equivalence relation are supported by the interaction device, the better suited it will be for executing the task. This is what the query in Fig. 8(a) tries to achieve. In this query, an interaction device which supports both modalities is  $m_4$  and  $m_1$  searched for and when it is found, the task is deployed on the device. Now suppose we have a Desktop device in the environment attached to a projector but not to a screen. This means the Desktop supports the ProjectorGUI modality only. The query in Fig. 8(a) will return no interaction device. As a result the system will reduce the query to find a device that supports only one specified modality. In this case this is feasible because the constraint defines an equivalence relation so the devices supporting only one (or more) of the required modalities will also be able to handle the task. The first query that will be tried is the query in Fig. 8(b) because the ProjectorGUI modality is defined first in the modality

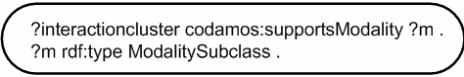
<sup>5</sup> <http://www.w3.org/TR/rdf-sparql-query/>

<sup>6</sup> <http://www.w3.org/RDF/>

<sup>7</sup> <http://www.w3.org/TR/owl-features/>

constraint. Because we have a Desktop individual in the environment which supports this modality, it will be returned and the task is deployed on the device. If such a device is still not found, the system will try the query in Fig.8 (c) after which the task is deployed on a device with a Screen attached.

Notice that the queries in Fig. 8 easily extend to support the other three CARE properties by adding/removing rules such as the one presented in Fig. 9. The ModalitySubClass in the query can be one of the leaf Modality subclasses. In case of the Assignment relation this is easy because we want to select a device supporting only one modality. Complementarity is analogue to the Equivalence relation. However, here all the modalities in the relation should be supported by the interaction device. In case of the Redundancy relation rules are added for each redundant modality.



```

?interactioncluster codamos:supportsModality ?m .
?m rdf:type ModalitySubclass .

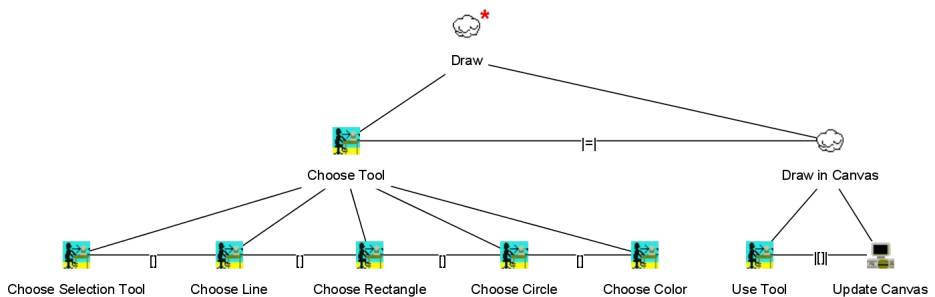
```

**Fig. 9.** Extension rule for generating SparQL queries from Modality Interaction Constraints

We can summarise our approach as the execution of queries searching for an appropriate device supporting the modalities according to the CARE property relating the modalities. Priority for the execution of the queries is given to the modality specified first in the CARE relation (e.g. *ProjectorGUI* in the example of Fig. 8).

### Partial Enabled Task Set Refinements

We now have presented a way to use de Modality Interaction Constraints to divide an enable task sets into partial enabled task sets for a feasible distribution. However this distribution is not always the best case scenario.



**Fig. 10.** Draw task of a drawing application

Consider the example in Fig. 10. This example shows the typical task in a drawing application where the user can choose a tool and use this tool to draw on a canvas using the direct manipulation paradigm. Suppose all the tasks are annotated with the same Modality Interaction Constraint:  $E(\text{MouseDirectManipulation}, \text{TouchScreenDirectManipulation})$ . This means the use of the *MouseDirect-Manipulation* modality is equivalent to the *TouchScreenDirectManipulation* modality. When we consider an

environment containing a desktop computer supporting the first modality and a PDA supporting the second modality, and we apply the approach described above, all the tasks will be assigned to the device supporting the first modality because neither device supports both. However in some cases a user might prefer to have the user interface distributed where the tasks concerning tool selection are deployed on the PDA and the large canvas is displayed on a desktop computer.

Another possible scenario could be a media player where the operation buttons are displayed on the user's cell phone and the actual media is playing on the user's PDA to maximize screen space for displaying the media. In order to know whether the user would prefer a maximal distribution of the tasks rather than a maximal combination of the tasks on one particular device, the user has to specify this in his/her user profile. In the latter case the approach discussed above where modalities are transformed to queries can be applied. When a maximal distribution is desirable, some more meta-information regarding the task composition should be necessary.

One way to solve this problem is to let the designer define *Task Set Constraints (TSC)* in the task model. These constraints enable the designer to specify which tasks are desirably grouped on the same Interaction Device, and which tasks are desirably not grouped together on the same Interaction Device. Applied to the example in Fig. 10 the designer can specify the subtasks of the *Choose Tool* tasks are desirably grouped together and these same tasks are desirably not grouped with the sub tasks of the *Draw in Canvas* task. Taking into account this information during the runtime, the distribution controller can decide to prioritise the break-up of the enabled task set even if deployment is possible on a single device according to the Modality Interaction Constraint if the property of maximal distribution is chosen.

## 4 Related Work

In this section we will discuss work related to our approach.

Berti et al. [3] describe a framework supporting migration of user interfaces from one platform to another. Unlike our goals they accentuate migratory interfaces where it is important that a user who is performing a certain task on one device can continue performing the same task on another device. In our approach we aim to distribute the subtasks a user is currently performing among several devices in the user's vicinity to exploit the available interaction resources. In their paper, they discuss three aspects to allow usable interaction of migratory interfaces that are also applicable to our focus:

- adaptability to the device's available Interaction Resources (our approach uses an ontology-based environment model);
- applying specified design criteria (allocation of devices is based on a designed augmented task model);
- and insurance of continuity of the task performance (the environment model can be updated and the allocation can be updated accordingly).

Furthermore they acknowledge the need for multimodal interaction to support smooth task execution.

Bandelloni et al. [2] also use interface migration as a starting point, but they extend their approach to support partial migration where only some parts of the user

interfaces are migrated to another device. In this way user interface distribution is accomplished. Migration and partial migration are executed by taking into account the source user interface, performing runtime task analysis, and finally deploying the updated concrete user interface on the target device(s). This is in contrast to our approach where first the environment is examined to determine which interaction resources are currently available, before mapping the abstract user interface description onto a concrete one. In this way at each state of the user interface an appropriate distribution among the interaction resources is achieved according to the available interaction resources.

Florins et al. [10] describe rules for splitting user interfaces being aimed at graceful degradation of user interfaces. Several algorithms are discussed to divide a complex user interface developed for a platform with few constraints in order to *degrade* the user interface with the purpose of presenting the interface in pieces to the user on a more constrained platform (e.g. with a smaller screen space). Although nothing is said about user interface distribution, these algorithms can be used in our approach complementary to the distribution plan discussed in 3.3.

CAMELEON-RT [1] is a reference model constructed to define the problem space of user interfaces released in ubiquitous computing environments. Their reference model covers user interface distribution, migration and plasticity [16]. This is also the problem domain of our approach. The work presents a conceptual middleware whereupon context-aware interactive systems can be deployed. The architecture is divided in several layers such as the platform layer, representing the hardware, the middleware layer, representing the software deducting the adaptation, and the interaction layer, where the interface is presented to the user in order to enable interaction with the system.

## 5 Conclusions and Future Work

In this paper we have described a development process where some decisions regarding user interface distribution and selection of modalities can be postponed to the runtime of the system. In this way the user interface can adapt to volatile environments because selection of devices and modalities accessible to the user's vicinity are taken into account. At the moment we are still performing some tests regarding the refinement of the division into partial enabled task sets. User tests are planned to find out whether the proposed information is enough to obtain a usable interface and whether more information regarding the user's preferences is needed.

In future work we will look at possibilities to extend the layout management. Since we are using XHTML in the rendering engine, Cascading Style Sheets<sup>8</sup> can be used to complement the layout management in obtaining a more visually attractive user interface. However, at the moment we have implemented a basic flow layout algorithm to align the graphical user interface components. We plan to use layout patterns which are commonly used in model-based user interface development, e.g. [15].

---

<sup>8</sup> <http://www.w3.org/Style/CSS/>

Another research direction we plan to follow in the future is the generalisation of the Modality Interaction Constraints to more general Interaction Constraints. The querying mechanism used at runtime, based on SparQL, can also be used at design time where designers can construct a more specific query than the one generated by the runtime architecture. However we have to deliberate about the drawbacks: constructing these queries is not straightforward thus a mediation tool has to be implemented to let a designer postulate the requirements about user interface distribution in a way a more complex query can be generated.

## Acknowledgements

Part of the research at EDM is funded by EFRO (European Fund for Regional Development), the Flemish Government and the Flemish Interdisciplinary institute for Broadband Technology (IBBT). The CoDAMoS (Context-Driven Adaptation of Mobile Services) project IWT 030320 is directly funded by the IWT (Flemish subsidy organization).

## References

1. Balme, L., Demeure, A., Barralon, N., Coutaz, J., Calvary, G.: Cameleon-rt: A software architecture reference model for distributed, migratable, and plastic user interfaces. In: Markopoulos, et al. (eds.) [11], pp. 291–302
2. Bandelloni, R., Paternò, F.: Flexible interface migration. In: *IUI 2004: Proceedings of the 9th international conference on Intelligent user interface*, pp. 148–155. ACM Press, New York (2004)
3. Berti, S., Paternò, F.: Migratory multimodal interfaces in multidevice environments. In: *ICMI 2005: Proceedings of the 7th international conference on Multimodal interfaces*, pp. 92–99. ACM Press, New York (2005)
4. Clerckx, T., Van den Bergh, J., Coninx, K.: Modeling multi-level context influence on the user interface. In: *PerCom Workshops*, pp. 57–61. IEEE Computer Society, Los Alamitos (2006)
5. Clerckx, T., Luyten, K., Coninx, K.: DynaMo-AID: A design process and a runtime architecture for dynamic model-based user interface development. In: Bastide, R., Palanque, P., Roth, J. (eds.) *DSV-IS 2004 and EHCI 2004*. LNCS, vol. 3425, pp. 77–95. Springer, Heidelberg (2005)
6. Clerckx, T., Vandervelpen, C., Luyten, K., Coninx, K.: A task-driven user interface architecture for ambient intelligent environments. In: *IUI 2006: Proceedings of the 11th international conference on Intelligent user interfaces*, pp. 309–311. ACM Press, New York (2006)
7. Clerckx, T., Winters, F., Coninx, K.: Tool Support for Designing Context-Sensitive User Interfaces using a Model-Based Approach. In: Dix, A., Dittmar, A. (eds.) *International Workshop on Task Models and Diagrams for user interface design 2005 (TAMODIA 2005)*, Gdansk, Poland, September 26–27, 2005, pp. 11–18 (2005)
8. Coutaz, J., Nigay, L., Salber, D., Blandford, A., May, J., Young, R.M.: Four easy pieces for assessing the usability of multimodal interaction: the care properties. In: Nordby, K., Helmersen, P.H., Gilmore, D.J., Arnesen, S.A. (eds.) *INTERACT, IFIP Conference Proceedings*, pp. 115–120. Chapman & Hall, Boca Raton (1995)

9. Dey, A.K.: Providing Architectural Support for Building Context-Aware Applications. PhD thesis, College of Computing, Georgia Institute of Technology (December 2000)
10. Florins, M., Simarro, F.M., Vanderdonckt, J., Michotte, B.: Splitting rules for graceful degradation of user interfaces. In: AVI 2006: Proceedings of the working conference on Advanced visual interfaces, pp. 59–66. ACM Press, New York (2006)
11. Markopoulos, P., Eggen, B., Aarts, E.H.L., Crowley, J.L.: EUSAI 2004. LNCS, vol. 3295. Springer, Heidelberg (2004)
12. Paternò, F.: Model-Based Design and Evaluation of Interactive Applications. Springer, Heidelberg (1999)
13. Paternò, F., Santoro, C.: One model, many interfaces. In: Kolski, C., Vanderdonckt, J. (eds.) CADUI, pp. 143–154. Kluwer, Dordrecht (2002)
14. Preuveneers, D., Van den Bergh, J., Wagelaar, D., Georges, A., Rigole, P., Clerckx, T., Berbers, Y., Coninx, K., Jonckers, V., De Bosschere, K.: Towards an extensible context ontology for ambient intelligence. In: Markopoulos, et al. (eds.) [11], pp. 148–159
15. Sinnig, D., Gaffar, A., Reichart, D., Seffah, A., Forbrig, P.: Patterns in model-based engineering. In: Jacob, R.J.K., Limbourg, Q., Vanderdonckt, J. (eds.) CADUI 2004, pp. 195–208. Kluwer, Dordrecht (2004)
16. Thevenin, D., Coutaz, J.: Plasticity of user interfaces: Framework and research agenda. In: Interact 1999, vol. 1, pp. 110–117. IFIP, IOS Press, Edinburgh (1999)
17. Vanderdonckt, J.M., Bodart, F.: Encapsulating knowledge for intelligent automatic interaction objects selection. In: CHI 1993: Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 424–429. ACM Press, New York (1993)
18. Vandervelpen, C., Coninx, K.: Towards model-based design support for distributed user interfaces. In: Proceedings of the third Nordic Conference on Human-Computer Interaction, pp. 61–70. ACM Press, New York (2004)
19. Weiser, M.: The Computer for the 21st Century. Scientific American (1991)

## Questions

### **Michael Harrison:**

*Question: You seem to have a static scheme. You do not deal with the possibility that the ambient noise level might change and therefore cause a change in the configuration. Would you not require a more procedural (task level) description to describe what to do in these different situations?*

Answer: It is a static technique. Extensions to CTT have been considered that relate to similar features of ubiquitous systems and it would be interesting to see how there could be an extension to deal with dynamic function allocation.

### **Laurence Nigay:**

*Question: We developed a tool called ICARE in Grenoble, describing ICARE diagrams for each elementary task of a CTT. We found it difficult to see the link between the task level and the ICARE description, the border is not so clean. Do you have the same problem?*

Answer: Depends on the granularity of the task model. When it is a rather abstract task, you have a different situation than when it is concrete. This is a factor that comes into play.

# A Comprehensive Model of Usability

Sebastian Winter, Stefan Wagner, and Florian Deissenboeck

Institut für Informatik  
Technische Universität München  
Boltzmannstr. 3, 85748 Garching b. München, Germany  
{winterse,wagnerst,deissenb}@in.tum.de

**Abstract.** Usability is a key quality attribute of successful software systems. Unfortunately, there is no common understanding of the factors influencing usability and their interrelations. Hence, the lack of a comprehensive basis for designing, analyzing, and improving user interfaces. This paper proposes a 2-dimensional model of usability that associates system properties with the activities carried out by the user. By separating activities and properties, sound quality criteria can be identified, thus facilitating statements concerning their interdependencies. This model is based on a tested quality meta-model that fosters preciseness and completeness. A case study demonstrates the manner by which such a model aids in revealing contradictions and omissions in existing usability standards. Furthermore, the model serves as a central and structured knowledge base for the entire quality assurance process, e.g. the automatic generation of guideline documents.

**Keywords:** Usability, quality models, quality assessment.

## 1 Introduction

There is a variety of standards concerning the quality attribute *usability* or *quality in use* [1, 2]. Although in general all these standards point in the same direction, due to different intuitive understandings of usability, they render it difficult to analyze, measure, and improve the usability of a system. A similar situation also exists for other quality attributes, e.g. *reliability* or *maintainability*. One possibility to address this problem is to build a comprehensive model of the quality attribute. Most models take recourse to the decomposition of quality proposed by Boehm et al. [3]. However, this decomposition is still too abstract and imprecise to be used concretely for analysis and measurement.

More comprehensive models have been proposed for product quality in general [4] or even usability [5]. However, these models have three problems: First, they do not decompose the attributes and criteria to a level that is suitable for actually assessing them for a system. Secondly, these models tend to omit rationale of the required properties of the system. Thirdly, the dimensions used in these models are heterogeneous, e.g. the criteria mix properties of the system with properties of the user. The first problem constrains the use of these models as the basis for analyses. The second one makes it difficult to describe impacts precisely and therefore to



convince developers to use it. The third problem hampers the revelation of omissions and inconsistencies in these models. The approach to quality modeling by Broy, Deissenboeck, and Pizka [6] is one way to deal with these problems. Using an explicit meta-model, it decomposes quality into system properties and their impact on activities carried out by the user. This facilitates a more structured and uniform means of modeling quality.

*Problem.* Although usability is a key quality attribute in modern software systems, the general understanding of its governing factors is still not good enough for profound analysis and improvement. Moreover, currently there are no comprehensive objective criteria for evaluating usability.

*Contribution.* This paper proposes a comprehensive 2-dimensional model of usability based on a quality meta-model that facilitates a structured decomposition of usability and descriptions of the impacts of various facts of the system. This kind of model has proven to be useful for the quality attribute maintainability [6]. Several benefits can be derived by using this type of model:

1. The ability to reveal omissions and contradictions in current models and guidelines.
2. The ability to generate guidelines for specific tasks automatically.
3. A basis for (automatic) analysis and measurement.
4. The provision of an interface with other quality models and quality attributes.

We demonstrate the applicability of the 2-dimensional model in a case study of the ISO 15005 [7] which involves domain-specific refinements. By means of this model we are able to identify several omissions in the standard and suggest improvements.

*Consequences.* Based on the fact that we can pinpoint omissions and inconsistencies in existing quality models and guidelines, it seems advisable to use an explicit meta-model for usability models, precisely to avoid the weaknesses of the other approaches. Furthermore, it helps to identify homogeneous dimensions for the usability modeling. We believe that our model of usability is a suitable basis for domain- or company-specific models that must be structured and consistent.

*Outline.* In Sec. 2 we describe prior work in the area of quality models for usability and the advances and shortcomings it represents. In Sec. 3, using an explicit meta-model, we discuss the quality modeling approach. The 2-dimensional model of usability that we constructed using this approach is presented in Sec. 4. This model is refined to a specific model based on an ISO standard in the case study of Sec. 5. The approach and the case study are discussed in Sec. 6. In Sec. 7 we present our final conclusions.

## 2 Related Work

This section describes work in the area of quality models for usability. We discuss general quality models, principles and guidelines, and first attempts to consolidate the quality models.

## 2.1 Quality Models for Usability

Hierarchical structures as quality models which focus mainly on *quality assurance* have been developed. A model first used by Boehm [3] and McCall et al. [8] consists of three layers: factors, criteria, and metrics. Consequently, the approach is referred to as the factor-criteria-metrics model (FCM model). The high-level factors model the main quality goals. These factors are divided into criteria and sub-criteria. When a criterion has not been divided, a metric is defined to measure the criteria. However, this kind of decomposition is too abstract and imprecise to be used for analysis and measurement. In addition, since usability is not a part of the main focus, this factor is not discussed in detail.

In order to provide means for the operational measurement of usability several attempts have been made in the domain *human-computer interaction* (HCI). Prominent examples are the models from Shackel and Richardson [9] or Nielsen [10]. Nielsen, for example, understands usability as a property with several dimensions, each consisting of different components. He uses five factors: *learnability*, *efficiency*, *memorability*, *errors*, and *satisfaction*. *Learnability* expresses how well a novice user can use the system, while the efficient use of the system by an expert is expressed by *efficiency*. If the system is used occasionally the factor *memorability* is used. This factor differentiates itself from *learnability* by the fact that the user has understood the system previously. Nielsen also mentions that the different factors can conflict with each other.

The ISO has published a number of standards which contain usability models for the operational evaluation of usability. The ISO 9126-1 [11] model consists of two parts. The first part models the *internal* as well as the *external* quality, the second part the *quality in use*. The first part describes six characteristics which are further divided into sub-characteristics. These measurable attributes can be observed during the use of the product. The second part describes attributes for *quality in use*. These attributes are influenced by all six product characteristics. Metrics are given for the assessment of the sub-characteristics. It is important to note that the standard does not look beyond the sub-characteristics intentionally.

The ISO 9241 describes human-factor requirements for the use of software systems with user interface. The ISO 9241-11 [12] provides a framework for the evaluation of a running software system. The framework includes the context of use and describes three basic dimensions of usability: *efficiency*, *effectiveness*, and *satisfaction*.

## 2.2 Principles and Guidelines

In addition to the models which define usability operationally, a lot of design principles have been developed. Usability principles are derived from knowledge of the HCI domain and serve as a design aid for the designer. For example, the “eight golden rules of dialogue design” from Shneiderman [13] propose rules that have a positive effect on usability. One of the rules, namely *strive for consistency*, has been criticized by Grudin [14] for its abstractness. Grudin shows that consistency can be decomposed into three parts that also can be in conflict with each other. Although Grudin does not offer an alternative model, he points out the limitations of the design guidelines.

Dix et al. [15] argue as well that if principles are defined in an abstract and general manner, they do not help the designer. In order to provide a structure for a comprehensive catalogue of usability principles Dix et al. [15] divide the factors which support the usability of a system into three categories: *learnability*, *flexibility*, and *robustness*. Each category is further divided into sub-factors. The ISO 9241-110 [16] takes a similar approach and describes seven high-level principles for the design of dialogues: *suitability for the task*, *self-descriptiveness*, *controllability*, *conformity with user expectations*, *error tolerance*, *suitability for individualization*, and *suitability for learning*. These principles are not independent of each other and some principles have an impact on other principles. For example *self-descriptiveness* influences *suitability for learning*. Some principles have a part-of relation to other principles. For example, *suitability for individualization* is a part of *controllability*. The standard does not discuss the relations between the principles and gives little information on how the principles are related to the overall framework given in [12].

### 2.3 Consolidated Quality Models for Usability

There are approaches which aim to consolidate the different models. Seffah et al. [5] applied the FCM model to the quality attribute *usability*. The developed model contains 10 factors which are subdivided into 26 criteria. For the measurement of the criteria the model provides 127 metrics.

The motivation behind this model is the high abstraction and lack of aids for the interpretation of metrics in the existing hierarchically-based models. Put somewhat differently, the description of the relation between metrics and high-level factors is missing. In addition, the relations between factors, e.g. *learnability* vs. *understandability*, are not described in the existing models. Seffah et al. [5] also criticize the difficulty in determining how factors relate to each other, if a project uses different models. This complicates the selection of factors for defining high-level management goals. Therefore, in [5] a consolidated model that is called *quality in use integrated measurement model* (QUIM model) is developed.

Since the FCM decomposition doesn't provide any means for precise structuring, the factors used in the QUIM model are not independent. For example, *learnability* can be expressed with the factors *efficiency* and *effectiveness* [12].

The same problem arises with the criteria in the level below the factors: They contain attributes as well as principles, e.g. *minimal memory load*, which is a principle, and *consistency* which is an attribute. They contain attributes about the user (*likeability*) as well as attributes about the product (*attractiveness*). And lastly, they contain attributes that are similar, e.g. *appropriateness* and *consistency*, both of which are defined in the paper as capable of indicating whether visual metaphors are meaningful or not.

To describe how the architecture of a software system influences usability, Folmer and Bosch [17] developed a framework to model the quality attributes related to usability. The framework is structured in four layers. The high-level layer contains *usability definitions*, i.e. common factors like *efficiency*. The second layer describes concrete measurable *indicators* which are related to the high-level factors. Examples of indicators are *time to learn*, *speed*, or *errors*. The third layer consists of *usability properties* which are higher level concepts derived from design principles like *provide*

*feedback*. The lowest layer describes the *design knowledge* in the community. Design heuristics, e.g. the *undo pattern*, are mapped to the *usability properties*. Van Welie [18] also approaches the problem by means of a layered model. The main difficulty with layered models is the loss of the exact impact to the element on the high-level layer at the general principle level when a design property is first mapped to a general principle.

Based on Norman's action model [19] Andre et al. developed the USER ACTION FRAMEWORK [20]. This framework aims toward a structured knowledge base of usability concepts which provides a means to classify, document, and report usability problems. By contrast, our approach models system properties and their impact on activities.

## 2.4 Summary

As pointed out, existing quality models generally suffer from one or more of the following shortcomings:

1. *Assessability*. Most quality models contain a number of criteria that are too coarse-grained to be assessed directly. An example is the *attractiveness* criterion defined by the ISO 9126-1 [11]. Although there might be some intuitive understanding of attractiveness, this model clearly lacks a precise definition and hence a means to assess it.
2. *Justification*. Additionally, most existing quality models fail to give a detailed account of the impact that specific criteria (or metrics) have on the user interaction. Again the ISO standard cited above is a good example for this problem, since it does not provide any explanation for the presented metrics. Although consolidated models advance on this by providing a more detailed presentation of the relations between criteria and factors, they still lack the desired degree of detail. An example is the relationship between the criterion *feedback* and the factor *universality* presented in [5]. Although these two items are certainly related, the precise nature of the relation is unclear.
3. *Homogeneity*. Due to a lack of clear separation of different aspect of quality most existing models exhibit inhomogeneous sets of quality criteria. An example is the set of criteria presented in [5] as it mixes attributes like *consistency* with mechanisms like *feedback* and principles like *minimum memory load*.

## 3 A 2-Dimensional Approach to Model Quality

To address the problems with those quality models described in the previous section we developed the novel two-dimensional quality meta-model QMM. This meta-model was originally based on our experience with modeling maintainability [6], but now also serves as a formal specification for quality models covering different quality attributes like *usability* and *reliability*. By using an explicit meta-model we ensure the well-structuredness of these model instances and foster their preciseness as well as completeness.

### 3.1 The 2-Dimensional Quality Meta-model

This model is based on the general idea of hierarchical models like FCM, i.e. the breaking down of fuzzy criteria like *learnability* into sub-criteria that are tangible enough to be assessed directly. In contrast to other models, it introduces a rigorous separation of system *properties* and *activities* to be able to describe quality attributes and their impact on the usage of a software product precisely.

This approach is based on the finding that numerous criteria typically associated with usability, e.g. *learnability*, *understandability*, and of course *usability* itself, do not actually describe the properties of a system but rather the activities performed on (or with) the system. It might be objected that these activities are merely expressed in the form of adjectives. We argue, by contrast, that this leads precisely to the most prevalent difficulty of most existing quality models, namely to a dangerous mixture of activities and actual system properties. A typical example of this problem can be found in [5] where *time behavior* and *navigability* are presented as the same type of criteria. Where *navigability* clearly refers to the navigation activity carried out by the user of the system, *time behavior* is a property of the system and not an activity. One can imagine that this distinction becomes crucial, if the usability of a system is to be evaluated regarding different types of users: The way a user navigates is surely influenced by the system, but is also determined by the individuality of the user. In contrast, the response times of systems are absolutely independent of the user. A simplified visualization of the system property and activity decompositions as well as their interrelations is shown in Fig. 1. The activities are based on Norman’s action model [19]. The whole model is described in detail in Sec. 4.

The final goal of usability engineering is to improve the *usage* of a system, i.e. to create systems that support the activities that the user performs on the system. Therefore, we claim that usability quality models must not only feature these activities as first-class citizens, but also precisely describe how properties of the system influence them and therewith ultimately determine the usability of the system.

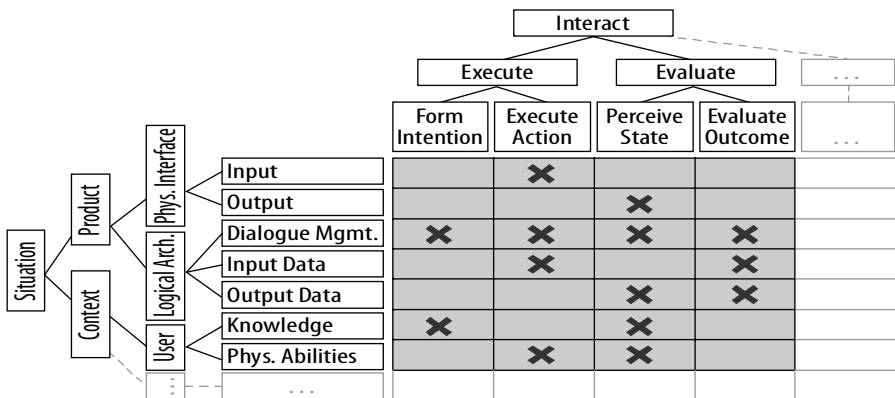


Fig. 1. Simplified quality model

### 3.2 Facts, Activities, Attributes, and Impacts

Our usability model does not only describe the product, i.e. the user interface, itself, but also comprises all relevant information about the situation of use (incl. the user). To render this description more precisely the model distinguishes between *facts* and *attributes*. Facts serve as a means to describe the situation of use in a hierarchical manner but do not contain quality criteria. For example, they merely model that the fact *user interface* consists of the sub-facts *visual interface* and *aural interface*.

*Attributes* are used to equip the facts with desired or undesired low-level quality criteria like *consistency*, *ambiguousness*, or even the simple attribute *existence*. Thus, tuples of facts and attributes express system properties. An example is the tuple [Font Face | CONSISTENCY] that describes the consistent usage of font faces throughout the user interface. Please note, that for clarity's sake the attributes are not shown in Fig. 1.

The other part of the model consists of a hierarchical decomposition of the activities performed by a user as part of the interaction with the system. Accordingly, the root node of this tree is the activity *interact* that is subdivided into activities like *execute* and *evaluate* which in turn are broken down into more specific sub-activities.

Similar to facts, activities are equipped with attributes. This allows us to distinguish between different properties of the activities and thereby fosters model preciseness. Attributes typically used for activities are *duration* and *probability of error*. The complete list of attributes is described in Sec. 4.

The combination of these three concepts enables us to pinpoint the impact that properties of the user interface (plus further aspects of the situation of use) have on the user interaction. Here impacts are always expressed as a relation between fact-attribute-tuples and activity-attribute-tuples and qualified with the direction of the impact (positive or negative):

$$[\text{Fact } f \mid \text{ATTRIBUTE } A_1] \rightarrow +/\text{-} [\text{Activity } a \mid \text{ATTRIBUTE } A_2]$$

For example, one would use the following impact description

$$[\text{Font Face} \mid \text{CONSISTENCY}] \rightarrow - [\text{Reading} \mid \text{DURATION}]$$

to express that the consistent usage of font faces has a positive impact on the time needed to read the text. Similarly the impact

$$[\text{Input Validity Checks} \mid \text{EXISTENCE}] \rightarrow - [\text{Data Input} \mid \text{PROBABILITY OF ERROR}]$$

is used to explain that the existence of validity checks for the input reduces the likelihood of an error.

### 3.3 Tool Support

Our quality models are of substantial size (e.g. the current model for maintainability has > 800 model elements) due to the high level of detail. We see this as a necessity and not a problem, since these models describe very complex circumstances. However, we are well aware that models of this size can only be managed with proper tool support. We have therefore developed a graphical editor, based on the ECLIPSE platform<sup>1</sup> that supports quality engineers in creating models and in adapting these

---

<sup>1</sup> <http://www.eclipse.org>

models to changing quality needs by refactoring functionality<sup>2</sup>. Additionally, the editor provides quality checks on the quality models themselves, e.g. it warns about facts that do not have an impact on any activity.

For the distribution of quality models the editor provides an export mechanism that facilitates exporting models (or parts thereof) to different target formats. Supported formats are, e.g., simple graphs that illustrate the activity and system decomposition, but also full-fledged quality guideline documents that serve as the basis for quality reviews. This export functionality can be extended via a plug-in interface.

## 4 Usability Quality Model

Based on the critique of existing usability models described in Sec. 2 and using the quality modeling approach based on the meta-model from Sec. 3, we propose a 2-dimensional quality model for usability. The complete model is too large to be described in total, but we will highlight specific core parts of the model to show the main ideas.

Our approach to quality modeling includes *high-level* and *specific* models. The aim of the high-level model is to define a basic set of facts, attributes, and activities that are independent of specific processes and domains. It is simultaneously abstract and general enough to be reusable in various companies and for various products. In order to fit to specific projects and situations the high-level models are refined and tailored into specific models.

### 4.1 Goals

In accordance with existing standards [21], we see four basic principles needed for defining usability:

- *Efficiency*. The utilization of resources.
- *Effectiveness*. The sharing of successful tasks.
- *Satisfaction*. The enjoyment of product use.
- *Safety*. The assurance of non-harmful behavior.

Frøkjær, Hertzum, and Hornbæk [22] support the importance of these aspects: “Unless domain specific studies suggest otherwise, effectiveness, efficiency, and satisfaction should be considered independent aspects of usability and all be included in usability testing.” However, we do not use these principles directly for analysis, but rather to define the usability goals of the system. The goals are split into several attributes of the activities inside the model. For example, the effectiveness of the user interface depends on the probability of error for all activities of usage. Therefore, all impacts on the attribute *probability of error* of activities are impacts on the effectiveness and efficiency. We describe more examples below after first presenting the most important facts, activity trees, and attributes.

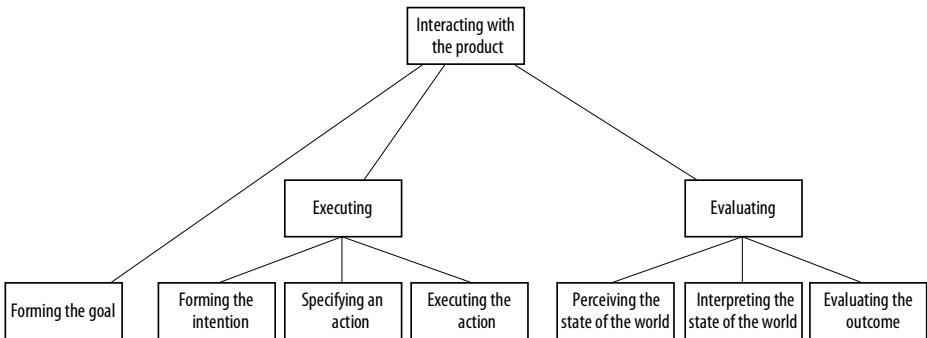
---

<sup>2</sup> A beta version of the editor can be downloaded from <http://www4.cs.tum.edu/~ccsm/qmm>

## 4.2 The Activity Subtree “Interacting with the Product”

The activity tree in the usability model has the root node *use* that denotes any kind of usage of the software-based system under consideration. It has two children, namely *execution of secondary tasks* and *interacting with the product*. The former stands for all additional tasks a user has that are not directly related to the software product. The latter is more interesting in our context because it describes the interaction with the software itself. We provide a more detailed explanation of this subtree in the following.

**Activities.** The activity *interacting with the product* is further decomposed, based on the seven stages of action from Norman [19] that we arranged in a tree structure (Fig. 2). We believe that this decomposition is the key for a better understanding of the relationships in usability engineering. Different system properties can have very different influences on different aspects of the use of the system. Only if these are clearly separated will we be able to derive well-founded analyses. The three activities, *forming the goal*, *executing*, and *evaluating*, comprise the first layer of decomposition. The first activity is the mental activity of deciding which goal the user wants to achieve. The second activity refers to the actual action of planning and realizing the task. Finally, the third activity stands for the gathering of information about the world’s state and understanding the outcome.



**Fig. 2.** The subtree for “Interacting with the Product” (adapted from [19])

The *executing* node has again three children: First, the user forms his intention to do a specific action. Secondly, the action is specified, i.e. it is determined what is to be done. Thirdly, the action is executed. The *evaluating* node is decomposed into three mental activities: The user perceives the state of the world that exists after executing the action. This observation is then interpreted by the user and, based on this, the outcome of the performed action is evaluated. Scholars often use and adapt this model of action. For example, Sutcliffe [23] linked error types to the different stages of action and Andre et al. [20] developed the USER ACTION FRAMEWORK based on this model.



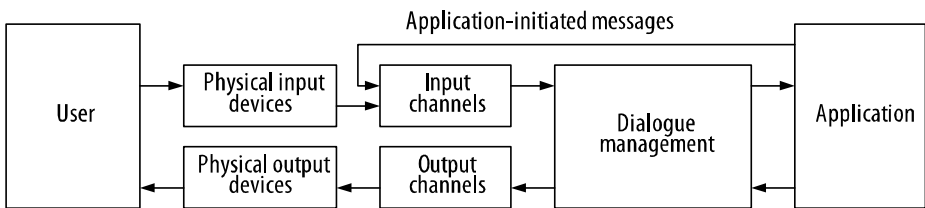
**Attributes.** To be able to define the relation of the facts and activities to the general usability goals defined above, such as *efficiency* or *effectiveness*, we need to describe additional properties of the activities. This is done by a simple set of attributes that is associated with the activities:

- *Frequency.* The number of occurrences of a task.
- *Duration.* The amount of time a task requires.
- *Physical stress.* The amount of physical requirements necessary to perform a task.
- *Cognitive load.* The amount of mental requirements necessary to perform a task.
- *Probability of error.* The distribution of successful and erroneous performances of a task.

As discussed in Sec. 4.1, these activity attributes can be used to analyze the usability goals defined during requirements engineering. We already argued that the effectiveness of a user interface is actually determined by the probability of error of the user tasks. In our model, we can explicitly model which facts and situations have an impact on that. The efficiency sets the frequency of an activity into relation to a type of resources: time (duration), physical stress, or cognitive load. We can explicitly model the impacts on the efficiency of these resources. Further attributes can be used to assess other goals.

### 4.3 The Fact Subtree “Logical User Interface”

The fact tree in the usability model contains several areas that need to be considered in usability engineering, such as the physical user interface or the usage context. By means of the *user* component, important properties of the user can be described. Together with the *application* it forms the context of use. The *physical output devices* and the *physical input devices* are assumed to be part of the physical user interface. However, we concentrate on a part we consider very important: the *logical user interface*. The decomposition follows mainly the logical architecture of a user interface as shown in Fig. 3.



**Fig. 3.** The user interface architecture

**Facts.** The logical user interface contains *input channels*, *output channels*, and *dialogue management*. In addition to the architecture, we also add data that is sent via the channels explicitly: *input data* and *output data*. The architecture in Fig. 3 also contains a specialization of input data, *application-initiated messages*. These messages, which are sent by the *application*, report interrupts of the environment or the application itself to the *dialogue management* outside the normal response to inputs.

**Attributes.** The attributes play an important role in the quality model because they are the properties of the facts that can actually be assessed manually or automatically. It is interesting to note that it is a rather small set of attributes that is capable of describing the important properties of the facts. These attributes are also one main building block that can be reused in company- or domain-specific usability models. Moreover, we observe that the attributes used in the usability model differ only slightly from the ones contained in the maintainability model of [6]. Hence, there seems to be a common basic set of those attributes that is sufficient – in combination with facts – for quality modeling.

- *Existence.* The most basic attribute that we use is whether a fact exists or not. The pure existence of a fact can have a positive or negative impact on some activities.
- *Relevance.* When a fact is relevant, it means that it is appropriate and important in the context in which it is described.
- *Unambiguousness.* An unambiguous fact is precise and clear. This is often important for information or user interface elements that need to be clearly interpreted.
- *Simplicity.* For various facts it is important that in some contexts they are simple. This often means something similar to small and straightforward.
- *Conformity.* There are two kinds of conformity: conformity to existing standards and guidelines, and conformity to the expectations of the user. In both cases the fact conforms to something else, i.e. it respects and follows the rules or models that exist.
- *Consistency.* There are also two kinds of consistency: internal consistency and external consistency. The internal consistency means that the entire product follows the same rules and logic. The external consistency aims at correspondence with external facts, such as analogies, or a common understanding of things. In both cases it describes a kind of homogeneous behavior.
- *Controllability.* A controllable fact is a fact which relates to behavior that can be strongly influenced by the actions of the user. The user can control its behavior.
- *Customizability.* A customizable fact is similar to a controllable fact in the sense that the user can change it. However, a customizable fact can be preset and fixed to the needs and preferences of the user.
- *Guardedness.* In contrast to customizability and controllability, a guarded fact cannot be adjusted by the user. This is a desirable property for some critical parts of the system.
- *Adaptability.* An adaptive fact is able to adjust to the user's needs or to its context dependent on the context information. The main difference to customizability is that an adaptive fact functions without the explicit input of the user.

#### 4.4 Examples

The entire model is composed of the activities with attributes, the facts with the corresponding attributes and the impacts between attributed facts and attributed activities. The model with all these details is too large to be described in detail, but we present some interesting examples: triplets of an attributed fact, an attributed activity, and a corresponding impact. These examples aim to demonstrate the structuring that can be achieved by using the quality meta-model as described in Sec. 3.

*Consistent Dialogue Management.* A central component in the logical user interface concept proposed in Sec. 4.3 is the *dialogue management*. It controls the dynamic exchange of information between the product and the user. In the activities tree, the important activity is carried out by the user by interpreting the information given by the user interface. One attribute of the dialogue management that has an impact on the interpretation is its *internal consistency*. This means that its usage concepts are similar in the entire dialogue management component. The corresponding impact description:

[Dialogue Management | INTERNAL CONSISTENCY] → – [Interpretation | PROB. OF ERROR]

Obviously, this is still too abstract to be easily assessed. This is the point where company-specific usability models come in. This general relationship needs to be refined for the specific context. For example, menus in a graphical user interface should always open the same way.

*Guarded Physical Interface.* The usability model does not only contain the logical user interface concept, but also the physical user interface. The *physical interface* refers to all the hardware parts that the user interacts with in order to communicate with the software-based system. One important attribute of such a physical interface is *guardedness*. This means that the parts of the interface must be guarded against unintentional activation. Hence, the guardedness of a physical interface has a positive impact on the *executing* activity:

[Physical Interface | GUARDEDNESS] → – [Executing | PROBABILITY OF ERROR]

A physical interface that is not often guarded is the touchpad of a notebook computer. Due to its nearness to the location of the hands while typing, the cursor might move unintentionally. Therefore, a usability model of a notebook computer should contain the triplet that describes the impact of whether the touchpad is guarded against unintentional operation or not.

## 5 Case Study: Modeling the ISO 15005

To evaluate our usability modeling approach we refine the high-level model described in Sec. 4 into a specific model based on the ISO 15005 [7]. This standard describes ergonomic principles for the design of *transport information and control systems* (TICS). Examples for TICS are driver information systems (e.g. navigation systems) and driver assistance systems (e.g. cruise control). In particular, principles related to dialogues are provided, since the design of TICS must take into consideration that a TICS is used in addition to the driving activity itself.

The standard describes three main *principles* which are further subdivided into eight *sub-principles*. Each sub-principle is motivated and consists of a number of *requirements* and/or *recommendations*. For each requirement or recommendation a number of examples are given.

For example, the main principle *suitability for use while driving* is decomposed among others into the sub-principle *simplicity*, i.e. the need to limit the amount of information to the task-dependent minimum. This sub-principle consists, among others, of the recommendation to optimize the driver's mental and physical effort. All in all the standard consists of 13 requirements, 16 recommendations, and 80 examples.

## 5.1 Approach

We follow two goals when applying our method to the standard: First, we want to prove that our high-level usability model can be refined to model such principles. Secondly, we want to discover inconsistencies, ill-structuredness, and implicitness of important information.

Our approach models every element of the standard (e.g. high-level principles, requirements, etc.) by refinement of the high-level model. For this, the meta-model elements (e.g. facts, attributes, impacts, etc.) are used. We develop the specific model by means of the tool described in Sec. 3.3. The final specific model consists of 41 facts, 12 activities, 15 attributes, 48 attributed facts, and 51 impacts.

## 5.2 Examples

To illustrate how the elements of the standard are represented in our specific model, we present the following examples.

*Representation of Output Data.* An element in the logical user interface concept proposed in Sec. 4.3 is the *output* data, i.e. the information sent to the driver. A central aspect is the representation of the data. One attribute of the representation that has an impact on the interpretation of the state of the system is its *unambiguouslyness*, i.e. that the representation is precise and clear. This is especially important so that the driver can identify the exact priority of the data. For example, warning messages are represented in a way that they are clearly distinguishable from status messages.

[Output Data | UNAMBIGUOUSNESS] → – [Interpretation | PROBABILITY OF ERROR]

Another attribute of the representation that has an impact on the interpretation is the *internal consistency*. If the representations of the output data follow the same rules and logic, it is easier for the driver to create a mental model of the system. The ease of creating a mental model has a strong impact on the ease of interpreting the state of the system:

[Output Data | INTERNAL CONSISTENCY] → – [Interpretation | DURATION]

One attribute of the representation that has an impact on the perception is *simplicity*. It is important for the representation to be simple, since this makes it easier for the driver to perceive the information:

[Output Data | SIMPLICITY] → – [Perception | COGNITIVE LOAD]

*Guarded Feature.* A TICS consists of several features which must not be used while driving the vehicle. This is determined by the manufacturer as well as by regulations. One important attribute of such features is its *guardedness*. This means that the feature is inoperable while the vehicle is moving. This protects the driver from becoming distracted while using the feature. The guardedness of certain features has a positive impact on the *driving* activity:

[Television | GUARDEDNESS] → – [Driving | PROBABILITY OF ERROR]

### 5.3 Observations and Improvements

As a result of the meta-model-based analysis, we found the following inconsistencies and omissions:

*Inconsistent Main Principles.* One of the three main principles, namely *suitability* for the *driver*, does not describe any activity. The other two principles use the activities to define the high-level usability goals of the system. For example, one important high-level goal is that the TICS dialogues do not interfere with the driving activity. Hence, we suggest that every main principle should describe an activity and the high-level goals of usability should be defined by means of the attributes of the user's activities.

*Mixed Sub-Principles.* The aspects described by the sub-principles are mixed: Three sub-principles describe activities without impacts, three describe facts without impacts, and the remaining two describe impacts of attributes on activities. This mix-up of the aspects described by the sub-principles must be resolved.

We believe that in order to make a design decision it is crucial for the software engineer to know which high-level goals will be influenced by it. Sub-principles which only describe attributes of system entities do not contribute toward design decisions. The same holds true for sub-principles which only describe activities, since they are not related to system entities. For this reason we suggest that all sub-principles that only describe activities should be situated at the main principle level, while those sub-principles that describe software entities should be situated at the requirement level.

*Requirements with Implicit Impacts.* 9 out of 13 requirements do not explicitly describe impacts on activities. Requirements serve to define the properties which the system entities should fulfill. If a requirement does not explicitly describe its impacts on activities, the impact could be misunderstood by the software engineer. Hence, we suggest that requirements should be described by attributed facts and their impacts on activities.

*Incomplete Examples.* 14 out of 80 examples only describe facts and their attributes, leaving the impacts and activities implicit. To provide complete examples we suggest that the examples should be described with explicit impacts and activities.

## 6 Discussion

The usability model acts as a central knowledge base for the usability-related relationships in the product and process. It documents in a structured manner how the properties of the system, team, and organization influence different usage activities. Therefore, it is a well-suited basis for quality assurance (QA). It can be used in several ways for constructive as well as analytical QA. Some of these have been shown to be useful in an industrial context w.r.t. maintainability models.

*Constructive QA.* The knowledge documented in the quality model aids all developers and designers in acquiring a common understanding of the domain, techniques, and influences. This common understanding helps to avoid misunderstandings, and

improvements to the quality model become part of a continuous learning process for all developers. For example, by describing the properties of the system artifacts, a glossary or terminology is built and can be easily generated into a document. This glossary is a living artifact of the development process, not only because it is a materiality itself, but also because it is inside and part of a structured model. Hence, by learning and improving the way developers work, it is possible to avoid the introduction of usability defects into the product.

*Analytical QA.* The identified relationships in the usability model can also be used for analytical QA. With our quality model we aim to break down the properties and attributes to a level where we can measure them and, therefore, are easily able to give concrete instructions in analytical QA. In particular, we are able to generate guidelines and checklists for reviews from the model. The properties and attributes are there and subsets can easily be selected and exported in different formats so that developers and reviewers always have the appropriate guidelines at hand. Moreover, we annotate the attributed properties in the model, whether they are automatically, semi-automatically, or only manually assessable. Hence, we can identify quality aspects that can be analyzed automatically straightforwardly. Thus, we are able to use all potential benefits of automation.

*Analyses and Predictions.* Finally, more general analysis and predictions are possible based on the quality model. One reason to organize the properties and activities in a tree structure is to be able to aggregate analysis to higher levels. This is important to get concise information about the quality of the system. To be able to do this, the impacts of properties on activities must be quantified. For example, the usability model is a suitable basis for cost/benefit analysis because the identified relationships can be quantified and set into relation to costs similar to the model in [24]. In summary, we are able to aid analytical QA in several ways by utilizing the knowledge coded into the model.

## 7 Conclusion

Usability is a key criterion in the quality of software systems, especially for its user.

It can be decisive for its success on the market. However, the notion of usability and its measurement and analysis are still not fully understood. Although there have been interesting advances by consolidated models, e.g. [5], these models suffer from various shortcomings, such as inconsistencies in the dimensions used. An approach based on an explicit meta-model has proven to be useful for the quality attribute *maintainability*. Hence, we propose a comprehensive usability model that is based on the same meta-model.

Using the meta-model and constructing such a usability model allows us to describe completely the usability of a system by its facts and their relationship with (or impact on) the activities of the user. We support the consistent and unambiguous compilation of the usability knowledge available. The general model still needs to be refined for specific contexts that cannot be included in a general model. By utilizing a specific usability model, we have several benefits, such as the ability to generate guidelines and glossaries or to derive analyses and predictions.

The usefulness of this approach is demonstrated by a case study in which an ISO standard is modeled and several omissions are identified. For example, the standard contains three sub-principles which describe activities, but no impacts on them, as well as nine requirements that have no described impacts. This hampers the justification of the guideline: A rule that is not explicitly justified will not be followed.

For future work we plan to improve further the general usability model and to carry out more case studies in order to validate further the findings of our current research. Furthermore, other quality attributes, e.g. *reliability*, will also be modeled by means of the meta-model to investigate whether this approach works for all attributes. If this be the case, the different models can be combined, since they are all based on a common meta-model.

## References

1. Bevan, N.: International standards for HCI and usability. *Int. J. Hum.-Comput. Stud.* 55, 533–552 (2001)
2. Seffah, A., Metzker, E.: The obstacles and myths of usability and software engineering. *Commun. ACM* 47(12), 71–76 (2004)
3. Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., Macleod, G.J., Merrit, M.J.: *Characteristics of Software Quality*. North-Holland, Amsterdam (1978)
4. Dromey, R.G.: A model for software product quality. *IEEE Trans. Software Eng.* 21(2), 146–162 (1995)
5. Seffah, A., Donyaee, M., Kline, R.B., Padda, H.K.: Usability measurement and metrics: A consolidated model. *Software Quality Control* 14(2), 159–178 (2006)
6. Broy, M., Deissenboeck, F., Pizka, M.: Demystifying maintainability. In: *Proc. 4th Workshop on Software Quality (WoSQ 2006)*. ACM Press, New York (2006)
7. ISO 15005: Road vehicles – Ergonomic aspects of transport information and control systems – Dialogue management principles and compliance procedures (2002)
8. Cavano, J.P., McCall, J.A.: A framework for the measurement of software quality. In: *Proc. Software quality assurance workshop on functional and performance issues*, pp. 133–139 (1978)
9. Shackel, B., Richardson, S. (eds.): *Human Factors for Informatics Usability*. Cambridge University Press, Cambridge (1991)
10. Nielsen, J.: *Usability Engineering*. AP Professional (1993)
11. ISO 9126-1: Software engineering – Product quality – Part 1: Quality model (2001)
12. ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability (1998)
13. Shneiderman, B.: *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 3rd edn. Addison-Wesley, Reading (1998)
14. Grudin, J.: The case against user interface consistency. *Commun. ACM* 32(10), 1164–1173 (1989)
15. Dix, A., Finley, J., Abowd, G., Beale, R.: *Human-Computer Interaction*, 2nd edn. Prentice-Hall, Englewood Cliffs (1998)
16. ISO 9241-110: Ergonomics of human-system interaction – Part 110: Dialogue principles (2006)
17. Folmer, E., Bosch, J.: Architecting for usability: A survey. *The Journal of Systems and Software* 70, 61–78 (2004)

18. van Welie, M., van der Veer, G.C., Eliëns, A.: Breaking down usability. In: Proc. International Conference on Human-Computer Interaction (INTERACT 1999), pp. 613–620. IOS Press, Amsterdam (1999)
19. Norman, D.A.: Cognitive engineering. In: Norman, D.A., Draper, S.W. (eds.) User Centered System Design: New Perspectives on Human-Computer Interaction, pp. 31–61. Lawrence Erlbaum Associates, Mahwah (1986)
20. Andre, T.S., Hartson, H.R., Belz, S.M., McCreary, F.A.: The user action framework: A reliable foundation for usability engineering support tools. *Int. J. Hum.-Comput. Stud.* 54(1), 107–136 (2001)
21. ISO 9126-4: Software engineering – Product quality – Part 4: Quality in use metrics (2004)
22. Frøkjær, E., Hertzum, M., Hornbæk, K.: Measuring usability: Are effectiveness, efficiency, and satisfaction really correlated? In: Proc. Conference on Human Factors in Computing Systems (CHI 2000), pp. 345–352. ACM Press, New York (2000)
23. Sutcliffe, A.: *User-Centered Requirements Engineering: Theory and Practice*. Springer, Heidelberg (2002)
24. Wagner, S.: A model and sensitivity analysis of the quality economics of defect-detection techniques. In: Proc. International Symposium on Software Testing and Analysis (ISSTA 2006), pp. 73–83. ACM Press, New York (2006)

## Questions

### **Laurence Nigay:**

*Question: You describe the product using two models but there are a lot of usability models, why only the two? Task models can be used to describe properties such as reachability.*

Answer: Factors and activity can capture all this information in these models and then relate it to activities.

### **Michael Harrison:**

*Question: Much is said at the moment about the need to consider the features of complex systems that cannot be characterized by a decompositional approach – so-called emergent properties. So for example a high reliability organization is one for reasons that cannot easily be understood using the probing style techniques that you have described. What is your opinion of this perspective and do you agree that there is a need to explore alternatives to the style of analysis that you describe?*

Answer: This technique is better than other techniques that exist and none of them handle these emergent properties of complex systems.

### **Thomas Memmel:**

*Question: If you say you are building a model-based system to understand design would you say that simulation is not also a good idea?*

Answer: Of course both are required. I have described just one aid for the developer.



# Suitability of Software Engineering Models for the Production of Usable Software

Karsten Nebe<sup>1</sup> and Dirk Zimmermann<sup>2</sup>

<sup>1</sup> University of Paderborn, C-LAB, Fürstenallee 11, 33098 Paderborn, Germany

<sup>2</sup> T-Mobile Deutschland GmbH, Landgrabenweg 151,  
53227 Bonn, Germany

Karsten.Nebe@c-lab.de, Dirk.Zimmermann@t-mobile.de

**Abstract.** Software Engineering (SE) and Usability Engineering (UE) both provide a wide range of elaborated process models to create software solutions. Today, many companies have understood that a systematic and structured approach to usability is as important as the process of software development itself. However, theory and practice is still scarce how to incorporate UE methods into development processes. With respect to the quality of software solutions, usability needs to be an integral aspect of software development and therefore the integration of these two processes is a logical and needed step. One challenge is to identify integration points between the two disciplines that allow a close collaboration, with acceptable additional organizational and operational efforts. This paper addresses the questions of where these integration points between SE and UE exist, what kind of fundamental UE activities have to be integrated in existing SE processes, and how this integration can be accomplished.

**Keywords:** Software Engineering, Usability Engineering, Standards, Models, Processes, Integration.

## 1 Introduction

Software engineering is a discipline that adopts various engineering approaches to address all phases of software production, from the early stages of system specification up to the maintenance phase after the release of the system ([14],[17]). Software engineering tries to provide a systematic and planable approach for software development. To achieve this, it provides comprehensive, systematic and manageable procedures, in terms of software engineering process models (SE Models).

SE Models usually define detailed activities, the sequence in which these activities have to be performed and the resulting deliverables. The goal in using SE Models is a controlled, solid and repeatable process in which the project achievement do not depend on individual efforts of particular people or fortunate circumstances [5]. Hence, SE Models partially map to process properties and process elements, adding concrete procedures.

Existing SE Models vary with regards to specific properties (such as type and number of iterations, level of detail in the description or definition of procedures or activities, etc.) and each model has specific advantages and disadvantages, concerning

predictability, risk management, coverage of complexity, generation of fast deliverables and outcomes, etc.

Examples of such SE Models are the Linear Sequential Model (also called Classic Life Cycle Model or Waterfall Model) [15], Evolutionary Software Development [12], the Spiral Model by Boehm [1], or the V-Model [9].

### 1.1 Linear Sequential Model

The Linear Sequential Model divides the process of software development into several successive phases: *System Requirements, Software Requirements, Analysis, Program Design, Coding, Testing and Operations*. On the transition from one phase to the other it is assumed that the previous phase has been completed. Iterations between neighboring phases are planned to react on problems or errors which are based on the results of the previous phase. The Linear Sequential Model is document-driven. Thus, the results of each phase are documents that serve as milestones to track the development progress.

### 1.2 Evolutionary Development

In the Evolutionary Development the phases *Software Specification, Development and Validation* are closely integrated. Evolutionary Development is especially well suited for software projects where the requirements cannot be defined beforehand or in which the requirements are likely to change during the development process. The procedure is always a sequence of iterative development-cycles which results in an improved version of a product on the end of each sequence. There is no explicit maintenance phase at the end of the lifecycle. Necessary changes after the product delivery are solved in further iterations. Within Evolutionary Development the end users and the customers are closely involved in the development process. The goal of Evolutionary Development is “to avoid a single-pass sequential, document-driven, gated-step approach“ [10].

### 1.3 Spiral Model

The Spiral Model is a refinement of the Linear Sequential Model in which the single phases are spirally run through. This cycle in the spiral is repeated four times, for *System Definition, Software Requirements, Conception (Architecture Design) and Realisation (Detail Conception, Coding, Test, Integration and Installation)*. The nature of the model is risk-driven. At the end of each cycle the current project progress is being analyzed and the risk of project failure is evaluated. Depending on the evaluation outcome the project goals are (re)defined and resources are (re)allocated or – in the worst case - the development is being discontinued if necessary for the subsequent phases. Unlike the Linear Sequential Model, risks are identified throughout the process which leads to a more control- and planable process. The failure of a project can be significantly minimized.

### 1.4 V-Model

The V-Model represents the development process in a symmetric model in which the validation is performed inversely to the system compilation, starting from module up

to the acceptance test [13]. The V-Model is based upon the Linear Sequential Model but emphasis is laid on the assurance of quality (e.g. connections between basic concepts and the resulting products). Inspections take place at multiple test phases testing different levels of detail of the solution and not only at the end of development as other models propose. Compared to the Linear Sequential Model or the Spiral Model, the V-Model is more precise in its description of procedures and measures.

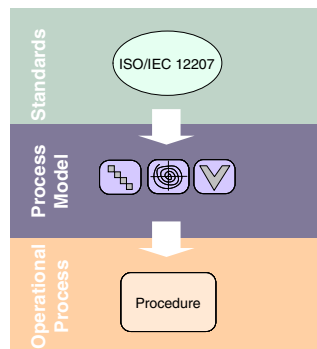
## 1.5 Standards in Software Engineering

Software engineering standards define a framework for SE Models on a higher abstraction level. They define rules and guidelines as well as properties of process elements as recommendations for the development of software. Thereby, standards support consistency, compatibility and exchangeability, and cover the improvement of quality and communication.

The ISO/IEC 12207 provides such a general process framework for the development and management of software. “The framework covers the life cycle of software from the conceptualization of ideas through retirement and consists of processes for acquiring and supplying software products and services.” [7]. It defines processes, activities and tasks and provides descriptions about how to perform these items on an abstract level.

In order to fulfill the superordinate conditions of software engineering standards (and the associated claim of ensuring quality) the SE Models should comply with these conditions. In general, standards as well as SE Models can not be directly applied. They are adapted and/or tailored according to the corresponding organizational conditions. The resulting instantiation of a SE Model, fitted to the organizational aspects, is called software development process, which can then be used and put to practice. Thus, the resulting Operational Process is an instance of the underlying SE Model and the implementation of activities within the organization.

This creates a hierarchy of different levels of abstractions for software engineering: Standards that define the overarching framework, process models that describe systematic and traceable approaches and the operational level in which the models are tailored to fit the specifics of an organization (Figure 1).



**Fig. 1.** Hierarchy of standards, process models and operational processes in software engineering

## 1.6 Usability Engineering

Usability Engineering is a discipline that is concerned with the question of how to design software that is easy to use (usable). Usability engineering is “an approach to the development of software and systems which involves user participation from the outset and guarantees the efficacy of the product through the use of a usability specification and metrics.” [4]

Usability engineering provides a wide range of methods and systematic approaches for the support of development. These approaches are called Usability Engineering Models (UE Models) or Usability Lifecycles, such as the Goal-Directed-Design [2], the Usability Engineering Lifecycle [11] or the User-Centered Design-Process Model of IBM [6]. All of them have much in common since they describe an idealized approach that ensures the development of usable software, but they differ their specifics, in the applied methods and the general description of the procedure (e.g. phases, dependencies, goals, responsibilities, etc.) [18]. UE Models usually define activities and their resulting deliverables as well as the order in which specific tasks or activities have to be performed. The goal of UE Models is to provide tools and methods for the implementation of the user’s needs and to guarantee the efficiency, effectiveness and users’ satisfaction of the solution.

Thus, usability engineering and software engineering address different needs in the development of software. Software engineering aims at systematic, controllable and manageable approaches to software development, whereas usability engineering focuses on the realization of usable and user-friendly solutions.

The consequence is that there are different views between the two disciplines during system development, which sometimes can be competing, e.g. SE focuses on system requirements and the implementation of system concepts and designs, whereas UE focuses on the implementation of user requirements and interaction concepts and designs. However, both views need to be considered in particular.

## 1.7 Standards in Usability Engineering

Usability Engineering provides standards similar to the way Software Engineering does. They also serve as a framework to ensure consistency, compatibility, exchangeability, and quality which is in line with the idea of software engineering standards. However, usability engineering standards lay the focus on the users and the construction of usable solutions. Examples for such standards are the DIN EN ISO 13407 [3] and the ISO/PAS 18152 [8].

The DIN EN ISO 13407 introduces a process framework for the human-centered design of interactive systems. Its’ overarching aim is to support the definition and management of human-centered design activities, which share the following characteristics:

- 1) the active involvement of users and a clear understanding of user and task requirements (Context of use)
- 2) an appropriate allocation of function between users and technology (User Requirements)
- 3) the iteration of design solutions (Produce Design Solutions)
- 4) multi-disciplinary design (Evaluation of Use)

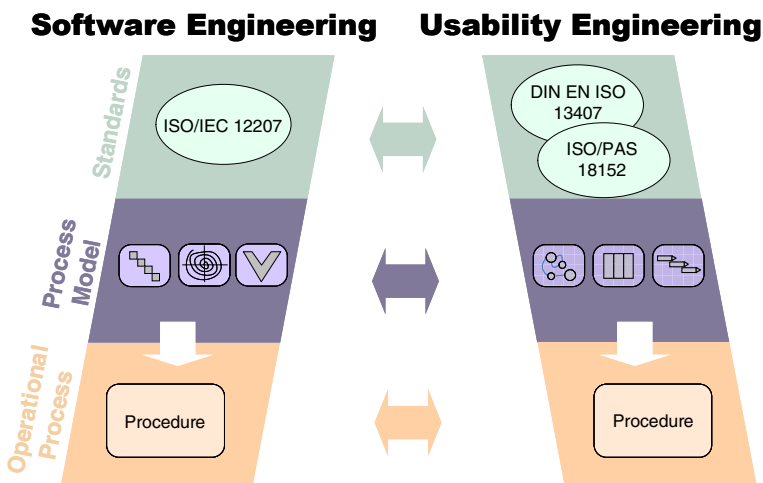
These characteristics are reflected by the activities (named in brackets), which define the process framework of the human centered design process, and have to be performed iteratively.

The ISO/PAS 18152 is partly based on the DIN EN ISO 13407, and describes a reference model to measure the maturity of an organization in performing processes that make usable, healthy and safe systems. It describes processes and activities that address human-system issues and the outcomes of these processes. It provides details on the tasks and artifacts associated with the outcomes of each process and activity.

There is a sub-process called *Human-centered design* which describes the activities that are commonly associated with a User Centered Design Process. These activities are *Context of use*, *User requirements*, *Produce design solutions* and *Evaluation of use*, which are in line with the DIN EN ISO 13407. However, by being more specific in terms of defining lists of activities (so called Base Practices), that describe how the purpose of each activity is achieved (e.g. what needs to be done to gather the user requirements in the right way). The ISO/PAS 18152 enhances the DIN EN ISO 13407 in terms of the level of detail and contains more precise guidelines.

In order to ensure the claims of the overarching standards, UE Models need to adhere to the demands of the corresponding framework. Thus, a connection between the standards and the UE Models exists which is similar to the one the authors described for software engineering. There is a hierarchy of standards and subsequent process models, too.

Additionally there are similarities on the level of operational processes. The selected UE Model needs to be adjusted to the organizational guidelines. Therefore, a similar hierarchy of the different abstraction levels exists for software engineering and for usability engineering (Figure 2). Standards define the overarching framework, models describe systematic and traceable approaches and on the operational level these models are adjusted and put into practice.



**Fig. 2.** Similar hierarchies in the two disciplines software engineering and usability engineering: standards, process models and operational processes

## 2 Motivation

For development organizations SE Models are an instrument to plan and systematically structure the activities and tasks to be performed during software creation.

Software development organizations aim to fulfill specific goals when they plan to develop a software solution. Such goals could be the rapid development of a new software solution, to become the leader in the application area or to develop a very stable and reliable solution e.g. because to enhance the organization's prestige – and of course, to generate revenue with it. Depending on their goals an organization will choose one (or the combination of multiple ones) SE Model for the implementation that will in their estimate fit best. However, these goals are connected with criteria which can manifest themselves differently. These could be organization-specific characteristics, such as the planability of the process or project, quality of the process, size/volume of the project, organizational structures, types of qualification, etc. These could also be product-specific characteristics, like security and reliability, verification and validation, innovation, etc.

Thus depending on the goals of an organization the decision of selecting an appropriate SE Model for the implementation is influenced by the underlying criteria. As an example, the Linear Sequential Model with its' predefined results at the end of each phase and its sequential flow of work certainly provides a good basis for a criterion such as planability. On the other hand, the Evolutionary Development might not be a good choice if the main focus of the solution is put on error-robustness because the continuous assembling of the solution is known to cause problems in structure and the maintenance of software code.

As usability engineering put the focus on the user and usability of products, which is an important aspect of quality, usability is important for the development process. Usability could take up both either product-specific characteristics (such as the efficiency, effectiveness and satisfaction of using a product) or organization-specific characteristics (like legal restrictions or company guidelines such as producing usable products to distinguish on the market). Thus, usability is also an important – even crucial – criterion for organizations to choose a well-suited SE Model.

However, one problem remains – usability engineering activities are not an inherent part of software engineering, respectively of SE Models. Indeed, many different models for software engineering and usability engineering exist but there is a lack of systematic and structured integration [16]. They often coexist as two separate processes in an organization and therefore need to be managed separately and in addition need to be synchronized. However, as usability is an important quality aspect it needs to be an integral part of software engineering and of SE Models. It seems reasonable to extend the more extensive proceeding with the missing parts, which in this case means to add usability engineering activities to the software engineering process models, to integrate these two disciplines.

Beside the need for integration it is, however, important to consider both views, the systematic, controllable and manageable approaches of SE and the realization of usable and user-friendly solutions of UE, respectively. It should not be tried to cover one view with the other. The goal is to guarantee an efficient coexistence but to retain the specific goals and approaches of each discipline.

According to the hierarchy of standards, process models and operational processes an integration of the disciplines has to be performed on each level. This means that for the level of standards needs to be proven that aspects of software engineering and usability engineering can coexist and can be integrated. On the level of process models it has to be ensured that usability engineering aspects can be incorporated with SE Models. And on the operational level activities a close collaboration needs to be achieved, resulting in acceptable additional organizational and operational efforts.

### 3 Proceedings

In order to identify the integration points between software engineering and usability engineering, the authors examined the three different levels, based on the hierarchies of standards, process models and operational processes (Figure 2):

1. On the abstract overarching level of Standards in software engineering and usability engineering, serving as a framework to ensure consistency, compatibility, exchangeability, and quality within and beyond the organizational borders and to cover the improvement of quality and communication.
2. On the level of Process Models for software engineering and usability engineering, to provide a procedural model and more refined approach that can serve as a framework for an organization, providing specific advantages and disadvantages, like predictability, risk management, coverage of complexity, generation of fast deliverables and outcomes, etc.
3. On the Operational Process level which reflects the execution of activities and the processing of information within the organization. It is an instance of the underlying model and the implementation of activities and information processing within the organization.

The goal of analysis on the level of standards is to identify similarities in the description of standards between SE and UE. They could be found in definitions of activities, tasks, goals, procedures or deliverables. With the focus on activities the authors will create a framework of activities, representing SE and UE likewise. Such a framework can be used to set limits for the following analysis, on the level of process models.

Based on the framework different SE Models are being analyzed in terms of how they already support the implementation of activities from a usability point of view. Criteria are being defined to measure the significance of UE activities within the SE Models. Based on the results and identified gaps recommendations for the enhancements of SE Models are being derived. These enable the implementation of activities on the level of models to ensure the development of user friendly solutions.

On the operational level the analysis is used to examine whether the recommendation meet the requirements of the practice. Measures regarding a specific SE Model in practice are being derived, evaluated and analyzed. As a result statements about the efficiency of the measures in making a contribution to the user-centeredness of the operational process could be made.

In this paper the authors will show the proceedings and first results of the analysis on the level of standards and of the level of process models. The derivation of recommendations, the refinement of the analysis methods and the analysis on the operational level are currently in progress and will be published by future work.

### 3.1 Analysis of Standards

To figure out whether software engineering and usability engineering have similarities on the level of standards, the standards' detailed descriptions of processes, activities and tasks, output artifacts, etc. have been analyzed and compared. For this the software engineering standard ISO/IEC 12207 was chosen to be compared with the usability engineering standard DIN EN ISO 13407.

The ISO/IEC 12207 defines the process of software development as a set of 11 activities: *Requirements Elicitation, System Requirements Analysis, Software Requirements Analysis, System Architecture Design, Software Design, Software Construction, Software Integration, Software Testing, System Integration, System Testing* and *Software Installation*. It also defines specific development tasks and details on the generated output to provide guidance for the implementation of the process.

The DIN EN ISO 13407 defines four activities of human-centered design that should take place during system development. These activities are the *Context of use, User Requirements, Produce Design Solutions* and *Evaluation of Use*. The DIN EN ISO 13407 also describes in detail the kind of output to be generated and how to achieve it.

On a high level, when examining the descriptions of each activity, by relating tasks and outputs with each other, similarities were found in terms of the characteristics, objectives and proceedings of activities. Based on these similarities single activities were consolidated as groups of activities (so called, Common Activities). These common activities are part of both disciplines software engineering and usability engineering on the high level of standards. An example of such a common activity is the Requirement Analysis. From a software engineering point of view (represented by the ISO/IEC 12207) the underlying activity is the *Requirement Elicitation*. From the usability engineering standpoint, specifically the DIN EN ISO 13407, the underlying activities are the Context of Use and User Requirements, which are grouped together. Another example is the Software Specification, which is represented by the two software engineering activities *System Requirements Analysis* and *Software Requirements Analysis*, as well as by Produce Design Solutions from a usability engineering perspective.

The result is a compilation of five common activities: Requirement Analysis, Software Specification, Software Design and Implementation, Software Validation, Evaluation that represent the process of development from both, a software engineering and a usability engineering point of view (Table 1).

These initial similarities between the two disciplines lead to the assumption of existing integration points on this overarching level of standards. Based on this, the authors used these five common activities as a general framework for the next level in the hierarchy, the level of process models.

However, the identification of these similar activities does not mean that one activity is performed in equal measure in SE and UE practice. They have same goals on the abstract level of standards but they differ in the execution at least on the operational level. Thus, Requirement Analysis in SE focuses mainly on system based requirements whereas UE requirements describe the users' needs and workflows. The activity of gathering requirements is equal but the view on the results is different. Another example is the Evaluation. SE evaluation aims at correctness and correctness of code whereas UE focuses on the completeness of users' workflows and the fulfillment of users' needs.



**Table 1.** Comparison of software engineering and usability engineering activities on the level of standards and the identified similarities (Common Activities)

<b>ISO/IEC 12207 Sub- Process: Development</b>	<b>Common Activities</b>	<b>DIN EN ISO 13407</b>
Requirements Elicitation	<b>Requirement Analysis</b>	Context of Use User Requirements
System Requirements Analysis Software Requirements Analysis	<b>Software Specification</b>	Produce Design Solutions
System Architecture Design Software Design Software Construction Software Integration	<b>Software Design and Implementation</b>	n/a
Software Testing System Integration	<b>Software Validation</b>	Evaluation of Use
System Testing Software Installation	<b>Evaluation</b>	Evaluation of Use

Consequently it is important to consider these different facets of SE and UE likewise. And as usability has become an important quality aspect in software engineering, the identified common activities have not only to be incorporated in SE Models from a software engineering point of view, but also from the usability engineering point of view. Some SE models might already adhere to this but obviously not all of them. To identify whether usability engineering aspects of the common activities are already implemented in SE Models (or not), the authors performed a gap-analysis with selected SE Models. The overall goal of this was to identify integration points on the level of process models.

Therefore, the authors first needed a deep understanding about the selected SE Models and second, needed an accurate specification of the requirements that put demands on the SE Models from the usability engineering perspective, on which the SE Models then could be evaluated.

### 3.2 Analyzed SE Models

For the analysis of SE Models four commonly used models were selected: the Linear Sequential Model, the Evolutionary Development, the Spiral Model and the V-Model. They were examined and classified, in particular regards to their structural characteristics (e.g. classification of activities, proceedings, etc.), their specifics (e.g. abilities, disabilities, etc.) and their individual strengths and weaknesses.

The descriptions of the SE Models in literature served as the basis of the analysis. Improvements or extensions based on expert knowledge or practical experiences were not taken into account to retain the generality of statements. A sample of the results is represented in the following table (Table 2).

The gap-analysis surfaced particular characteristics of the considered models. Based on the identified strengths and weaknesses first indicators were derived that are in the authors eyes crucial for the model selection on the operational level. For example, the Evolutionary Development could be a good choice if the organization wants

to get results fast because of its ability to produce solution design successively and its ability to deal with unspecific requirements. A disadvantage of Evolutionary design is however, that due to the continuous changes and adjustments, the software quality and structure can suffer. However for the development of safety-relevant products,

**Table 2.** Strength/Weaknesses-Profiles of software engineering models

	<b>Basic properties</b>	<b>Specifics</b>	<b>Strength</b>	<b>Weakness</b>
<b>Linear Sequential Model</b>	<ul style="list-style-type: none"> <li>- division of the development process into sequent phases</li> <li>- completeness of previous phase requirement for the next phase</li> <li>- successive development</li> <li>- iterations between contiguous phases</li> <li>- deliverables define project's improvement</li> </ul>	<ul style="list-style-type: none"> <li>- document-driven</li> <li>- phase-oriented</li> </ul>	<ul style="list-style-type: none"> <li>- controllable management</li> <li>- controlling the complexity by using encapsulation</li> </ul>	<ul style="list-style-type: none"> <li>- lack of assistance with imprecise or incorrect product definitions</li> <li>- problems with supplementary error identification and experiences from development</li> </ul>
<b>Evolutionary Development</b>	<ul style="list-style-type: none"> <li>- intertwined specification, development and evaluation phases</li> <li>- no distinct phases</li> <li>- successive requirement processing (and elicitation, if applicable)</li> <li>- sequence of development cycles</li> <li>- version increment at the end of every cycle</li> <li>- no explicit but implicit maintenance phase</li> <li>- high customer and user involvement</li> </ul>	<ul style="list-style-type: none"> <li>- successive solution design</li> <li>- ability to deal with unspecific requirements</li> <li>- avoids single-pass sequential, document-driven, gated-step approaches</li> </ul>	<ul style="list-style-type: none"> <li>- compilation of "quick solutions"</li> <li>- ability to react to changing requirements</li> <li>- small changes lead to measurable improvements</li> <li>- user-oriented</li> <li>- early identification of problems and shortcomings</li> </ul>	<ul style="list-style-type: none"> <li>- problems in software quality and structure caused by continuous changes and adoptions</li> <li>- maintainability</li> <li>- maintenance and quality of the documentation</li> <li>- difficulties in measuring the project progress</li> <li>- precondition is a flexible system</li> </ul>
<b>Spiral Model</b>	<ul style="list-style-type: none"> <li>- enhancement of the phase model</li> <li>- phases are distributed in a spiral-shaped form</li> <li>- development within four cycles</li> <li>- evaluation, decision making, goal definition &amp; planning of resources at end of each cycle</li> </ul>	<ul style="list-style-type: none"> <li>- successive solution design</li> <li>- risk-driven</li> </ul>	<ul style="list-style-type: none"> <li>- risk management</li> <li>- simultaneous control of budget and deliverables</li> <li>- independent planning and budgeting of the single spiral cycles</li> <li>- flexible, pure risk oriented but controlled response to current status</li> </ul>	<ul style="list-style-type: none"> <li>- high effort on management and planning</li> </ul>
<b>V-Model</b>	<ul style="list-style-type: none"> <li>- based on the Linear Sequential Model</li> <li>- enhancement regarding quality assurance</li> <li>- symmetric process</li> <li>- evaluation reverse to system development</li> <li>- evaluation on different levels of detail</li> </ul>	<ul style="list-style-type: none"> <li>- continuous evaluation</li> <li>- quality assurance</li> </ul>	<ul style="list-style-type: none"> <li>- measures for continuous evaluation of the quality assurance</li> <li>- verification and evaluation on all levels of detail</li> </ul>	<ul style="list-style-type: none"> <li>- initial planning efforts</li> <li>- basically for large projects</li> </ul>

which need to adhere to a detailed specification, the Linear Sequential Model could be a good choice because of its stepwise and disciplined process. For developing new, complex and expensive software solutions the Spiral Model could be the method of choice because of its risk-oriented and successive development approach.

The results of the SE Model analysis are Strength/Weakness-Profiles that guide the selection of a specific SE Model based on organization specific criteria. Afterwards, with the detailed knowledge about the selected SE Models the maturity of these models in creating usable products was examined.

### 3.3 Gap-Analysis of SE Models

To assess the ability of SE Models to create usable products, requirements need to be defined first that contain the usability engineering demands and that can be used for evaluation later on.

As mentioned above the DIN EN ISO 13407 defines a process framework with the four activities Context of Use, User Requirements, Produce Design Solutions and Evaluation of Use. The reference model of the ISO/PAS 18152 represents an extension to parts of the DIN EN ISO 13407. Particularly the module *Human-centered design* of the ISO/PAS 18152 defines base practices for the four activities of the framework. These base practices describe in detail how the purpose of each activity is achieved. Thus, it is an extension on the operational process level. Since the ISO/PAS 18152 is aimed for processes assessments, its base practices describe the optimal steps. Therefore they can be used as usability engineering requirements that need to be applied by the SE Models to ensure to create usable products. According to this, there is an amount of requirements where each activity can be evaluated against. The following Table (Table 3) shows the base practices of the activity User Requirements.

**Table 3.** Base practices of the module HS.3.2 *User Requirements* given in the ISO/PAS 18152

HS.3.2 User Requirements	
BP1	Set and agree the expected behaviour and performance of the system with respect to the user.
BP2	Develop an explicit statement of the user requirements for the system.
BP3	Analyse the user requirements.
BP4	Generate and agree on measurable criteria for the system in its intended context of use.
BP5	Present these requirements to project stakeholders for use in the development and operation of the system.

Based on these requirements (base practices) the authors evaluated the selected SE Models. The comparison was based on the description of the SE Models. For each requirement the authors determined whether the model complied to it or not. The results for each model and the regarding requirements are displayed in Table 4. The quantity of fulfilled requirements for each activity of the framework informs about the level of compliance of the SE Model satisfying the usability engineering requirements. According to the results statements about the ability of SE Models to create usable products were made. Table 5 shows the condensed result of the gap-analysis.

**Table 4.** Results of the gap-analysis: Coverage of the base practices for the Linear Sequential Model (LSM), Evolutionary Development (ED), Spiral Model (SM) and V-Model (VM)

Modul	Activity	LSM	ED	SM	VM
<b>HS 3.1</b>	<b>Context of use</b>				
1	Define the scope of the context of use for the system.	-	-	+	+
2	Analyse the tasks and worksystem.	-	-	-	+
3	Describe the characteristics of the users.	-	-	-	+
4	Describe the cultural environment/organizational/management regime.	-	-	-	+
5	Describe the characteristics of any equipment external to the system and the working environment.	-	-	-	+
6	Describe the location, workplace equipment and ambient conditions.	-	-	-	+
7	Analyse the implications of the context of use.	-	-	-	+
8	Present these issues to project stakeholders for use in the development or operation of the system.	-	+	-	-
<b>HS 3.2</b>	<b>User Requirements</b>				
1	Set and agree the expected behaviour and performance of the system with respect to the user.	-	-	+	+
2	Develop an explicit statement of the user requirements for the system.	-	+	+	+
3	Analyse the user requirements.	-	+	+	+
4	Generate and agree on measurable criteria for the system in its intended context of use.	-	-	+	+
5	Present these requirements to project stakeholders for use in the development and operation of the system.	-	-	-	-
<b>HS 3.3</b>	<b>Produce design solutions</b>				
1	Distribute functions between the human, machine and organizational elements of the system best able to fulfil each function.	-	-	-	-
2	Develop a practical model of the user's work from the requirements, context of use, allocation of function and design constraints for the system.	-	-	-	-
3	Produce designs for the user-related elements of the system that take account of the user requirements, context of use and HF data.	-	-	-	-
4	Produce a description of how the system will be used.	-	+	+	+
5	Revise design and safety features using feedback from evaluations.	-	+	+	+
<b>HS 3.4</b>	<b>Evaluation of use</b>				
1	Plan the evaluation.	-	+	+	+
2	Identify and analyse the conditions under which a system is to be tested or otherwise evaluated.	-	-	+	+
3	Check that the system is fit for evaluation.	+	+	+	+
4	Carry out and analyse the evaluation according to the evaluation plan.	+	+	+	+
5	Understand and act on the results of the evaluation.	+	+	+	+

The compilation of findings shows, that for none of the SE Models all Base Practices of ISO/PAS 18152 can be seen as fulfilled. However, there is also a large variability in the coverage rate between the SE Models. For example, the V-Model shows

a very good coverage for all modules except for smaller fulfillment of HS 3.3 Produce Design Solution criteria, whereas the Linear Sequential Model only fulfills a few of the HS 3.4 Evaluation of use criteria and none of the other modules.

Evolutionary Design and the Spiral Model share a similar pattern of findings, where they show only little coverage for Context of Use, medium to good coverage of User Requirements, limited coverage for Produce Design Solution and good support for Evaluation of Use activities.

**Table 5.** Results of the gap-analysis, showing the level of sufficiency of SE Models covering the requirements of usability engineering

	Context of Use	User Requirements	Produce Design Solutions	Evaluation of Use	Across Activities
<b>Linear Sequential Model</b>	0 %	0 %	0 %	60 %	13 %
<b>Evolutionary Development</b>	13 %	40 %	40 %	80 %	39 %
<b>Spiral Model</b>	13 %	80 %	40 %	100 %	52 %
<b>V-Modell</b>	88 %	80 %	40 %	100 %	78 %
<b>Across Models</b>	28 %	50 %	30 %	85 %	

By looking at the summary of results (Table 5) and comparing the percentage of fulfilled requirements for each SE Model, it shows that the V-Model performs better than the other models and can be regarded as basically being able to produce usable products. With a percentage of 78% it is far ahead of the remaining three SE Models. In the comparison, the Linear Sequential Model cuts short by only 13%, followed by Evolutionary Development (39%) and the Spiral Model (52%).

If one takes both the average values of fulfilled requirements and the specific base practices for each usability engineering activity into account, it shows that the emphasis for all SE Models is laid on evaluation (Evaluation of Use), especially comparing the remaining activities. The lowest overall coverage could be found in the Context of Use and Produce Design Solution, indicating that three of the four SE models don't consider the relevant contextual factors of system usage sufficiently, and also don't include (user focused) concept and prototype work to an extent that can be deemed appropriate from a UCD perspective.

### 3.4 Interpretation and Results

Based on the relatively small compliance values for the Context of Use (28%), User Requirements (50%) and Produce Design Solutions (30%) activities across all SE

models, the authors see this as an indicator that there is only a loose integration between usability engineering and software engineering. There are less overlaps between the disciplines regarding these activities and therefore it is necessary to provide suitable interfaces to create a foundation for the integration.

The results of the gap-analysis can be used to extend the Strength/Weakness-Profiles in a way that these can be supplemented by statements about the ability of the SE Models to produce usable products. Thus, the quality criterion usability becomes an additional aspect of the profiles and for the selection of appropriate SE Models.

The presented approach does not only highlight weaknesses of SE Models regarding the usability engineering requirements and corresponding activities, it also pinpoints the potential for integration between software engineering and usability engineering:

- Where requirements are not considered as fulfilled, recommendations could be derived, which would contribute to an accomplishment.
- The underlying base practices and their detailed descriptions provide appropriate indices what needs to be considered in detail on the level of process models.

To give some examples, first high-level recommendations e.g. for the Linear Sequential Model could be made as followed: Besides phases like *System Requirements* and *Software Requirements* there needs to be a separate phase for gathering user requirements and analysis of the context of use. As the model is document driven and completed documents are post-conditions for the next phase it has to be ensured that usability results are part of this documentation. The evaluation is a downstream phase that is performed after completing of the solution (or at least of a complex part of the solution). User centered validation aspects should take place already as early as possible, e.g. during the *Preliminary Design*. For the Spiral Model user validations should be introduced as an explicit step at the end of each cycle in order to avoid the risk of developing a non-usable solution.

According to the given approach and the results it shows that any SE Model can be similarly analyzed and mapped with the requirements of usability engineering to be then adapted or extended according to the recommendations based on the gap-analysis results in order to ensure the creation of usable products. This can be used a foundation for implementing the operational process level and will guarantee the interplay of software engineering and usability engineering in practice.

## 4 Summary and Outlook

The approach presented in this paper was used to identify integration points between software engineering and usability engineering on three different levels of abstractions. The authors showed that standards define an overarching framework for both disciplines. Process models describe systematic and planable approaches for the implementation and the operational process in which the process models are tailored to fit the specifics of an organization.

On the first level of standards the authors analyzed, compared and contrasted the software engineering standard ISO/IEC 12207 with the usability engineering standard

DIN EN ISO 13407 and identified common activities as part of both disciplines. They define the overarching framework for the next level of process models.

Based on this, the authors analyzed different software engineering process models. These models were classified, in particular regarding their structural characteristics (e.g. classification of activities, proceedings, etc.), their specifics (e.g. abilities, disabilities, etc.) and their individual strengths and weaknesses. As a result, the authors derived Strength/Weaknesses-Profiles for each model that helps organizations to select the appropriate process model for the implementation.

In order to identify the maturity of these software engineering process models' ability to create usable products, the authors synthesized demands of usability engineering and performed an assessment of the models. The results provide an overview about the degree of compliance of the models with usability engineering demands. It turned out that there is a relatively small compliance to the usability engineering activities across all software engineering models. This is an indicator that there only little integration between usability engineering and software engineering exists. There are less overlaps between the disciplines regarding these activities and therefore it is necessary to provide suitable interfaces to create a foundation for the integration.

But, the presented approach does not only highlight weaknesses of software engineering process models, it additionally identifies opportunities for the integration between software engineering and usability engineering. These can be used a foundation to implement the operational process level and will help to guarantee the interplay of software engineering and usability engineering in practice, which is part of the authors' future work.

However, the analysis results and regarding statements about the software engineering models are currently only based on their documented knowledge in literature. The authors are aware of the fact that there are several adoptions of the fundamental/basic models in theory and practice. Hence, in future research the authors will include more software engineering process models, even agile development models, to provide more guidance in selecting the most suitable model and to give more precise and appropriate criteria for selection.

The demands of usability engineering used in this paper are based on the base practices of the ISO/PAS 18152, which was a valid basis for a first analysis of the selected software engineering models. It is expected that there is a need for a different procedure in analyzing agile models because they are not as document and phase driven as classical software engineering models and the ISO/PAS 18152 are. The authors will apply the given approach to evaluate whether agile process models are better able to suit the demands of usability engineering than formalized approaches compared in this paper.

Regarding the current procedure the authors discovered that more detailed/adequate criteria for the assessment are necessary by which objective and reliable statements about process models and their ability to create usable software could be made. Therefore the authors plan to conduct expert interviews as a follow-up task to elicit appropriate criteria for the evaluation of SE models. Based on these criteria the authors will perform another gap-analysis of selected software engineering models (including agile approaches). The authors expect to derive specific recommendations to enrich the SE Models by adding or adapting usability engineering activities, phases, artifacts, etc. By doing this, the development of usable software on the level of proc-

ess models will be guaranteed. Furthermore, hypothesizes about the process improvements are expected to be made for each recommendation which then can be evaluated on the Operational Process level. Therefore, case studies will be identified based on which the recommendations could be transferred in concrete measures. These measures will then be evaluated by field-testing to verify their efficiency of user-centeredness of software engineering activities. This will help to derive concrete measures that result in better integration of software engineering and usability engineering in practice and hopefully more usable products.

## References

1. Boehm, B.: A Spiral Model of Software Development and Enhancement. *IEEE Computer* 21, 61–72 (1988)
2. Cooper, A., Reimann, R.: *About Face 2.0*. Wiley, Indianapolis (2003)
3. DIN EN ISO 13407. Human-centered design processes for interactive systems. CEN - European Committee for Standardization, Brussels (1999)
4. Faulkner, X.: *Usability Engineering*, pp. 10–12. PALGARVE, New York (2000)
5. Glinz, M.: Eine geführte Tour durch die Landschaft der Software-Prozesse und - Prozessverbesserung. *Informatik – Informatique*, 7–15 (6/1999)
6. IBM: *Ease of Use Model* (November 2004), [http://www-3.ibm.com/ibm/easy/eou\\_ext.nsf/publish/1996](http://www-3.ibm.com/ibm/easy/eou_ext.nsf/publish/1996)
7. ISO/IEC 12207. Information technology - Software life cycle processes. Amendment 1, 2002-05-01. ISO copyright office, Switzerland (2002)
8. ISO/PAS 18152. Ergonomics of human-system interaction — Specification for the process assessment of human-system issues. First Edition 2003-10-01. ISO copyright office, Switzerland (2003)
9. KBST: *V-Modell 97* (May 2006), <http://www.kbst.bund.de>
10. Larman, C., Basili, V.R.: Iterative and Incremental Development: A Brief History. *Computer* 36(6), 47–56 (2003)
11. Mayhew, D.J.: *The Usability Engineering Lifecycle*. Morgan Kaufmann, San Francisco (1999)
12. McCracken, D.D., Jackson, M.A.: Life-Cycle Concept Considered Harm-ful. *ACM Software Engineering Notes*, 29–32 (April 1982)
13. Pagel, B., Six, H.: *Software Engineering: Die Phasen der Softwareentwicklung*, 1st edn., vol. 1. Addison-Wesley Publishing Company, Bonn (1994)
14. Patel, D., Wang, Y. (eds.): *Annals of Software Engineering*. In: Editors' introduction: Comparative software engineering: Review and perspectives, vol. 10, pp. 1–10. Springer, Netherlands (2000)
15. Royce, W.W.: Managing the Delopment of Large Software Systems. *Proceedings IEEE*, 328–338 (1970)
16. Seffah, A. (ed.): *Human-Centered Software Engineering – Integrating Usability in the Development Process*, pp. 3–14. Springer, Dordrecht (2005)
17. Sommerville, I.: *Software Engineering*, 7th edn. Pearson Education Limited, Essex (2004)
18. Woletz, N.: *Evaluation eines User-Centred Design-Prozessassessments - Empirische Untersuchung der Qualität und Gebrauchstauglichkeit im praktischen Einsatz*. Doctoral Thesis. University of Paderborn, Paderborn, Germany (April 2006)



## Questions

**Jan Gulliksen:**

*Question: One thing I miss is one of the key benefits of the ISO standards, namely the key values provided by the principles. Your analysis of process steps fails to address these four values. The software engineering process covers much more.*

Answer: The goal is to be more specific in describing the assessment criteria and therefore it does not address the principles. We plan to develop more specific criteria through interviews and use these criteria to assess the process models including software engineering models in more detail. Then we will go back to the companies to see how they fit.

**Ann Blandford:**

*Question: Work is analytical looking at how things should be – what confidence do you have about how these things can work in practice? Methods are always subverted and changed in practice anyway.*

Answer: Documentation is not representative enough – plan to do more specific work with experts in the field. SE experts could answer, for example, whether the criteria for usability engineering fit into an underlying model. We will then map these to criteria in order apply them in practice.

# A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces

Jean-Sébastien Sottet, Gaëlle Calvary, Joëlle Coutaz, and Jean-Marie Favre

Université Joseph Fourier, 385 rue de la Bibliothèque, BP 53,  
38041 Grenoble Cedex 9, France

{Jean-Sébastien.Sottet, Gaëlle.Calvary, Joëlle.Coutaz,  
Jean-Marie.Favre}@imag.fr

**Abstract.** Plastic User Interfaces (UI) are able to adapt to their context of use while preserving usability. Research efforts have focused so far, on the functional aspect of UI adaptation, while neglecting the usability dimension. This paper investigates how the notion of mapping as promoted by Model Driven Engineering (MDE), can be exploited to control UI adaptation according to explicit usability criteria. In our approach, a run-time UI is a graph of models related by mappings. Each model (e.g., the task model, the Abstract UI, the Concrete UI, and the final UI) describes the UI from a specific perspective from high-level design decisions (conveyed by the task model) to low-level executable code (i.e. the final UI). A mapping between source and target models specifies the usability properties that are preserved when transforming source models into target models. This article presents a meta-model for the notion of mapping and shows how it is applied to plastic UIs.

**Keywords:** Adaptation, Context of use, Mapping, Meta-model, Model, Model transformation, Plasticity, Usability.

## 1 Introduction

In Human-Computer Interaction (HCI), plasticity refers to the ability of User Interfaces (UI) to withstand variations of context of use while preserving usability [36]. Context of use refers to a set of observables that characterize the conditions in which a particular system is running. It covers three information spaces: the user model, the platform model, and the physical and social environment model. UI adaptation has been addressed using many approaches over the years, including Machine Learning [21], Model-Driven Engineering (MDE) [8,17,18,32,33], and Component-oriented services [30]. Regardless of the approach, the tendency has been to focus on the functional aspects of adaptation. Usability has generally been regarded as a natural by-product of whatever approach was being used. In this article, we propose to promote usability as a first class entity using a model-based approach.

This article is structured in the following way. Section 2 introduces the concepts of MDE followed in Section 3, by the instantiation of the MDE principles when applied to the problem of UI plasticity. Section 4 presents HHCS (Home Heating Control System), a simple case study used as a running example to illustrate the principles.

The rest of the paper is dedicated to the notion of mappings. First, in Section 5, we show how different UIs can be produced for HHCS using different mappings. Then we switch to a more abstract discussion with the definition of a meta-model for mappings (Section 6).

## 2 Motivations for an MDE Approach

Although promising, the model-based approach to the development of UIs has not met wide acceptance: developers have to learn a new specification language, the connection between the specification and the resulting code is hard to understand and control, and the kinds of UI's that can be built are constrained by the underlying conventional toolkit [19]. However, this early work has established the foundations for transforming high-level specifications into executable code. In particular, the following steps now serve as references for designing and developing UIs: from the domain-dependent Concepts and Task models, an Abstract UI (AUI) is derived which in turn is transformed into a Concrete UI (CUI), followed by the Final UI (Figure 1) [36].

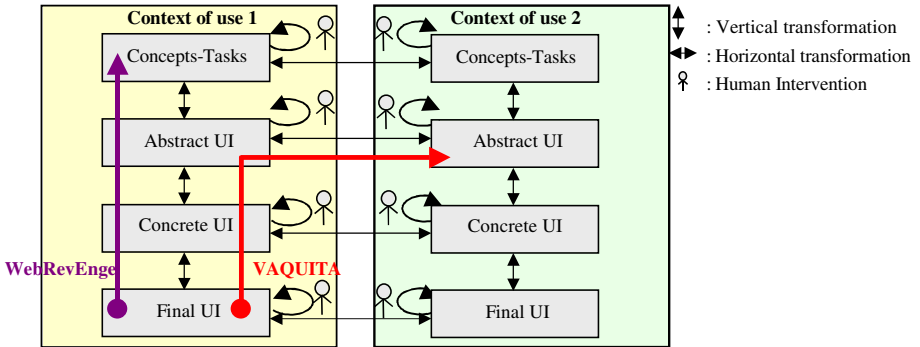


Fig. 1. A model-based framework [7] for UI plasticity

As discussed in [7], transformations can be combined and applied to any of these models to support UI adaptation. For example, VAQUITA [5] and WebRevEng [23] reverse engineer HTML source files into more abstract descriptions (respectively AUI and task levels), and from there, depending on the tool, either retarget and generate the UI or are combined with retargeting and/or forward engineering tools (Figure 1). This means that developers can produce the models they are familiar with – including source code for fine-tuned elegant UIs, and then use the tools that support the appropriate transformations to retarget the UI to a different context of use. Transformations and models are at the heart of MDE.

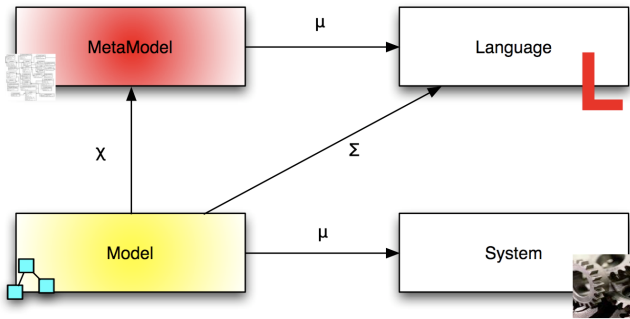
The motivation for MDE is the integration of very different know-how and software techniques. Over the years, the field of software engineering has evolved into the development of many paradigms and application domains leading to the emergence of multiple Technological Spaces (TS). "A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities" [14]. Examples of technological spaces include documentware concerned with digital

documents using XML as the fundamental language to express specific solutions, dataware related to data base systems, ontologyware, etc. In HCI, a java-based control panel running on a PDA can be used to control a web-based application running on a PC. Today, technological spaces can no longer evolve in autarky. Most of them share challenges of increasing complexity, such as adaptation, to which they can only offer partial solutions. Thus, we are in a situation where concepts, approaches, skills, and solutions, need to be combined to address common problems. MDE aims at achieving integration by defining gateways between technological spaces. The hypothesis is that models, meta-models, model transformations, and mappings, offer the appropriate means.

*A model is a representation of a thing (e.g., a system), with a specific purpose.* It is “able to answer specific questions in place of the actual thing under study” [4]. Thus, a model, built to address one specific aspect of a problem, is by definition a simplification of the actual thing. For example, a task model is a simplified representation of some human activities (the actual thing under study), but it provides answers about how “representative users” proceed to reach specific goals. Things and models are *systems*. Model is a *role of representation* that a system plays for another one. Models form oriented graphs ( $\mu$  graphs) whose edges denote the  $\mu$  relation “is represented by” (Figure 2). Models may be contemplative (they cannot be processed automatically by computers) or productive (they can be processed by computers). Typically, scenarios developed in HCI [27] are contemplative models of human experience in a specified setting. On the other hand, the task model exploited in TERESA [3] is productive.

In order to be processed (by humans, and/or by computers), a model must comply with some shared syntactic and semantic conventions: it must be a well-formed expression of a language. This is true both for productive and contemplative models: most contemplative models developed in HCI use a mix of drawings and natural language. A TERESA [3] task model is compliant with CTT [25]. A language is the set of all well-formed expressions that comply with a grammar (along with a semantics). In turn, a grammar is a model from which one can produce well-formed expressions (or models). Because a grammar is a model of a set of models ( $\epsilon$  relation “is part of” on Figure 2), it is called a meta-model. CTT [25] is a meta-model for expressing specific task models.

*A meta-model is a model of a set of models that comply with it.* It sets the rules for producing models. It does not represent models. Models and meta-models form a  $\chi$  tree: a model *complies* to a single meta-model, whereas a meta-model may have multiple compliant models. In the same way, a *meta-meta-model is a model of a set of meta-models that are compliant with it.* It does not represent meta-models, but sets the rules for producing distinct meta-models. The OMG Model-Driven Architecture (MDA) initiative has introduced a four-layer modeling stack as a way to express the integration of a large diversity of standards using MOF (Meta Object Facility) as the unique meta-meta-model. This top level is called M3, giving rise to meta-models, models and instances (respectively called M2, M1 and M0 levels). MDA is a specific MDE deployment effort around industrial standards including MOF, UML, CWM, QVT, etc. The  $\mu$  and  $\chi$  relations, however, do not tell how models are produced within a technological space, nor how they relate to each other across distinct technological spaces. The notions of *transformation* and *mapping* is the MDE answer to these issues.



**Fig. 2.** Basic concepts and relations in MDE

In the context of MDE, a transformation is the production of a set of target models from a set of source models, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how source models are transformed into target models [16]. Source and target models are related by the  $\tau$  relation “is transformed into”. Note that a set of transformation rules is a model (a transformation model) that complies with a transformation meta-model.  $\tau$  expresses an overall dependency between source and target models. However, experience shows that finer grain of correspondence needs to be expressed. Typically, the incremental modification of one source element should be propagated easily into the corresponding target element(s) and vice versa. The need for traceability between source and target models is expressed as *mappings* between source and target elements of these models. For example, each task of a task model and the concepts involved to achieve the task, are rendered as a set of interactors in the CUI model. Rendering is a transformation where tasks and their concepts are mapped into workspaces which, in turn, are mapped into windows populated with widgets in case of graphical UIs. The correspondence between the source task (and concepts) and its target workspace, window and widgets, is maintained as mappings. Mappings will be illustrated in Section 5 for the purpose of UI plasticity and meta-modeled in Section 6.

Transformations can be characterized within a four-dimension space: The transformation may be *automated* (it can be performed by a computer autonomously), it may be *semi-automated* (requiring some human intervention), or it may be *manually performed* by a human. A transformation is *vertical* when the source and target models reside at different levels of abstraction (Figure 1). Traditional UI generation is a vertical top down transformation from high-level descriptions (such as a task model) to code generation. Reverse engineering is also a vertical transformation, but it proceeds bottom up, typically from executable code to some high-level representation by the way of abstraction. A transformation is *horizontal* when the source and target models reside at the same level of abstraction (Figure 1). For example, translating a Java source code into C code preserves the original level of abstraction. Transformations are *endogenous* when the source and target models are expressed in the same language (i.e., are compliant to the same meta-model). Transformations are *exogenous* when sources and targets are expressed in different languages while belonging to the same technological space. When crossing technological spaces (e.g., transforming a Java source code into a

JavaML document), then additional tools (called exporters and importers) are needed to bridge the gap between the spaces. *Inter-technological transformations* are key to knowledge and technical integration.

As discussed next, our approach to the problem of plastic UI is to fully exploit the MDE theoretic framework opening the way to the explicit expression of usability to drive the adaptation process.

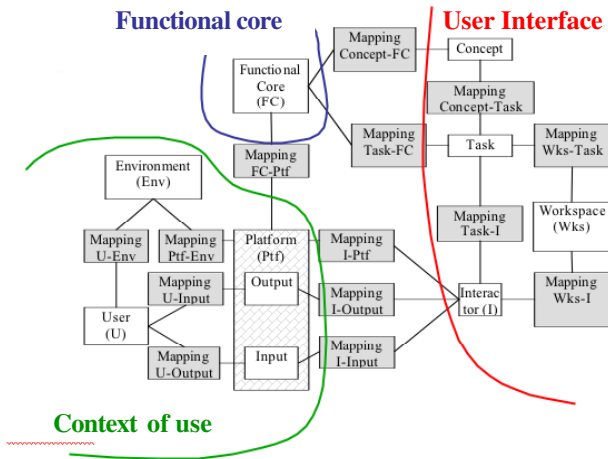
### 3 MDE for UI Plasticity

Early work in the automatic generation of UIs [32] as well as more recent work in UI adaptation adhere only partially to the MDE principles. Our approach differs from previous work [8,17,18,32] according to the following four principles.

**Principle#1:** *An interactive system is a graph of MI-level models.* This graph expresses and maintains multiple perspectives on the system both at design-time and run-time (Fig. 3). As opposed to previous work, an interactive system is not limited to a set of linked pieces of code. The models developed at design-time, which convey high-level design decision, are still available at run-time. A UI may include a task model, a concept model, a workspace (i.e. an AUI) model, and an interactor (i.e. a CUI) model linked by mappings. In turn, the UI components are mapped to items of the Functional Core of the interactive system, whereas the CUI elements (the interactors) are mapped to input and output (I/O) devices of the platform. Mappings between interactors and I/O devices support the explicit expression of centralized versus distributed UIs. The whole graph (Fig. 3) forms an ecosystem: a set of entities that interact to form an organized and self-regulated unit until some threshold is reached. When the threshold is reached, Principle #3 comes into play.

**Principle #2:** *Transformations and mappings are models.* In the conventional model-driven approach to UI generation, transformation rules are diluted within the tool. Consequently, “the connection between specification and final result can be quite difficult to control and to understand” [19]. In our approach, transformations are promoted as models. As any model, they can be modified both at design-time and run-time at different degrees of automation. The same holds for mappings. In particular, mappings are decorated with properties to convey usability requirements. As motivated in Section 6, *the usability framework used for mappings is left opened.* This aspect will be discussed in detail in Sections 5 and 6.

**Principle #3:** *Design-time tools are run-time services.* The idea of creating UIs by dynamically linking software components was first proposed in the mid-eighties for the Andrew Toolkit [24], followed by OpenDoc, Active X, and Java Beans. However, these technical solutions suffer from three limitations: they are code centric, the assembly of components is specified by the programmer, and the components are supposed to belong to the same technological space. In our approach, any piece of code is “encapsulated” as a service. Some of them implement portions of the UI. We call them *UI services*. Others, the *UI transformers*, interpret the models that constitute the interactive system. In other words, the model interpreters used at design-time are also services at run-time. As a result, if no UI service can be found to satisfy a new context of use, a new one can be produced on the fly by UI transformers. In particular,



**Fig. 3.** A UI is a graph of models. Mappings define both the rationale of each UI element and the UI deployment on the functional core and the context of use.

the availability of a task model at run-time makes it possible to perform deep UI adaptation based on high-level abstractions.

**Principle #4: Humans are kept in the loop.** HCI design methods produce a large body of contemplative models such as scenarios, drawings, storyboards, and mock-ups. These models are useful reference material during the design process. On the other hand, because they are contemplative, they can only be transformed manually into productive models. Manual transformation supports creative inspiration, but is prone to wrong interpretation and to loss of key information. On the other hand, experience shows that automatic generation is limited to very conventional UIs. To address this problem, we accept to support a mix of *automated*, *semi-automated*, and *manually* performed transformations. For example, given our current level of knowledge, the transformation of a “value-centered model” [9] into a “usability model” such as that of [2], can only be performed manually by designers. Semi-automation allows designers (or end-users) to adjust the target models that result from transformations. For example, a designer may decide to map a subset of an AUI with UI services developed with the latest post-WIMP toolkit. The only constraint is that the hand-coded executable piece is modeled according to an explicit meta-model and is encapsulated as a service. This service can then be dynamically retrieved and linked to the models of the interactive system by the way of mappings. With productive models at multiple levels of abstraction, the system can reason at run-time about its own design. In a nutshell, the components of a particular system at run-time can be a mix of generated and hand-coded highly tuned pieces of UI. By the way of a meta-UI [11], end-users can dynamically inform the adaptation process of their preferences.

To summarize, our approach to the problem of UI plasticity brings together MDE (Model Driven Engineering) and SOA (Service Oriented Approach) within a unified framework that covers both the development stage and the run-time phase of interactive systems. In this paper, we investigate how usability can be described and controlled by the way of mappings given that an interactive system is a graph of

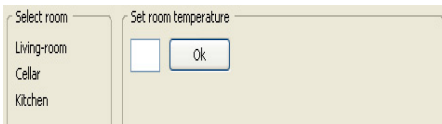
models. We use HHCS as an illustrative example before going into a more formal definition of the notion of mapping and its relation with that of transformation.

### 4 The Home Heating Control System: Overall Description

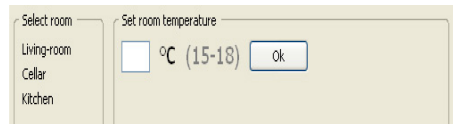
Our Home Heating Control System (HHCS) makes it possible for users to control the temperature of their home using different devices. Examples include a dedicated wall-mounted display, a Web browser running on a PDA, or a Java-enabled watch. As shown in Fig. 4, many UI variants are made possible, depending on the device screen size, as well as on the set of usability properties that HCI designers have elicited as key:

- From a functional perspective, the four UI’s of Fig. 4 are equivalent: they support the same set of tasks, with the same set of rooms (the living room, the cellar and the kitchen) whose temperature may be set between 15°C and 18°C;
- From a non-functional perspective, these UI’s do not satisfy the same set of usability properties. In particular, according to C. Bastien and D. Scapin’s usability framework [2], *prompting* (a factor for *guidance*), *prevention against errors* (a factor for *error management*), and *minimal actions* (a factor for *workload*) are not equally supported by the four UI solutions. In Fig. 4-a), the unit of measure (i.e. Celsius versus Fahrenheit) is not displayed. The same holds for the room temperature whose range of values is not made observable. As a result, prompting is not fully supported. In Fig. 4-b), the lack of prompting is repaired but the user is still not prevented from entering wrong values. Solutions in Fig. 4-c) and Fig. 4-d) satisfy the prompting criteria as well as prevention against error. Moreover, Fig. 4-d) improves the *minimal actions* recommendation (a factor for *workload*) by eliminating the “Select room” navigation (also called articulatory) task. The UIs of Fig. 4-a to 4-c satisfy *homogeneity-consistency* because the same type of interactor (i.e. a web link) is used to choose a room.

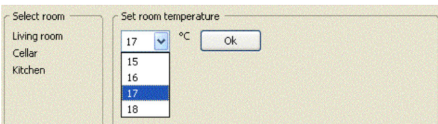
(a) The unit of measure and the valid temperature values are not observable



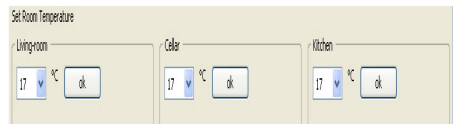
(b) The unit of measure and the valid temperature values are both observable



(c) The user is prevented from making errors

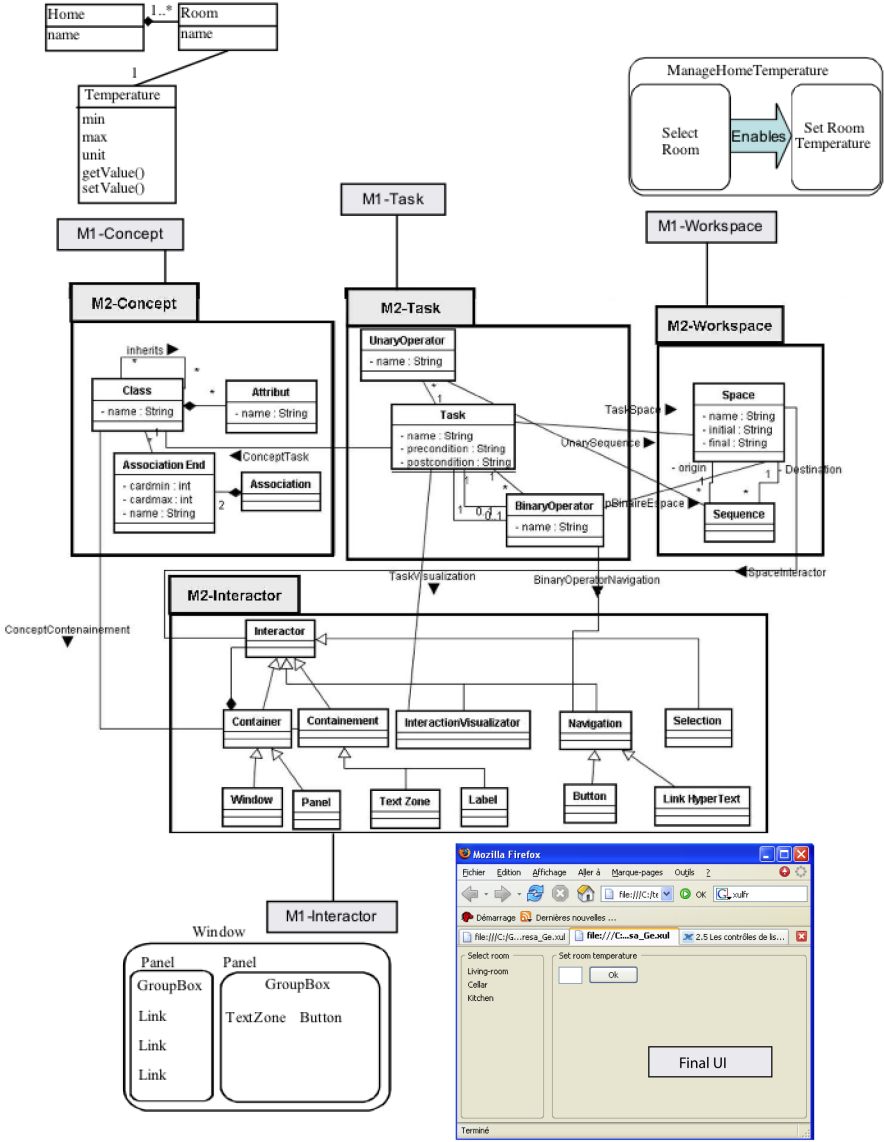


(d) The user is prevented from navigation tasks



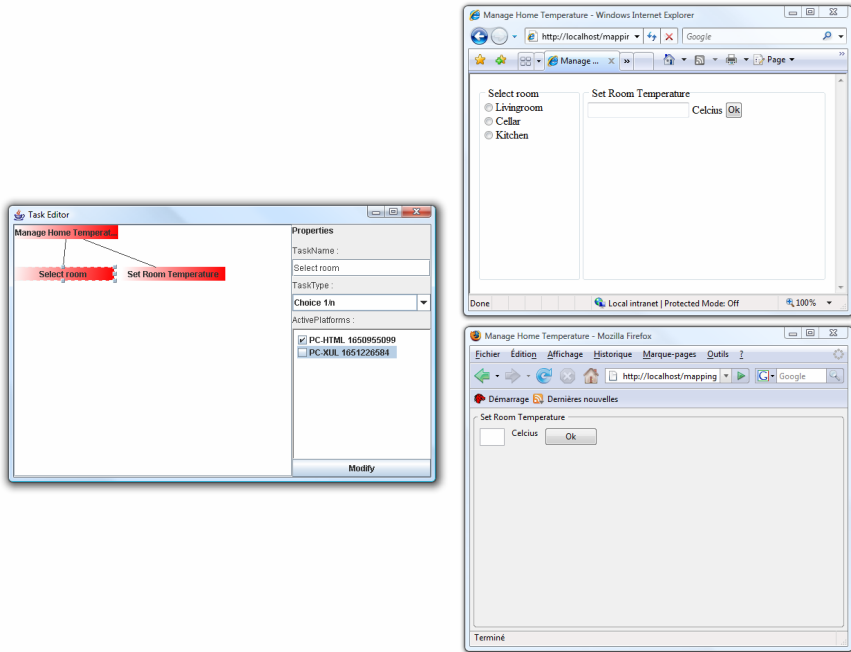
**Fig. 4.** Four functionally-equivalent UIs that differ from the set of usability criteria used to produce them





**Fig. 5.** A subset of the graph of M1-models for HHCS. Each model is compliant to a meta-model (M2-level). Each M1-level model of this figure is related to another M1-level model by the way of some mapping to form a sub-graph of Fig.1.

The purpose of this paper is not to define new meta-models but to show how mappings are appropriate for conveying usability properties. Whatever the UI is (Fig.4-a, b, c or d), HHCS is a graph of models, each of them depicting a specific perspective. Each model (M1-level) is compliant to a meta-model (M2-level). Fig. 5



**Fig. 6.** An early meta-UI making it possible for the user to redistribute the UI by changing the mappings between tasks and platforms

shows a subset of the HHCS graph of models corresponding to Fig. 4a. The deployment on the functional core and the context of use is not depicted. Here, we use UML as meta-meta-model (M3-level model).

- The task meta-model (M2) defines a task as a goal that can be reached by the execution of a set of subtasks related by binary operators (e.g., enabling). A task may be decorated with unary operators (e.g. optional, iterative). Managing temperature at home is a goal that can iteratively be achieved by first selecting a room and then specifying the desired temperature (M1-level). The relations between the tasks and the domain concepts (e.g., select a room) are mappings that make explicit the roles that the concepts play in the tasks (input and/or output, centrality, etc.).
- A domain concept is a concept that is relevant to users to accomplish tasks in a particular domain (e.g., home, room, temperature). Concepts are classes that are linked together by the way of associations (e.g., home is made of a set of rooms).
- A workspace is an abstract structuring unit that supports a set of logically connected tasks. To support a task, a workspace is linked to the set of domain concepts involved within that task. A workspace may recursively be decomposed into workspaces whose relations (should) express the semantics

of tasks operators (e.g., *gives access to* for the *enabling* operator). In Fig. 4a-b-c, there are three workspaces: one per task.

- An interactor is the basic construct for CUIs (e.g., window, panel, group box, link, text field, button). It is a computational abstraction that allows the rendering and manipulation of entities that require interaction resources (e.g., input/output devices). Each interactor is aware of the task and domain concepts it represents, and the workspace in which it takes place.

Fig. 6 shows a basic meta-UI that allows the user (either the designer and/or the end-user) to observe and manipulate a sub-graph of the M1-level models of HHCS. In this early prototype, the meta-UI is limited to the task and platform models. By selecting a task of the task model, then selecting the platform(s) onto which the user would like to execute the task, the user can dynamically redefine the redistribution of the UI over the resources currently available. The UI is re-computed and redistributed on the fly, thus ensuring UI consistency. On Fig. 6, two platforms are available (a PC HTML and a PC XUL-enabled). End-users can map the tasks “Select room” and “Set room temperature” respectively, to the PDA-HTML platform and to the PC-XUL platform, resulting in the Final UI shown in Fig.6.

This toy meta-UI shows only the mappings. The properties that these mappings convey are neither observable nor controllable. This is the next implementation step for fully demonstrating the conceptual advances that we present next. Section 5 is about the mappings used in HHCS whereas Section 6 goes one step further with the definition of a meta-model for mappings.

## 5 Mappings in HHCS

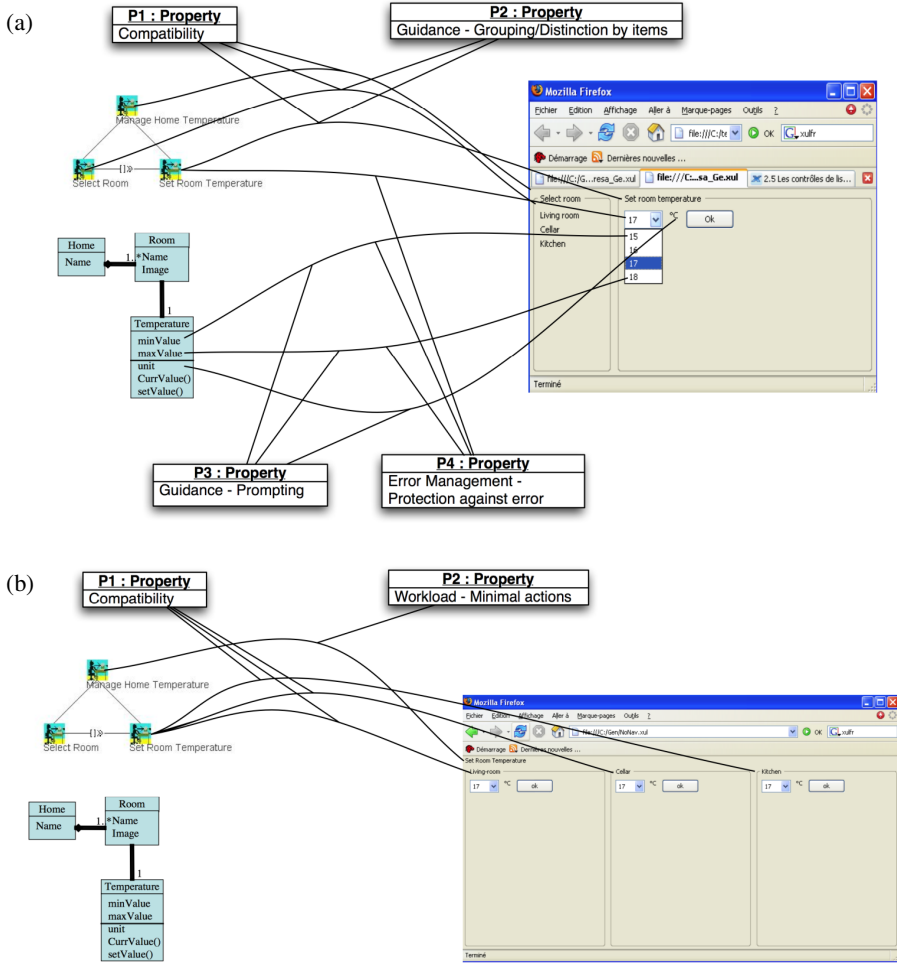
In HHCS, we have used Bastien-Scapin’s recommendations as our usability framework<sup>1</sup>. Due to lack of space, we limit our analysis to four of the eight criteria of this framework:

- *Task compatibility*;
- *Guidance* in terms of *Prompting* and *Grouping/Distinction of items*;
- *Error Management* in terms of *Error prevention*;
- *Workload* in terms of *Minimal actions*.

In model-driven UI generation, usability criteria motivate the way abstract models are vertically transformed into more concrete models. Typically, *Grouping/Distinction of items* motivates the decomposition of UIs in terms of *workspaces* so that the concepts manipulated within a task are *grouped* together. By doing so, the *distinction* between the tasks that users can accomplish, is made salient. In our approach, we use usability criteria not only to motivate a particular design, but also to support plasticity at run-time. A mapping between elements of source and target models, is specified either manually in a semi-formal way by the designer, or is created automatically by the system as the result of a transformation function. The choice of the appropriate transformation function is performed, either by the system, or specified by users (the

---

<sup>1</sup> As discussed in Section 6, other frameworks are valid as well.



**Fig. 7.** Examples of mappings in HHCS resulting in different UIs depending on usability properties

designer or end-users if conveniently presented in a well-thought meta-UI). Fig. 7 shows the mappings defined for HHCS between the task model, the concept model and the CUI. These mappings are generated by the system, but the choice of the transformation functions is specified by the designer. In the current implementation, transformations are expressed in ATL. They are executed by an ATL interpreter encapsulated as an OSGi service.

Fig. 7-a corresponds to the UI shown in Fig.4-c. Here, four properties have been elicited as key: *Compatibility* (property P1), *Grouping/Distinction of items* (property P2), *Prompting* (property P3) and *Protection against error* (property P4). P1 and P2

are attached to the mappings that result from the transformation function between tasks and workspaces. As shown in Fig. 7-a, this transformation function has generated one workspace per task (a workspace for selecting a room, and a workspace to set room temperature). These workspaces are spatially close to each other (they correspond to tasks that share the same parent task), and each workspace makes observable the concepts manipulated by the corresponding task. As a result, the CUI fully supports user's tasks and is well-structured. Property P3 (*Prompting*) and Property P4 (*Protection against errors*) influences the way concepts and tasks are represented in terms of interactors. Because of Property P3, the unit of measure as well as the min and max values for a room temperature are made observable. Because of Property P4, the possible values for a room temperature are rendered as a pull-down menu.

Fig. 7-b) shows a very different CUI for the same set of tasks and concepts, but using a different set of properties. In particular, the *Minimal actions* Property aims at eliminating navigation tasks. As a result, because the screen real estate is sufficient, there is one workspace per room, and the task "Select a room" is performed implicitly by setting the temperature directly in the appropriate workspace.

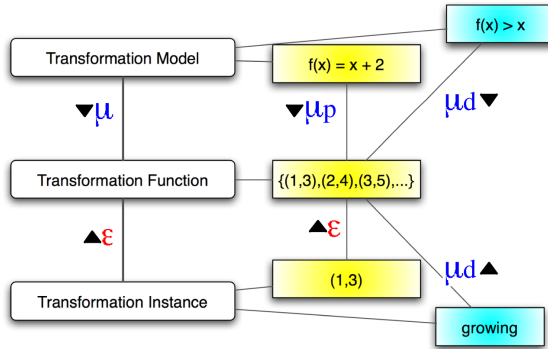
Next section presents our meta-model for mappings. This meta-model is general, applicable to HCI for reasoning on usability-driven transformations.

## 6 Formal Definition of Mapping

In mathematics, a mapping is "a rule of correspondence established between two sets that associates each member of the first set with a single member of the second" [The American Heritage Dictionary of the English Language, 1970, p. 797]. In MDE, the term "mapping" is related to the notion of "transformation function", but the overall picture is far from being clear. First, we clarify the notion of transformation as exploited in MDE. Then, we use this notion to propose a meta-model for mappings.

Fig. 8 introduces three terms: *transformation model*, *transformation function* and *transformation instance*. They are illustrated on the mathematical domain. " $f(x)=x+2$ " is a *transformation model* that is compliant to a mathematical meta-model. A transformation model describes ( $\mu$  relation) a *transformation function* in a predictive way: here the set  $\{(1,3),(2,4),(3,5)\dots\}$  for the function "f" when applied to integers. A transformation function is the set of all the *transformation instances* inside the domain variation (here, the integers). *Transformation instances* are subsets ( $\epsilon$  relation) of the *transformation function*. They are the execution trace of the function (here, "f").

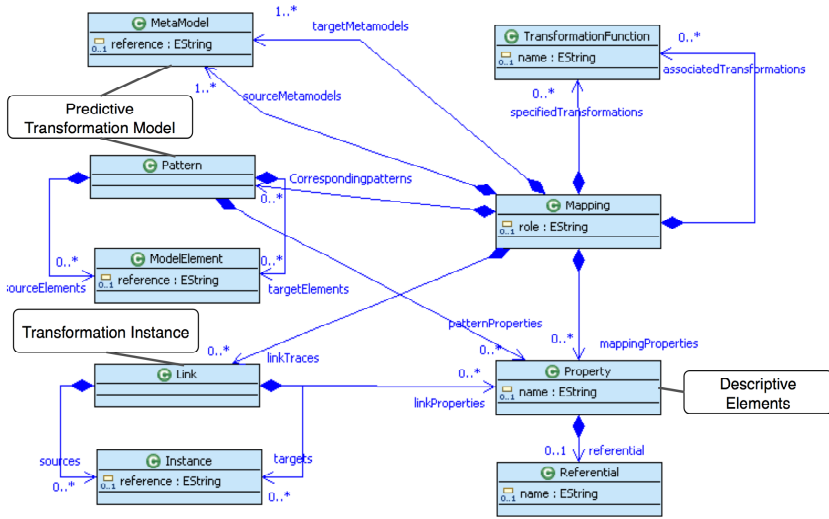
In Fig. 8, the  $\mu$  relation is refined into  $\mu_p$  and  $\mu_d$ . These relations respectively stand for *predictive* and *descriptive* representations. *Predictive* means that there is no ambiguity: the transformation model (e.g., " $f(x)=x+2$ ") fully specifies the transformation function. *Descriptive* refers to a qualifier (e.g., "growing"). It does not specify the transformation function, but provides additional information. In Fig. 8, two examples are provided: "growing" and " $f(x)>x$ ". They respectively deal with transformation instances and model. In the first case, the description is made a posteriori whilst it is made a priori in the second one. A posteriori descriptions are subject to incompleteness and/or errors due to too few samples.



**Fig. 8.** Clarification of the notions of transformation model, transformation function and transformation instance

Transformations are key for specifying mappings. The mapping meta-model provided in Fig. 9 is a general purpose mapping meta-model. The core entity is the *Mapping* class. A mapping links together entities that are compliant to *Meta-models* (e.g., Task and Interactor). A mapping may explicit the corresponding *Transformation functions*. The transformation model can be done by patterns (e.g., to the task pattern *Select a room*, apply the pattern: one hypertext link per room, the name of the link being the name of the room). A *Pattern* is a transformation model that links together source and target elements (*ModelElement*) to provide a predictive description of the transformation function. Patterns are powerful for ensuring the UI's *homogeneity-consistency*. In addition, a mapping may describe the execution trace of the transformation function. The trace is a set of *Links* between *Instances* of *ModelElements* (e.g., the hypertext link Kitchen and the task *Select a room* when applied to the concept of *kitchen*).

A mapping conveys a set of *Properties* (e.g., “Guidance-Prompting”). A property is described according to a given reference framework (*Referential*) (e.g., Bastien&Scapin [2]). Because moving to an unfamiliar set of tools would impose a high threshold on HCI and software designers, we promote an open approach that consists in choosing the appropriate usability framework, then generating and evaluating UIs according to this framework. General frameworks are available such as Shackel [29], Abowd et al., [1], Dix et al. [12], Nielsen [20], Preece [26], IFIP Properties [13], Schneiderman [31], Constantine and Lockwood [10], Van Welie et al. [39], as well as Seffah et al. [28] who propose QUIM, a unifying roadmap to reconcile existing frameworks. More specific frameworks are proposed for web engineering (Montero et al. [17]), or for specific domains (for instance, military applications). Closely related to UI plasticity, Lopez-Jacquero et al.’s propose a refinement of Bastien and Scapin’s framework, as a usability guide for UI adaptation [15]. Whatever the framework is, the properties are descriptive. They qualify either the global set of mappings or one specific element: a mapping, a pattern or a link.



**Fig. 9.** A mapping meta-model for general purpose. The composition between Mapping and Meta-model is due to the Eclipse Modeling Framework.

*Associated transformations* (see the UML association between the classes *Mapping* and *TransformationFunction* in Fig. 9) are in charge of maintaining the consistency of the graph of models by propagating modifications that have an impact on other elements. For instance, if replacing an interactor with another one decreases the UI’s *homogeneity-consistency*, then the same substitution should be applied to the other interactors of the same type. This is the job of the associated functions which perform this adaptation locally.

Our mapping meta-model is general. The HCI specificity comes from the nature of both the meta-models (*Metamodel*) and the framework (*Referential*). Currently in HCI, effort is put on meta-modeling (see UsiXML [38] for instance) but the mapping meta-model remains a key issue. Further work is needed to measure the extent to which traditional usability frameworks are still appropriate for reasoning on UI’s plasticity. Should new criteria such as continuity [37] be introduced? Whatever the criteria are, we need metrics to make it possible for the system to self-evaluate when the context of use changes. Next section elaborates on perspectives for both HCI and MDE communities.

## 7 Conclusion and Perspectives

In 2000, B. Myers stated that model-based approaches had not found a wide acceptance in HCI. They were traditionally used for automatic generation and appeared as disappointing because of a too poor quality of the produced UIs. He envisioned a second life for models in HCI empowered by the need of device independence. In our work, we promote the use, the description and the capitalization of elementary transformations that target a specific issue.

A UI is described as a graph of models and mappings both at design time and run-time. At design time, mappings convey properties that help the designer in selecting the most appropriate transformation functions. Either the target element of the mapping is generated according to the transformation function that has been selected, or the link is made by the designer who then describes the mapping using a transformation model. We envision adviser tools for making the designer aware of the properties he/she is satisfying or neglecting.

At run-time, mappings are key for reasoning on usability. However, it is not easy as (1) there is not a unique consensual reference framework; (2) ergonomic criteria may be inconsistent and, as a result, require difficult tradeoffs. Thus, (1) the meta-model will have to be refined according to these frameworks; (2) a meta-UI (i.e., the UI of the adaptation process) may be relevant for negotiating tradeoffs with the end-user.

Beyond HCI, this work provides a general contribution to MDE. It defines a mapping meta-model and clarifies the notions of mapping and transformation. Mappings are more than a simple traceability link. They can be either predictive (transformation specifications) or descriptive (the properties that are conveyed), as a result covering both the automatic generation and the hand-made linking. Moreover mapping models can embed transformation in order to manage models consistency. This is new in MDE as most of the approaches currently focus on direct transformation. Our mapping meta-model will be stored in the international Zoo of meta-models: the ZOOOMM project [40].

**Acknowledgments.** This work has been supported by the network of excellence SIMILAR and the ITEA EMODE project. The authors warmly thank Xavier Alvaro for the implementation of the prototype.

## References

1. Abowd, G.D., Coutaz, J., Nigay, L.: Structuring the Space of Interactive System Properties. In: Larson, J., Unger, C. (eds.) *Engineering for Human-Computer Interaction*, pp. 113–126. Elsevier, Science Publishers B.V. (North-Holland), IFIP (1992)
2. Bastien, J.M.C., Scapin, D.: *Ergonomic Criteria for the Evaluation of Human-Computer*, Technical report INRIA, N°156 (June 1993)
3. Berti, S., Correani, F., Mori, G., Paterno, F., Santoro, C.: TERESA: a transformation-based environment for designing and developing multi-device interfaces. In: *Conference on Human Factors in computing Systems, CHI 2004 extended abstracts on Human factors in computing systems*, Vienna, Austria, pp. 793–794 (2004)
4. Bézivin, J.: In Search of a Basic Principle for Model Driven Engineering, CEPIS, UPGRADE. *The European Journal for the Informatics Professional* (2), 21–24 (2004)
5. Bouillon, L., Vanderdonckt, J.: Retargeting of Web Pages to Other Computing Platforms with VAQUITA. In: *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE 2002)*, p. 339 (2002)
6. Calvary, G., Coutaz, J., Daassi, O., Balme, L., Demeure, A.: Towards a new generation of widgets for supporting software plasticity: the 'comet'. In: Bastide, R., Palanque, P., Roth, J. (eds.) *DSV-IS 2004 and EHCI 2004*. LNCS, vol. 3425, pp. 306–324. Springer, Heidelberg (2005)



7. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. *Interacting With Computers* 15/3, 289–308 (2003)
8. Clerckx, T., Luyten, K., Coninx, K.: Generating Context-Sensitive. In: Sasse, A., Johnson, C. (eds.) *Multiple Device Interact 1999*, Edinburgh, pp. 110–117. IOS Press Publ., Amsterdam (1999)
9. Cockton, G.: A development Framework for Value-Centred Design. In: *ACM Proc. CHI 2005, Late Breaking Results*, pp. 1292–1295 (2005)
10. Constantine, L.L., Lockwood, L.A.D.: *Software for Use: A Practical Guide to the Models and Methods of Usage-Centred Design*. Addison-Wesley, New-York (1999)
11. Coutaz, J.: Meta-User Interfaces for Ambient Spaces. In: Coninx, K., Luyten, K., Schneider, K.A. (eds.) *TAMODIA 2006*. LNCS, vol. 4385, pp. 1–15. Springer, Heidelberg (2007)
12. Dix, A., Finlay, J., Abowd, G., Beale, R.: *Human-Computer Interaction*. Prentice-Hall, New-Jersey (1993)
13. *IFIP Design Principles for Interactive Software*, IFIP WG 2.7 (13.4), Gram, C., Cockton, G. (eds.). Chapman&Hall Publ. (1996)
14. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. In: Meersman, R., Tari, Z., et al. (eds.) *CoopIS 2002, DOA 2002, and ODBASE 2002*. LNCS, vol. 2519. Springer, Heidelberg (2002)
15. Lopez-Jaquero, V., Montero, F., Molina, J.P., Gonzalez, P.: A Seamless Development Process of Adaptive User Interfaces Explicitly Based on Usability Properties. In: Bastide, R., Palanque, P., Roth, J. (eds.) *DSV-IS 2004 and EHCI 2004*. LNCS, vol. 3425, pp. 289–291. Springer, Heidelberg (2005)
16. Mens, T., Czarnecki, K., Van Gorp, P.: A Taxonomy of Model Transformations Language Engineering for Model-Driven Software Development, Dagstuhl (February-March 2004)
17. Montero, F., Vanderdonckt, J., Lozano, M.: Quality Models for Automated Evaluation of Web Sites Usability and Accessibility. In: Koch, N., Fraternali, P., Wirsing, M. (eds.) *ICWE 2004*. LNCS, vol. 3140. Springer, Heidelberg (2004)
18. Mori, G., Paternò, F., Santoro, C.: CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Transactions on Software Engineering*, 797–813 (August 2002)
19. Myers, B., Hudson, S.E., Pausch, R.: Past, Present, and Future of User Interface Software Tools. *Transactions on Computer-Human Interaction (TOCHI)* 7(1) (2000)
20. Nielsen, J.: Heuristic evaluation. In: Nielsen, J., Mack, R.L. (eds.) *Usability Inspection Methods*. John Wiley & Sons, New York (1994)
21. Njike, H., Artières, T., Gallinari, P., Blanchard, J., Letellier, G.: Automatic learning of domain model for personalized hypermedia applications. In: *International Joint Conference on Artificial Intelligence, IJCA, Edinburg, Scotland*, p. 1624 (2005)
22. Nobrega, L., Nunes, J.N., Coelho, H.: Mapping ConcurTaskTrees into UML 2.0. In: Gilroy, S.W., Harrison, M.D. (eds.) *DSV-IS 2005*. LNCS, vol. 3941, pp. 237–248. Springer, Heidelberg (2006)
23. Paganelli, L., Paternò, F.: Automatic Reconstruction of the Underlying Interaction Design of Web Applications. In: *Proceedings Fourteenth International Conference on Software Engineering and Knowledge Engineering, July 2002*, pp. 439–445. ACM Press, Ischia (2002)
24. Palay, A., Hansen, W., Kazar, M., Sherman, M., Wadlow, M., Neuendorffer, T., Stern, Z., Bader, M., Peters, T.: The Andrew Toolkit: An Overview. In: *Proc. On Winter 1988 USENIX Technical Conf.*, pp. 9–21. USENIX Ass., Berkeley, CA, (1988)

25. Paterno', F., Mancini, C., Meniconi, S.: ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In: Proceedings Interact 1997, Sydney, pp. 362–369. Chapman&Hall, Boca Raton (1997)
26. Preece, J., Rogers, Y., Sharp, H., Benyon, D., Holland, S., Carey, T.: Human-Computer Interaction. Addison Wesley Publ., Wokingham (1994)
27. Rosson, M.B., Carroll, J.M.: Usability Engineering: Scenario-Based Development of Human-Computer Interaction. Morgan Kaufman, San Francisco (2002)
28. Seffah, A., Donyae, M., Kline, R.B.: Usability and quality in use measurement and metrics: An integrative model. Software Quality Journal (2004)
29. Shackel, B.: Usability-Context, Framework, Design and Evaluation. In: Human Factors for Informatics Usability, pp. 21–38. Cambridge University Press, Cambridge (1991)
30. Sheshagiri, M., Sadeh, N., Gandon, F.: Using Semantic Web Services for Context-Aware Mobile Applications. In: Proceedings of ACM MobiSys. 2004 Workshop on Context Awareness, Boston, Massachusetts, USA (June 2004)
31. Schneiderman, B.: Designing User Interface Strategies for effective Human-Computer Interaction, 3rd edn., 600 pages. Addison-Wesley Publ., Reading (1997)
32. da Silva, P.: User Interface Declarative Models and Development Environments: A Survey. In: Palanque, P., Paternó, F. (eds.) DSV-IS 2000. LNCS, vol. 1946, pp. 207–226. Springer, Heidelberg (2001)
33. Sottet, J.S., Calvary, G., Favre, J.M., Coutaz, J., Demeure, A., Balme, L.: Towards Model-Driven Engineering of Plastic User Interfaces. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 191–200. Springer, Heidelberg (2006)
34. Sottet, J.S., Calvary, G., Favre, J.M.: Towards Mappings and Models Transformations for Consistency of Plastic User Interfaces. In: The Many Faces of Consistency, Workshop CHI 2006, Montréal, Québec, Canada, April 22-23 (2006)
35. Sottet, J.S., Calvary, G., Favre, J.M.: Mapping Model: A First Step to Ensure Usability for sustaining User Interface Plasticity. In: Proceedings of the MoDELS 2006 Workshop on Model Driven Development of Advanced User Interfaces, October 3 (2006)
36. Thevenin, D.: Plasticity of User Interfaces: Framework and Research Agenda. In: Sasse, A., Johnson, C. (eds.) Proc. Interact 1999, Edinburgh, pp. 110–117. IFIP IOS Press Publ., Amsterdam (1999)
37. Trevisan, D., Vanderdonckt, J., Macq, B.: Continuity as a usability property. In: HCI 2003 - 10th Intl Conference on Human-Computer Interaction, Heraklion, Greece, June 22-27, 2003, vol. I, pp. 1268–1272 (2003)
38. UsiXML, <http://www.usixml.org/>
39. Van Welie, M., van der Veer, G.C., Eliëns, A.: Usability Properties in Dialog Models. In: 6th International Eurographics Workshop on Design Specification and Verification of Interactive Systems DSV-IS 1999, Braga, Portugal, 2-4 June 1999, pp. 238–253 (1999)
40. Zoommm Project, <http://www.zoommm.org>

## Questions

**Yves Vandriessche:**

*Question: How are you handling the layouts, should there be a model?*

**Answer:** The layout model is in the transformation but we should really do that in another model.

*Question: There is no problem adding more models?*

Answer: No problems, this is what the Zoom project is about.

**Nick Graham:**

*Question: Tell me about your platform model?*

Answer: It is very simple, work by Dennis Wagelaar is interesting and I would like a more complex model.

**Jan Gulliksen:**

*Question: What about the end-user as designer, how difficult is it?*

Answer: I am interested in end-user programming. I would like to achieve that and this is what we would like to do in the future.

**Phil Gray:**

*Question: Single task single user, what about multiple user multiple task?*

Answer: Yes we have multiple users. How the task is described – we are talking about a Petri net model as a means of describing this. For some users some models are better than others, an evolution model is something we are working on in the team.

**Jo Vermeulen:**

*Comment: An interesting paper around a meta user interface editors is "User Interface Façades" which was presented at UIST last year. End-users are able to create new dialogs combining a couple of widgets from an existing dialog, or transform widgets (e.g. switch from a group of radio buttons to a combo box). This might be useful for your work if you want to look at extending it to enable user interface adaptation by end-user.*

*The exact details of the paper:*

*W. Stuerzlinger, O. Chapuis, D. Phillips and N. Roussel. User Interface Façades: Towards Fully Adaptable User Interfaces. In Proceedings of UIST'06, the 19th ACM Symposium on User Interface Software and Technology, pages 309-318, October 2006. ACM Press. URL: <http://insitu.lri.fr/metisse/facades/> PDF: <http://insitu.lri.fr/~roussel/publications/UIST06-facades.pdf>*

# Model-Driven Prototyping for Corporate Software Specification

Thomas Memmel<sup>1</sup>, Carsten Bock<sup>2</sup>, and Harald Reiterer<sup>1</sup>

<sup>1</sup> Human-Computer Interaction Lab, University of Konstanz, Germany  
{memmel, reiterer}@inf.uni-konstanz.de

<sup>2</sup> Dr. Ing. h.c. F. Porsche AG, Germany

**Abstract.** Corporate software development faces very demanding challenges, especially concerning the design of user interfaces. Collaborative design with stakeholders demands modeling methods that everybody can understand and apply. But when using traditional, paper-based methods to gather and document requirements, an IT organization often experiences frustrating communication issues between the business and development teams. We present ways of implementing model-driven prototyping for corporate software development. Without harming agile principles and practice, detailed prototypes can be employed for collaborative design. Model-driven prototyping beats a new path towards visual specifications and the substitution of paper-based artifacts.

**Keywords:** Prototyping, model-driven user interface design, UI specification, corporate software development, agile modeling.

## 1 Introduction

From the authors' experience with the automotive industry, we see that many companies strive to further increase their influence on the user interface design (UID) of corporate software systems. The risk of bad user interface (UI) design and usability is considerable, and it is an economic risk. But integrating usability engineering (UE) often causes conflicts with other stakeholders and faces shrinking IT budgets and pressure of time.

Several ingredients can therefore contribute to development failure: the increasing importance of the UI in the overall system, the separation of professions, particularly software engineering (SE) and UE, and consequently a lack of methods, tools and process models that integrate the UE knowledge of the UI expert with that of the software engineer and other stakeholders. All issues must be addressed from the very beginning, when the software systems are defined. Consequently, new approaches to requirements engineering (RE) and specification practice are necessary.

In this article we introduce a model-driven prototyping approach to the development of interactive corporate software systems. By employing prototypes as vehicles for both design and system specification, we are able to bridge existing gaps while making the overall design process more efficient and effective, resulting in lower development costs, but improved software quality. Simultaneously we address the different requirements of stakeholders by offering different levels of abstraction and formality.

In Section 2 we summarize the importance of UID for corporate software development and point out the various challenges that automotive engineering processes have to

face. We contrast the demands with the current shortcomings of wide-spread RE practice and propose a change of applied practice. In Section 3, we show how SE and UE can be bounded through changing the RE up-front, and how requirement specification takes place. We outline the interdisciplinary usage of prototyping and encourage an extended role for prototyping during RE. In Section 4, the resulting design approach is compared with the principles and practice of agile software development. We discuss in detail why the extension of the RE phase is still compatible with agile development. Consequently, in Section 5 we present our concept of a model-driven tailored tool support for developing prototyping-based specifications of interactive systems. We summarize our experiences and lessons learned in Section 6.

## 2 Corporate Software Development

The UI is the part of the software that can help users to work more efficiently and effectively. When users are unable to perform their tasks, the usage of the software may be entirely incorrect, slow to make progress, and may finally lead to reduced acceptance. With corporate software, the UI transports important (emotional) values such as corporate design (CD) and corporate identity (CI). In the automotive industry, a wide range of different software systems is available to the customer. For example, a company website is necessary to create brand awareness, transport product values, enable product search and configuration, allow contact to nearby retailers, and finally to increase customer loyalty. But in this article we concentrate on the development of in-car information systems. Such systems are intended to support the driver during traveling, e.g. with GPS navigation or dynamic traffic information. Such systems must never compromise road safety [1] and the respective UIs must be intuitive and easy to use. Embedded systems play an important role in the market success of an automotive brand and the customer acceptance of its products [2].

### 2.1 Challenges for Corporate Engineering Processes

As well as the implementation of pure functionality, corporate software products demand the integration of usability and design consideration into the development process. This frequently conflicts with strict timelines, leading to coding being started prematurely, while system requirements are still vague or unknown. Typical SE and UE processes lack flexibility and adaptability when facing changing requirements, which results in increased complexity and costs.

Consequently, many companies became receptive to agile methods of SE and UE. Agile methods travel in a light-weight fashion along the software lifecycle and due to less documentation, more communication, sharing of code and models, and special programming methods etc., they successfully address many issues of corporate software development. On the other hand, agile methods do not provide room for typical UE and UID and need to be extended by a certain degree of design and usability expertise that is integrated into the methods and tools applied.

### 2.2 Shortcomings of Current Requirements Engineering Practice

UE usually documents design knowledge in style guides that can easily reach a size of hundreds of pages and require hundreds of hours of effort. But written language is

ambiguous and the lack of visual cues leaves room for misinterpretation. Especially when interactive behavior has to be specified, a picture is worth a thousand words and “[...], the worst thing that any project can do is attempt to write a natural language specification for a user interface” [3].

In a survey of ergonomists, designers and technical experts, we found that a majority of stakeholders use typical office software products for the specification of automotive software systems [4]. This is caused by the difficulty of customizing or even building CASE-tools [5] and, more importantly, by their poor usability [6]. But as with the use of natural language, employing applications such as Microsoft PowerPoint, Word, Excel or Visio does also highlight critical shortcomings for engineering interactive systems. First of all, stakeholders choose their favorite software application independently and according to their individual preferences. This inevitably leads to a wide variety of formats that often cannot be interchanged without loss of precision or editability. Secondly, those who are responsible for actually coding the software system will use completely different tools during the implementation process. Consequently, the effort invested in drawing PowerPoint slides or Excel sheets does not help programming of the final system. Virtual prototypes cannot automatically be created from such specifications with justifiable effort [4].

But prototyping is necessary to provide rapid feedback and to guide the overall specification process towards an ultimate design solution [7]. The participation of non-technical personnel inevitably leads to the demand for a common modeling language throughout the lifecycle. Otherwise, stakeholders may think they all agree on a design, only to discover down the line that they had very different expectations and behaviors in mind. Hence, there is a need to have one common denominator of communication.

### **3 Prototyping for Visual Specification**

The Volere RE process outlines the most important activities for system specification [8]. This includes trawling for requirements, and their separation into functional (e.g. data structures, data models, algorithms, error handling, behavior) and non-functional (e.g. reliability, safety, processing time, compliance with guidelines and regulations, usability, look and feel) requirements. This categorization of requirements also mirrors the different competencies of SE and UE.

#### **3.1 The Interaction Layer: Where SE and UE Meet**

Software engineers are generally trained in topics such as system architecture or database design, while usability engineers are concerned with e.g. ease of use, ease of learning, user performance, user satisfaction and aesthetics. A usability expert normally has a black box view of the back-end system, while the software engineer has a deeper understanding of the architecture and code behind the UI [9]. Although both disciplines have reached a certain degree of maturity, they are still practiced very independently [10]. Consequently, usability engineers and software developers, as well as (interaction) designers, end-users and business personnel express themselves in quite different fashions, ranging from informal documents (e.g. scenarios) to formal models (e.g. UML).

However, the behavior of the system and the feel of the UI are very much dependent on each other. The more important the UI component becomes for a software application, the more significant is its impact on the back-end system. Hence, SE and UE need to overlap at the interaction layer in order to develop usable systems. If the collaboration at the interaction layer is well defined and working successfully, “the time to market can be dramatically shortened by (having) well defined interaction points for the teams to re-sync and communicate” [11].

### 3.2 Prototyping for the Visual Specification of Interactive Systems

The Volere RE process employs (throw-away) prototypes as vehicles for requirements elicitation [8]. SE recognizes prototyping as a method for inspections, testing and incremental development. HCI uses prototypes mainly for participatory design (PD). Prototypes are an “excellent means for generating ideas about how a UI can be designed and it helps to evaluate the quality of a solution at an early stage” [12]. Prototypes can therefore be boundary objects for SE, UE and other stakeholders as they are a common language to which all can relate [11, 12, 13].

With Volere, the purpose of prototypes is restricted to requirements gathering. After the requirements have been written down and forwarded to the quality gateway, they are still documented in a paper-based requirements specification. This is exactly where room for interpretation emerges and where misinterpretations can lead to misunderstandings and cause expensive late-cycle changes. Hence, the role of prototyping must be extended and the visual expressiveness of prototypes must be anchored in corporate software specification processes.

As the early externalization of design visualizations helps to elicit requirements and enables a better understanding of the desired functionality (SE), the users and their tasks (UE), then prototypes of specific fidelity can also be a cornerstone for system specification (Figure 1). Wherever something can be visually expressed in a more understandable and traceable way, prototypes should replace formal documents. For example, in usual UE practice, style guides are developed to be a reference document for designers, to share knowledge, to ensure consistency with UID standards and to save experience for future projects [14]. A running simulation also includes and externalizes much of this knowledge.

If a visual specification is created and assessed collaboratively, it could ensure that the final system design satisfies all requirements. Consequently, the responsibility of designing the UI is pre-drawn to the RE phase. Corporate RE practice must therefore extend the role of prototypes to visual specifications (Figure 1).

Executable specifications also have to fulfill the quality criteria of paper-based software specifications. By sharing and collaboratively discussing prototypes, different user groups (user roles) can cross-check the UID with their requirements (correctness, clearness). It is unlikely that certain user groups will be ignored when stakeholders have access to a UID prototype (completeness). When certain user tasks demand exceptional UID, a visual simulation will be more capable of expressing such complex parts of the system and able to illustrate their meaning and compliance in the system as a whole (consistency, traceability). Ambiguity, redundancy, missing information and conflicts will be also be more obvious. A usability engineer will be able to identify ambiguity through the evaluation of UI expressiveness, affordance and mapping. He will be able to identify redundancy and conflicts when assessing screen

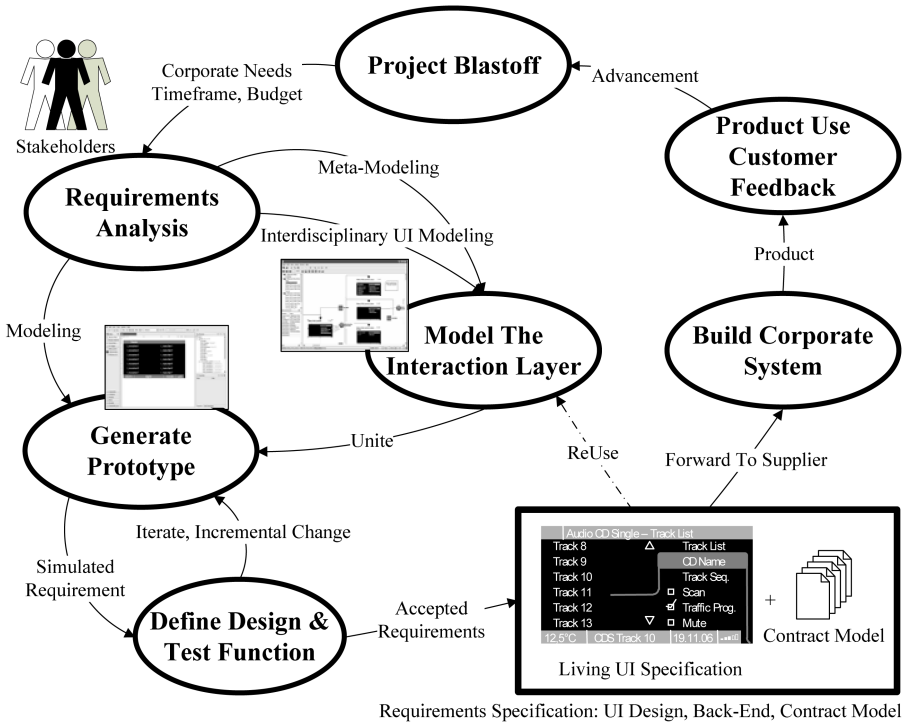


Fig. 1. Requirements engineering for visual specification, based on [8]

spacing, layout or navigation structure. The absence of specific information will attract attention through “white spots” on the screen or missing visual components. With interactive and expressive prototypes, interaction and functional issues can be addressed sooner and the identification of usability requirements can be done as soon as the early stages of design [15] and before coding starts (Figure 1). Unnecessary system functionality can be identified through UI evaluation methods rather than by reading through text. All in all, a visual requirements specification can be assessed more easily and to some extent the creation of a prototype (pilot system) proves the convertibility of the UID into a final system.

Prototyping can significantly reduce the effort on the programming side as well: to build the UI of a system with the help of a running simulation (prototype) is much easier than doing it from scratch based on textual descriptions. A developer can quickly look at the simulation in order to get a visual impression of the requirements. Moreover, when the creation of prototypes takes place in a model-driven engineering process, this results in further advantages for the RE process. Models can imply constraints and can carry forward standards and design rules. Once properly defined, they can be externalized by different representations and in a different level of abstraction and formality. On the one hand, this eases access for different stakeholders. On the other hand, while a running simulation is the visual outcome, the underlying models capture design knowledge and decisions. Consequently, we encourage a model-driven approach that employs prototypes as media of communication.



### 4 Compliance with Agile Software Development

Before presenting our model-driven prototyping approach, we want to outline its compliance with agile environments. At first sight, adding up-front visual specification activity does appear to contradict agile software development: agile software lifecycles, e.g. their most popular representative Extreme Programming (XP) [16], encourage

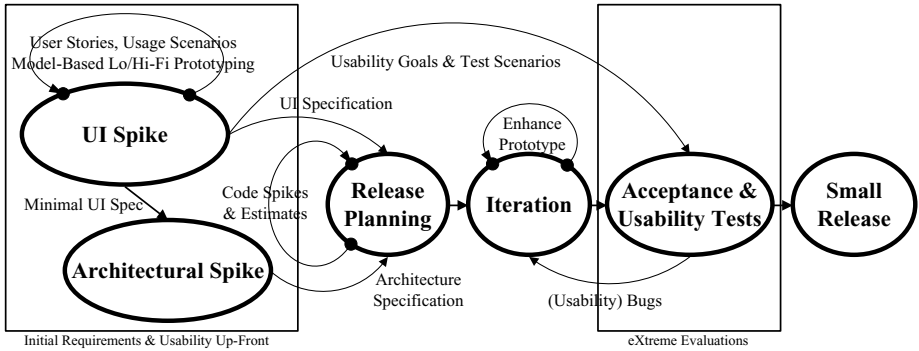


Fig. 2. The XP lifecycle extended by UE methods during up-front and test phase

Table 1. Core principles (excerpt) of agile development, and their compatibility with a model-driven software development approach (<http://www.agilemodeling.com>)

Agile principle	Compatibility with model-driven development
Model With A Purpose	Switching between different models allows the origin of a requirement to be traced and to understand its design rationale
Multiple Models	No single model is sufficient for all software development needs. By providing different modeling layers, several ways of expressing problems are available
Rapid Feedback	A model-driven approach allows a fast transformation of models into assessable, living simulations
Assume Simplicity	Models allow an easy cross-checking with actual requirements. Unnecessary requirements can be identified visually
Embrace change	Models should allow easy enhancement or change. Changes to an abstract representation layer should have impact on the generation of prototypes and code
Incremental Change	Different layers of detail allow domain-specific access for stakeholders and enable the creation of small models first, more sophisticated models later
Software Is Your Primary Goal	The primary goal of software development is to produce code rather than extraneous documentation. A model-driven approach does directly contribute to this goal by enabling fast generation of prototypes and reusable code

**Table 2.** Core practices of agile development, and their compatibility with a model-driven software development approach (<http://www.agilemodeling.com>)

<b>Agile Practice</b>	<b>Compatibility with model-driven development</b>
Active Stakeholder Participation	When different models are provided for different stakeholders, everybody can take part in system specification. Prototypes are a language everybody understands, and the perfect vehicle for discussion
Apply The Right Artifacts	Some modeling languages are inappropriate for describing specific parts of the system. For example, UML is insufficient for describing the UI, but helps in designing the architecture
Create Several Models In Parallel	A model-driven approach allows the parallel development of different models (e.g. regarding disciplines and competencies)
Iterate To Another Artifact	When a specific artifact is unable to express certain parts of the system, one should iterate to other methods of expression
Model in Small Increments	A model-driven approach allows models to be charged with different levels of detail. Small releases can be provided very quickly to get rapid feedback and can be refined later
Model with others	A model-driven approach allows a break-up of responsibility. Stakeholders can model according to their expertise. Different models are combined to simulations and code
Model to communicate	To some extent, models need to look attractive for showing them to decision makers. By including a design layer, a model-driven approach can provide system simulations that can be used for discussion and release planning
Model to understand	Modeling helps to understand and to explore the problem space. For exploring alternate solutions, you do not need to draw UML or class diagrams. A model-driven approach can provide appropriate modeling languages that support the stakeholders during early stages of design.
Prove it with code	A model is an abstraction of whatever you are building. To determine whether it will work, a model-driven approach allows the easy generation of prototypes and running code.
Formalize Contract Models	The code that is exported on the basis of the description models can be supported by e.g. an XML DTD. As other design knowledge and guidance is included as a running simulation, a visual specification is a detailed contract model.

coding from the very beginning. However, interdisciplinary research has agreed that a certain amount of UE and UID up-front is needed in XP, when the UI has great weight. [17] developed a model-driven, usage-centered UE approach. Similar to the idea of model-driven SE, the models of usage-centered design (e.g. user role models, task models, content models) make UID a more traceable and formal activity. Compared to typical UE practice, the suggested models are more light-weight and allow the early definition of a minimalist UI specification. In addition, a supplier can implement the system with less effort, as important parts of the UID are already pre-defined and

evaluated (Figure 1). Consequently, adding an UI spike to agile methods delays the overall system implementation, but only to a limited extent. Figure 2 shows how UID and UI specification can be integrated into XP's up-front. Additionally, our adjusted XP lifecycle model also envisages "extreme evaluations" [18] at later stages of design.

The bottom-line, using visual specifications will help in cutting down the number of iteration cycles and decrease software development costs, while simultaneously assuring corporate quality and compatibility with agile principles (Table 1) and practice (Table 2).

## 5 A Model-Driven Tool-Chain for Visual Specification

Based on our experience in corporate RE processes, we have developed a tool-chain for the agile and interdisciplinary specification of interactive automotive in-car information systems. Our tool-chain helps to bridge identified gaps and links up with the idea of model-driven software development.

### 5.1 Model-Driven Concepts as a Cornerstone for Structured RE

The synchronization of models and code is, above all, a regular source of serious problems in model-based development processes [19]. With its core concepts, the Model Driven Architecture (MDA) aims at overcoming the flaccidities of model-based software development by taking code as a by-product. As Figure 3 shows, this is possible since the code can be automatically generated in ideal model-driven development processes, and it results from model transformations (AM: Model With Purpose, Software Is Your Primary Goal).

The starting point for such model transformations is a platform-independent model (PIM) providing an abstract description of the system under development. By means of transformations a platform-independent model can be derived, holding additional information about a specific target platform. Finally, from this implementation we can generate specific model target code. Since this concept clearly resembles the Object Managements Group's (OMG's) four layer meta-model hierarchy [20] the MDA offers modularization and abstraction throughout software development processes.

Although the UML was established as an industry standard, its broad acceptance in many industries is hindered due to its general-purpose graphical language representations mapping only poorly onto the architecture of underlying platforms [21]. Despite UML profiles and many attempts for improvements [22, 23] the UML is still particularly unsuitable for modeling UIs [24]. UML is visually too awkward as it can not (visually) express the look and feel of an UI. Apart from software engineers, other stakeholders usually cannot understand UML. Moreover, even system developers find "CASE diagrams too dry, notations and editors too restrictive, impeding rather than helping in their work" [6].

This leads to the conclusion that, especially for RE in interdisciplinary teams, simultaneously interfacing with the realms of UE and SE, appropriate tool support is badly needed (see Section 3).

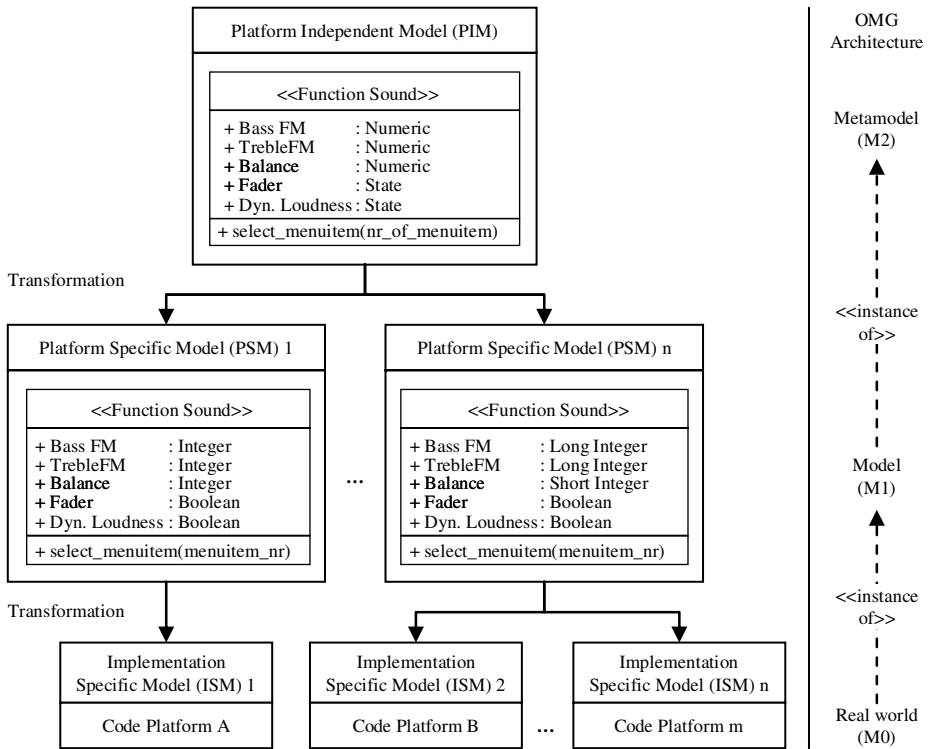


Fig. 3. Core models of MDA

## 5.2 Modularization and Abstraction as a Key for Interdisciplinary Cooperation

When developing interactive graphical systems, close cooperation between team members with different backgrounds, knowledge and experiences is one of the key success factors [25, 26, 27, 28]. As stated in Section 3, a common language is missing today. Although prototypes can be a common denominator for UID at the interaction layer each discipline still needs to employ other (more abstract) modeling languages during development (AM: Iterate To Another Artifact).

Therefore, one approach for coping with the inherent technical complexity of interactive systems [29] and the organizational complexity stemming from the indispensable interdisciplinarity is a strict separation of concerns [30], i.e. the modularization of development tasks (Figure 4). This is also consistent with the well-known Seeheim model [31]. The following categories are integral parts of UI development and thus constitute the intersecting domain of RE, UE and SE:

- **Layout:** relates to screen design and the ergonomic arrangement of dialog objects.
- **Content:** refers to the definition of information to be displayed.
- **Behavior:** describes the dynamic parts of a GUI with respect to controls available on a specific target platform and a system’s business logic.

Accordingly, in the case of embedded UIs such as automotive driver-information systems, GUI layout is specified by designers and ergonomists. Consequently, contents in the form of menu items are provided by technical experts responsible for functional (i.e. hardware) specifications as well as by ergonomists defining menu structures. Finally, system architects or programmers define a system’s behavior, again in close coordination with ergonomists. To some extent, these activities will always take place in parallel (AM: Multiple Models, Model In Parallel). Altogether, ergonomists are propelling and controlling the development process. Furthermore, they are responsible for integrating all artifacts and function as human interface between the competencies and professions of all stakeholders.

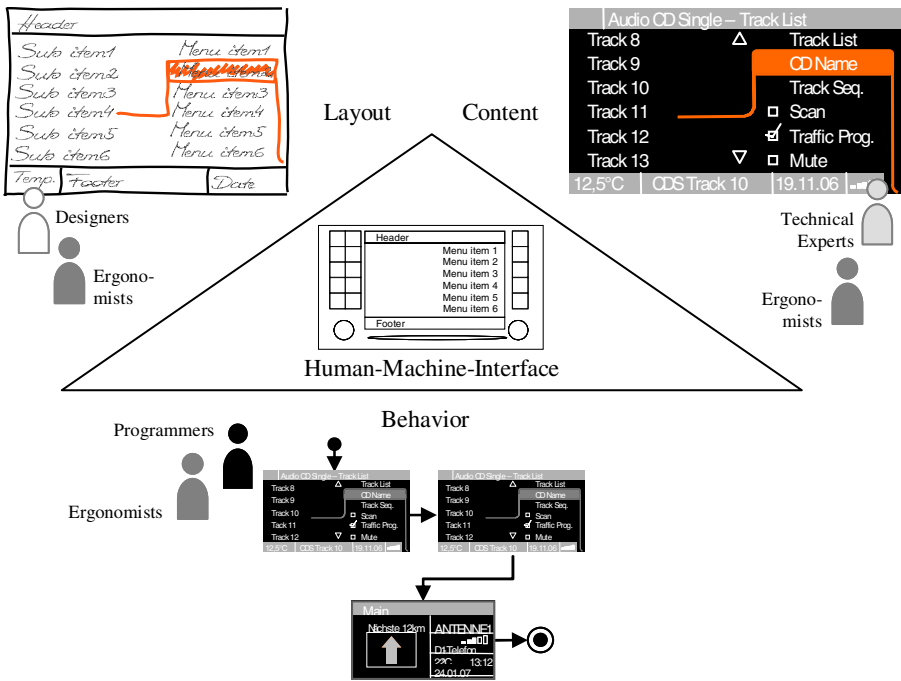


Fig. 4. Layout, content and behavior as constituent parts of interactive graphical systems

In the realm of UI development, the separate specification of layout, content and behavior must not take place completely independent and needs to follow certain constraints (meta-model, Section 5.3) to guarantee compatibility. Changing a specific model (e.g. an abstract one) must consistently affect dependent models (e.g. a more detailed one) and consequently the prototype as well (AM: Embrace Change). For the generation of the UI prototype, the different parts finally have to be integrated (Figure 4, compare Figure 1).

As well as modularization, abstraction also offers further means for coping with technical and organizational complexity. By presenting information at different levels of abstraction, it is possible to provide developers with only the relevant information

for their specific development tasks (AM: Apply The Right Artifacts). Thus, for working collectively on high-level specifications where contents and the overall system behavior is specified, developers are provided with a very abstract representation of a state machine by means of visual state charts [32]. These charts only reveal the details necessary for specifying the high-level interaction with the system, the “macro logic”. Developers can therefore connect different menu screens with interaction objects (e.g. rotary knobs, buttons) corresponding to transitions in UML state charts. This presentation must be extraordinarily abstract so that all the developers involved – and, if necessary, even project managers – can understand such high-level specifications (Figure 5).

High-level specification

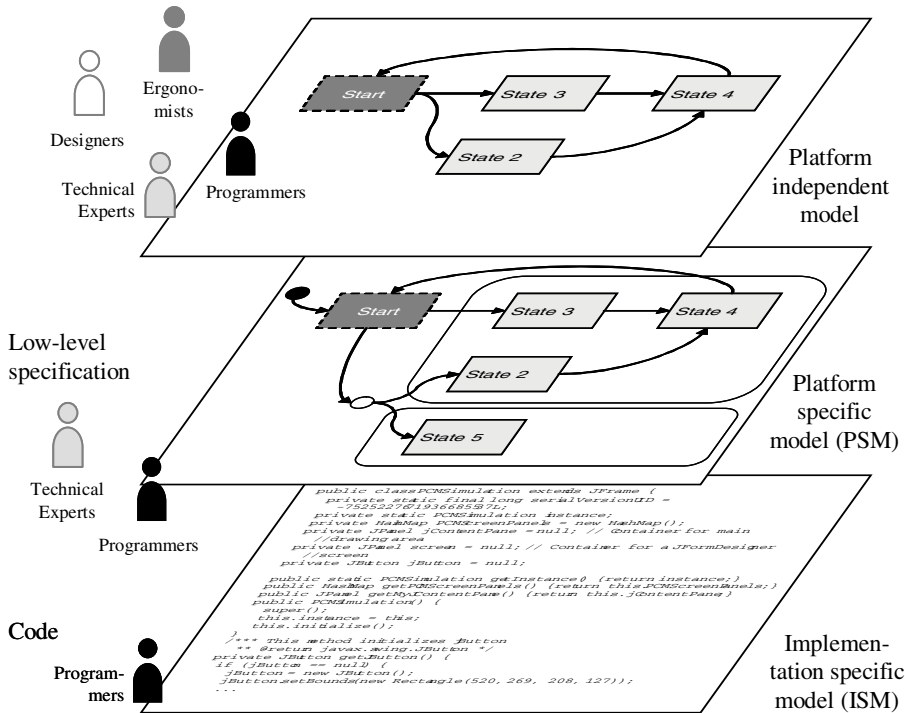


Fig. 5. Problem- and target group-specific presentation of information

Accordingly, low-level specifications are meant for smaller target groups such as technical experts and programmers. These specifications contain a more detailed system specification. At this level, the behavior of specific widgets (“micro logic”) such as a speller for inputting a navigation destination can explicitly be defined with fully featured state charts and class diagrams. Finally, the code level allows for a rigorous analysis of system features and their implementation, which can only be

conducted by programmers. These abstraction levels correspond to the MDA concepts of PIMs, PSMs and ISMs respectively. Despite this equivalence, model-driven tool support for creating UI specifications at different levels of abstraction is still missing in corporate software development. An approach for creating tailor-made CASE-tools is therefore presented subsequently. This enables clients to take full advantage of model-driven concepts during UI development with interdisciplinary development teams.

### 5.3 Developing a Model-Driven Tool Chain for UI Specification

In the following, domain-specific modeling is used for creating an individual tool support for UI specification. By leveraging current meta-CASE-tools, for instance MetaEdit+ 4.5, Generic Modeling Environment 5 or Microsoft DSL Tools, this modeling approach enables clients to utilize model-driven concepts at affordable time and budget for building tailor-made CASE-tools for any company- and/or project-specific domain. Thus, the procedure for developing a visual domain-specific language (VDSL) that fulfils the aforementioned requirements is described.

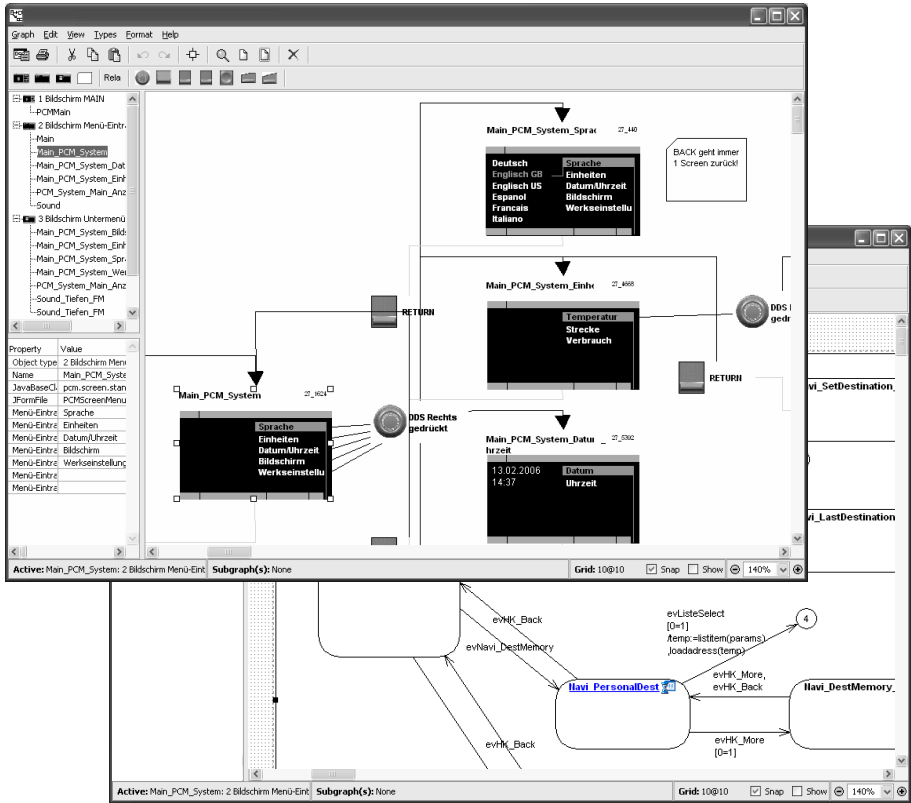
The creation of a domain meta-model by the mapping of domain concepts constitutes the starting point for developing a specific CASE-tool. Beyond the identification and abstraction of domain concepts, the development of a visual DSL comprises the definition of notations for the graphical representation of domain concepts and the definition of constraints underlying the specification process. For these tasks meta-CASE tools provide a meta-modeling environment that can subsequently also be used as a graphical modeling environment and thereby as a specification instrument for product development in a specific problem domain.

At the beginning, a small team of domain experts collaboratively identifies the essential concepts of the problem domain. At this stage, existing requirements documents such as style guides, and particularly the terminology used in daily project work, are analyzed (Figure 1). For instance, in the case of automotive driver-information systems, single menu screens and controls like rotary knobs and pushbuttons represent the main concepts of the problem domain.

The domain experts can quickly identify these concepts since they are frequently used for product specification. Additionally, the events to which the system should react are included, such as turning and pressing a rotary knob, or pressing and holding a pushbutton. Similarly, all the properties of every single domain concept necessary for specifying driver-information systems are defined.

Later, constraints are added to the meta-model in order to restrict the degrees of freedom for developers in a reasonable way. For instance, the use of some controls is limited to special circumstances. Moreover, constraints are defined that limit the number of subsequent menu screens after selecting a menu item to at most one, ensuring that specifications will be non-ambiguous. Additional constraints could prescribe a fixed pushbutton for return actions, for example. It must be stated that the definition of the meta-model therefore determines the overall design space. Consequently, the exploration of challenging design alternatives and innovative UID approaches must take place before the meta-model is fixed by the domain experts.

High-level specification



Low-level specification

**Fig. 6.** Individual CASE-tools for specifying content and behavior of driver-information systems (implemented with MetaEdit+ 4.5)

In a final step, meaningful pictograms are defined for domain concepts in the meta-model, thus allowing for intuitive use by developers during system specification. These self-made symbols clearly resemble the modeled domain concepts. Our experience from a comparison of different specification tools and interviews with developers reveals that especially these individual, domain-specific symbols lead to a very small semantic distance between the specification language and real-world objects of the driver-information system domain. Most notably this strongly increases developers’ acceptance of tailor-made CASE tool [33].

With this DSL, the content and the macro logic of driver-information systems can be specified. In order to also describe the micro logic (see Section 5.2) of widgets, another DSL is developed in exactly the same way as previously illustrated. This DSL enables IT-experts to create UML state charts for detailed UI specifications. These individual CASE tools are shown in Figure 6.



Besides the domain-specific CASE-tools (for modeling the dynamic parts of interactive systems, i.e. content and behaviour) a domain framework provides the static parts (i.e. a state machine and base widgets) for virtual simulations. Thus for creating virtual simulations from specifications the content and behavior needs to be extracted from the models and linked to the (static) framework. This is done with the help of a code generator enabling developers to create simulations from a specification on the push of a button.

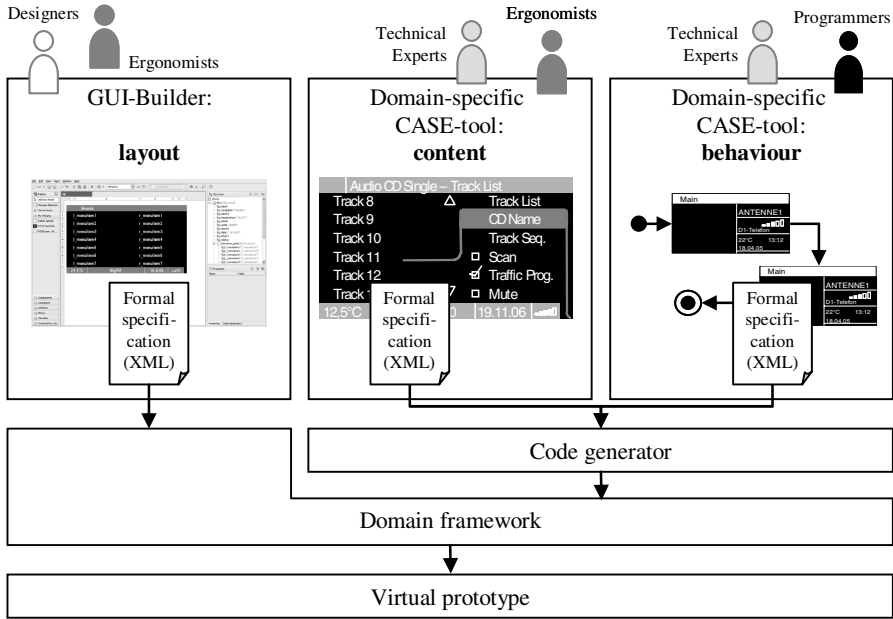


Fig. 7. Architecture of model-driven UI tool-chain

Finally, for specifying GUI layout designers and ergonomists ideally have to create specifications with a GUI-builder such as JFormdesigner, Expression Interactive Designer (Microsoft) or Aurora XAML Designer (Mobiform) capable of producing GUI descriptions in a XML format. These descriptions can also easily be integrated in the simulation framework by the code generator. The architecture of this tool chain is outlined in Figure 7.

## 6 Lessons Learned and Conclusion

Our experience reveals that model-driven approaches provide promising concepts for overcoming today's urgent problems in development processes for corporate software systems. Thus, visual domain-specific languages explicitly capture experts' knowledge and experience, since this is used for identifying and modeling essential domain concepts. With the help of languages created this way, new team members can become

acquainted with company-specific standards more easily and can thus be integrated in interdisciplinary development teams significantly more easily. Moreover, domain-specific modeling, together with the tool chain architecture presented, enables clients to create tailor-made tool support that offers all developers the appropriate level of abstraction for their individual development tasks. Furthermore, problems have to be solved only once at a high level of abstraction and not – as before – once in the implementation level and a second time for documentation. Specifications i.e. requirements engineering can therefore be established as the central backbone of model-driven development processes, with significant potential for clients and their collaboration with suppliers. If formal, electronic – and thus machine readable – specifications can be exchanged between all stakeholders, the specification problem as well as the communication problem in traditional development processes [27] can be overcome.

In principle, the concepts presented can be adopted for any other domain besides the UID of automotive embedded systems. For instance, a visual domain-specific language could be created for specifying the structure of corporate websites or the information flow in a business process. Despite this flexibility and the potential benefits, experience from a pilot project shows that current meta-CASE-tools can be improved. In particular, developers would expect interaction patterns from standard office applications e.g. auto layout, grids, object inspectors and tree views for object hierarchies. Additionally, if these tools provided better graphical capabilities, the GUI builder would not have to be integrated via the domain framework and layout could also be specified with a domain-specific language. This would be another important step in reducing complexity in usually heterogeneous IT landscapes.

Overall, meta-modeling offers promising beginnings for a unification of engineering disciplines. As demonstrated with the tool chain presented, this modeling approach is consistent with agile principles and practice. This offers an opportunity for bringing RE, SE and UE closer together, a convergence that is badly needed for coping with the technical and organizational complexity of interdisciplinary and networked development processes for corporate software systems.

## References

1. Rudin-Brown, C.: Strategies for Reducing Driver Distraction from In-Vehicle Telematics Devices: Report on Industry and Public Consultations, Technical Report No. TP 14409 E, Transport Canada, Road Safety and Motor Vehicle Regulation Directorate (2005) [cited: 21.5.2006], <http://www.tc.gc.ca/roadsafety/tp/tp14409/menu.htm>
2. Becker, H.P.: Der PC im Pkw: Software zwischen den Welten, automotive electronics systems (3-4), 42–44 (2005)
3. Horrocks, I.: Constructing the user interface with statecharts. Addison-Wesley, Harlow (1999)
4. Bock, C., Zühlke, D.: Model-driven HMI development – Can Meta-CASE tools relieve the pain? In: Proceedings of the First International Workshop on Metamodelling – Utilization in Software Engineering (MUSE), Setúbal, Portugal, September 11, 2006, pp. 312–319 (2006)
5. Isazadeh, H., Lamb, D.A.: CASE Environments and MetaCASE Tools, Technical Report No. 1997-403, Queen's University [cited: 25.10.2006], <http://www.cs.queensu.ca/TechReports/reports1997.html>
6. Jarzabek, S., Huang, R.: The case for user-centered CASE tools. Communications 41(8), 93–99 (1998)

7. Fitton, D., Cheverst, K., Kray, C., Dix, A., Rouncefield, M., Saslis-Lagoudakis, G.: Rapid Prototyping and User-Centered Design of Interactive Display-Based Systems. *Pervasive Computing* 4(4), 58–66 (2005)
8. Robertson, S., Roberston, J.: *Mastering the Requirements Process*. Addison-Wesley, Reading (2006)
9. Creissac Campos, J.: The modelling gap between software engineering and human-computer interaction. In: Kazman, R., Bass, L., John, B. (eds.) *ICSE 2004 Workshop: Bridging the Gaps II, The IEE*, pp. 54–61 (May 2004)
10. Pyla, P.S., Pérez-Quñones, M.A., Arthur, J.D., Hartson, H.R.: Towards a Model-Based Framework for Integrating Usability and Software Engineering Life Cycles, in *IFIP Working Group 2.7/13.4*, editor, *INTERACT 2003 Workshop on Bridging the Gap Between Software Engineering and Human-Computer Interaction* (2003)
11. Gunaratne, J., Hwong, B., Nelson, C., Rudorfer, A.: Using Evolutionary Prototypes to Formalize Product Requirements. In: *Proceedings of ICSE 2004 Bridging the Gaps Between Software Engineering and HCI*, Edinburgh, Scotland, pp. 17–20 (May 2004)
12. Bäumer, D., Bischofberger, W.R., Lichter, H., Züllighoven, H.: User Interface Prototyping - Concepts, Tools, and Experience. In: *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, Berlin, Germany, pp. 532–541 (March 1996)
13. Rudd, J., Stern, K., Isensee, S.: Low vs. high fidelity prototyping debate. *Interactions* 3(1), 76–85 (1996)
14. Mayhew, D.J.: *The usability engineering lifecycle - A Practicioners Handbook for User Interface Design*. Morgan Kaufmann, San Francisco (1999)
15. Folmer, E., Bosch, J.: Cost Effective Development of Usable Systems - Gaps between HCI and Software Architecture Design. In: *Proceedings of ISD 2005*, Karlsstad, Sweden (2005)
16. Beck, K.: *Extreme Programming Explained*. Addison-Wesley, Reading (1999)
17. Constantine, L.L., Lockwood, L.A.D.: *Software for Use: A Practical Guide to Models and Methods of Usage-Centered Design*. Addison-Wesley, Reading (1999)
18. Gellner, M., Forbrig, P.: Extreme Evaluations – Lightweight Evaluations for Software Developers, in *IFIP Working Group 2.7/13.4*, editor, *INTERACT 2003 Workshop on Bridging the Gap Between Software Engineering and Human-Computer Interaction* (2003)
19. Stahl, T., Völter, M.: *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, San Francisco (2006)
20. Object Management Group, *UML 2.0 Infrastructure Specification* (2003) [cited: 4.12.2006], <http://www.omg.org/docs/ptc/03-09-15.pdf>
21. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. *Computer* 39(2), 25–31 (2006)
22. Nunes, N.J.: *Object Modeling for User-Centered Development and User Interface Design: The Wisdom Approach*, PhD Thesis, Universidade Da Madeira (2001) [cited: 12.8.2006], <http://xml.coverpages.org/NunoWisdomThesis.pdf>
23. Blankenhorn, K., Jeckle, M.: A UML Profile for GUI Layout. In: *Proceedings of the 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts and Applications for a Networked World*, Net. *ObjectDays 2004*, Erfurt, pp. 110–121 (2004)
24. da Silva, P.P., Paton, N.W.: User Interface Modelling with UML. In: *Proceedings of the 10th European-Japanese Conference on Information Modelling and Knowledge Representation*, Saariselkä, Finland, May 8-11, pp. 203–217 (2000)
25. Wong, Y.Y.: Rough and ready prototypes: lessons from graphic design. In: *CHI 1992: Posters and short talks of the 1992 SIGCHI conference on Human factors in computing systems*, Monterey, California, pp. 83–84 (1992)

26. Hardtke, F.E.: Where does the HMI end and where does the Systems Engineering begin? In: Proceedings of the Systems Engineering, Test & Evaluation Conference (SETE) (2002) [cited: 27.10.2006], <http://www.seecforum.unisa.edu.au/Sete2002/ProceedingsDocs/09P-Hardtke.pdf>
27. Rauterberg, M., Strohm, O., Kirsch, C.: Benefits of user-oriented software development based on an iterative cyclic process model for simultaneous engineering. *International Journal of Industrial Ergonomics* 16(4-6), 391–410 (1995)
28. Borchers, J.O.: A pattern approach to interaction design. In: Proceedings of the conference on Designing interactive systems (DIS 2000), New York, pp. 369–378 (2000)
29. Szekely, P.: User Interface Prototyping: Tools and Techniques, Intelligent Systems Division, Technical Report, Intelligent Systems Division, University of Southern California (1994)
30. Dijkstra, E.W.: *A Discipline of Programming*. Prentice Hall, Englewood Cliffs (1976)
31. Green, M.: A survey of three dialogue models. *Transactions on Graphics* 5(3), 244–275 (1986)
32. Carr, D., Jog, N., Kumar, H., Teittinen, M., Ahlberg, C.: Using Interaction Object Graphs to Specify and Develop Graphical Widgets, Technical Report No. UMCP-CSD CS-TR-3344, Human-Computer Interaction Laboratory, University of Maryland [cited: 17.11.2006], <http://citeseer.ist.psu.edu/carr94using.html>
33. Bock, C.: Model-Driven HMI Development: Can Meta-CASE Tools do the Job? In: Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS 2007), Waikoloa, USA, January 3-6, p. 287b (2007)

## Questions

### ***Ann Blandford:***

*Question: How mature is this approach – are you evaluating it in context of use?*

Answer: Porsche are using it in their food chain. Porsche is now close to VAG – VAG using the tree soft tool – so they are looking at comparing approaches and finding that the techniques described are better and cheaper.

### ***Morten Borup Harning:***

*Question: You distinguish between layout and content, could you please elaborate?*

Answer: Layout is what the screen looks like, content is about the frequency of the radio, the station that is visible etc.

### ***Michael Harrison:***

*Question: How do these representations relate to the evaluation of the system?*

Answer: Prototypes can be generated at any stage in the process, either in the car or on the screen.

### ***Nick Graham:***

*Question: In practice in-car systems involve huge amounts of code, can you comment on issues of scale?*

Answer: Only 20 or so functions are frequently used – we work on the core functions before putting them in the car.

# Getting SW Engineers on Board: Task Modelling with Activity Diagrams

Jens Brüning, Anke Dittmar, Peter Forbrig, and Daniel Reichart

University of Rostock, Department of Computer Science  
Albert-Einstein-Str. 21,  
18059 Rostock, Germany  
{jens.bruening, anke.dittmar, peter.forbrig,  
daniel.reichart}@informatik.uni-rostock.de

**Abstract.** This paper argues for a transfer of knowledge and experience gained in task-based design to Software Engineering. A transformation of task models into activity diagrams as part of UML is proposed. By using familiar notations, software engineers might be encouraged to accept task modelling and to pay more attention to users and their tasks. Generally, different presentations of a model can help to increase its acceptance by various stakeholders. The presented approach allows both the visualization of task models as activity diagrams as well as task modelling with activity diagrams. Corresponding tool support is presented which includes the animation of task models. The tool itself was developed in a model-based way.

**Keywords:** HCI models and model-driven engineering, task modelling, UML.

## 1 Introduction

Model-based software development has a long tradition in HCI. Approaches like Humanoid [22], Mecano [20], or TRIDENT [4] aim to provide designers with more convenient means to describe user interfaces and to supply corresponding tool support. The main idea is to use different models for specifying different aspects which seem to be relevant in user interface design. Because of the dominant role of task models the terms model-based design and task-based design are often used interchangeably (e.g. [5], [25]). In this context, task analysis provides “an idealized, normative model of the task that any computer system should support if it is to be of any use in the given domain” [14]. In other words, it is assumed that parts of task knowledge of users can be described explicitly, for example, in terms of task decomposition, goals, task domain objects, and temporal constraints between sub-tasks. It is furthermore assumed that task models can be exploited to derive system specifications, and particularly user interface specifications, which help to develop more usable and task-oriented interactive systems (e.g. [26], [19]).

However, with the emergence of MDA [17] the model-based idea is often related to the object-oriented approach. Many supporters of MDA are not even aware of the origins. It also was recognized that some software engineers have problems to accept

the value of task modelling. This might be the case because task diagrams are not part of the Unified Modeling Language [24]. Taking into account that object-oriented techniques covering all phases of a software development cycle are currently the most successful approaches in Software Engineering it might be wise to integrate task-related techniques and tool support in order to transfer knowledge and experience from HCI to Software Engineering.

UML offers activity diagrams to describe behavioural aspects but does not prescribe how they have to be applied during the design process. They are often deployed to describe single steps of or an entire business process. We suggest to use activity diagrams for task modelling. Although a familiar notation does not guarantee that software engineers develop a deeper understanding of user tasks it might be a step in the right direction.

The paper shows how CTT-like task models can be transformed into corresponding activity diagrams. In addition, transformation rules to some extensions of CTT models are given. This approach has several advantages. First, task analysts and designers can still apply “classical” task notations but transform them into activity diagrams to communicate with software developers or other stakeholders who prefer this notation. However, it is also possible to use activity diagrams from scratch to describe tasks. Second, activity diagrams which are structured in the proposed way can be animated. We will present corresponding tool support. Third, the comparison of task models and activity diagrams enrich our understanding of the expressiveness of both formalisms. Activity diagrams, which are structured like task models, represent a subset of all possible diagrams only. However, their simple hierarchical structure might also be useful for specifying other aspects of systems under design in a more convenient way. On the other hand, elements as used in activity diagrams (e.g. object flows) might stimulate an enrichment of current task model notations.

The paper is structured as followed. Sect. 2 presents a short introduction to task modelling and to activity diagrams as well as related work. An example is introduced, which is used and extended throughout the paper. Transformation rules for CTT-like task models and their application to the example model are given in Sect. 3.1 and 3.2. In Sect. 3.3 we discuss extensions to CTT models and their transformation. Tool support for the suggested approach is discussed in Sect. 4. Model-based development ideas were used to implement tools for handling task models and their transformation into activity diagrams as well as for animating these activity diagrams by using a task-model animator which we developed earlier in our group (e.g. [10]). A summary is to be found in Sect. 5.

## 2 Background

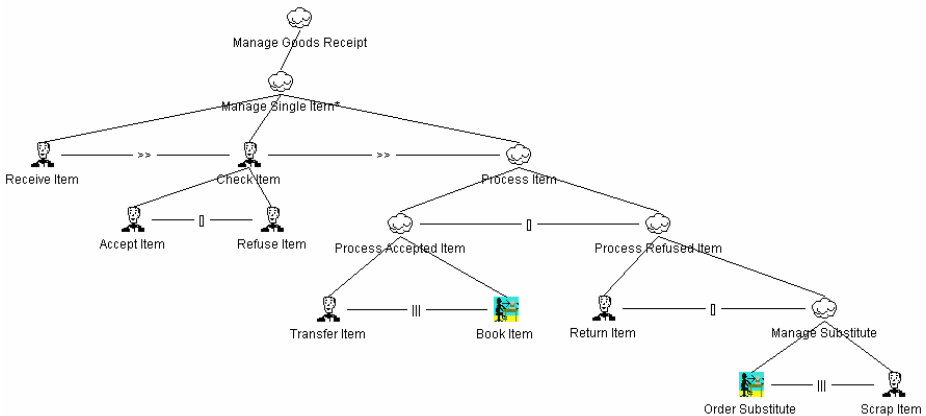
This section begins with a short overview of task modelling concepts and with an introduction of CTT as well-known task notation within the model-based design approach of interactive systems. Then, activity diagrams in UML 2.0 are introduced. After discussing related work we argue why activity diagrams seem to be a good starting point for integrating task modelling into the object-oriented design approach.

## 2.1 Task Modelling

The underlying assumption of all theories about tasks is that human beings use mental models to perform tasks. Task models - as cognitive models - are explicit descriptions of such mental structures. Nearly, if not all task analysis techniques (with HTA [1] as one of the first approaches) assume hierarchical task decomposition. In addition, behavioural representations are considered which control the execution of sub-tasks. TKS (Task Knowledge Structure) [14] is one of the first attempts to formalize the hierarchical and sequential character of tasks in order to make task models applicable to system design [26]. CTT (Concur Task Trees) [19] is the perhaps best known approach within the model-based community (HCI) today - thanks to corresponding tool support like CTTE [7]. In [16] a comparison of task models is to be found. The example model given in Fig. 1 shows the decomposition of task *Manage Goods Receipt* into the sub-tasks of receiving, checking, and processing single items and so on. In addition, temporal operators between sibling tasks serve to define temporal relations between sub-tasks. CTT supports the following operators (T1 and T2 are sub-tasks of T).

$T1 \gg T2$	enabling	$T1 [] T2$	choice
$T1     T2$	independent concurrency	$T1 [=] T2$	order independency
$T1 [> T2$	disabling/deactivation	$T1 \triangleright T2$	suspend-resume
$[T1]$	optional task	$T1^*$	iteration

Hence, a task model describes a set of possible (or rather, planned) sequences of basic tasks (leaves in the task tree) which can be executed to achieve the overall goal. For example, the model  $T = (T1 ||| T2) \gg T3$  would describe the sequences  $\langle T1, T2, T3 \rangle$  and  $\langle T2, T1, T3 \rangle$  (with  $T1, T2, T3$  as basic tasks).







**Fig. 1.** CTT model of task *Manage Goods Receipt*

Most task-based approaches do not support a formal description of the task domain, though often aware of the need for it. In Sect. 2.4, we use TaOSpec (e.g. [9]) to enrich our example task model by task-domain objects which help to describe pre-conditions and effects of sub-tasks in a formal way.

## 2.2 Activity Diagrams in UML 2.0





An activity diagram is a graph consisting of action, control, and/or object nodes, which are connected by control flows and data flows (object flows). UML 2.0 offers a variety of notations. We mention only some of them, which are used in the context of this paper. For more information we refer to [24].

*Actions* are predefined in UML and constitute the basic units of activities. Their executions represent some progress in the modelled system. It is distinguished between four types of actions.

	CallOperationAction: invokes user-defined behaviour
	CallBehaviorAction: invokes an activity
	SendSignalAction: creates an asynchronous signal
	AcceptEventAction: accept signal events generated by a SendSignalAction

An *activity* describes complex behaviour by combining actions with control and data flows. CallBehaviorActions allow a hierarchical nesting of activities.

*Control nodes* serve to describe the coordination of actions of an activity. Following node types are supported.

●	initial node		decision		fork
⦿	activity final		merge		join
⊗	flow final				

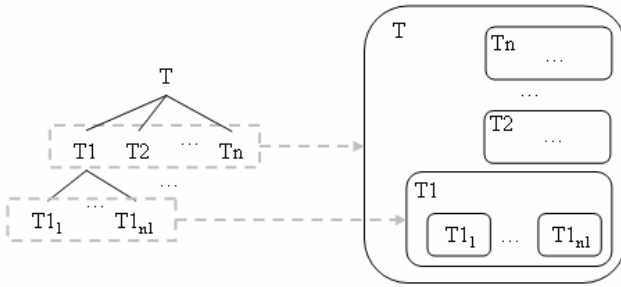
## 2.3 Related Work

The need for integrating knowledge from both the HCI field and from Software Engineering was seen by others. However, most of the current work in model-based design is concentrated on providing alternatives for task trees in CTT notation in form of structural UML diagrams (e.g. [18], [3], [2]). [15] suggests a mapping from tasks specified in a CTT model to methods in a UML class diagram. We believe that activity diagrams are a more appropriate UML formalism for integrating task-based ideas into the object-oriented methodology. They allow behavioural descriptions at different levels of abstraction. Hence, it is possible to specify task hierarchies and temporal constraints between sub-tasks in a straightforward way. Furthermore, activity diagrams are already used to specify workflows and business processes. Relations between workflows and task modelling are explored e.g. in [23], [6], and [21].

## 3 Transformation from Task Models to Activity Diagrams

In this section a transformation from task models into activity diagrams is presented. It preserves the hierarchical structure of models by deriving corresponding nested activities.





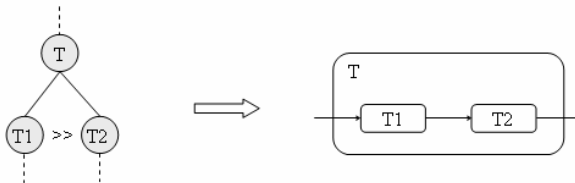
**Fig. 2.** The transformation preserves the hierarchical structure of the task model

Each level in a task hierarchy is mapped to one activity (that is called through a CallBehaviorAction in the level above) as illustrated in an abstract way in Fig. 2.

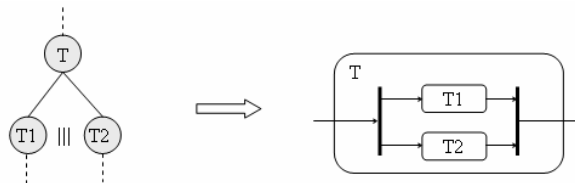
For each temporal operator mentioned in Sect. 2.1, a transformation rule is defined in the figures below. The left parts of the figures show task structures which are relevant for the transformation rules. On the right side, corresponding activity diagram fragments are given. For reasons of simplicity, the labels for CallBehaviorActions were omitted. The consideration of a node and its direct sub-nodes within a task tree is sufficient for most of the rules. The dashes above and below the nodes indicate the context. So, rules are applied to those parts of an actual task tree, which match the structures given on their left sides.

### 3.1 Transformation Rules for CTT-Like Tasks

Fig. 3 shows the enabling operator that describes that task T2 is started when task T1 was finished. In the activity diagram, this case is modelled with a simple sequence of activity T1 followed by T2.



**Fig. 3.** R1: Enabling relation as activity diagram



**Fig. 4.** R2: Concurrency relation as activity diagram

In Fig. 4, the  $\parallel$ -operator is used to express the concurrent execution of tasks T1 and T2. In a corresponding activity diagram fragment the fork node (first bar) and the join node (second bar) are applied to describe this behaviour.

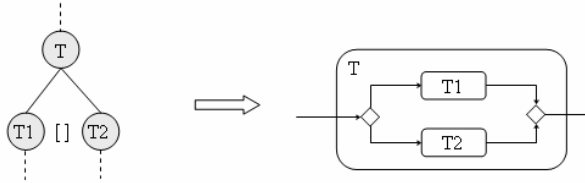


Fig. 5. R3: Choice relation as activity diagram

Fig. 5 deals with the alternative operator. So, either task T1 is allowed to be executed or task T2. In the corresponding activity diagram part, this operator is realised by using a decision and a merge node. Guards could be attached to the arrows behind the decision node to specify which of the tasks should be performed next.

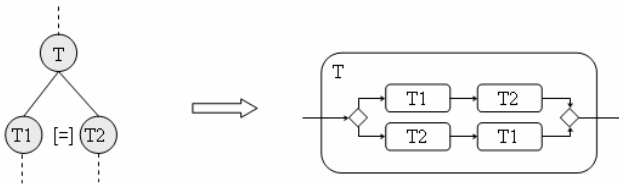


Fig. 6. R4: Order independent relation as activity diagram

The order independent operator is handled in Fig. 6. There, either task T1 and after that task T2 is executed or first T2 and then T1. In the corresponding activity diagram fragment the situation is modelled with two sequences, a decision and a merge node. It should be mentioned that a problem may arise if more than two activities are executed in an independent order because the number of possible sequences in the activity diagram is growing very fast. In such cases, the readability could be improved by using a stereotype as also proposed below for task deactivation.

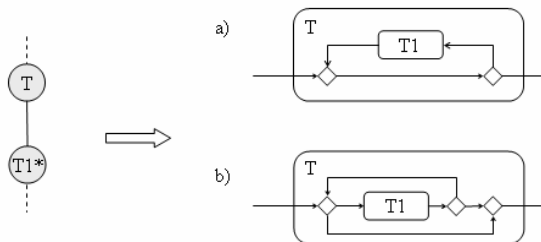
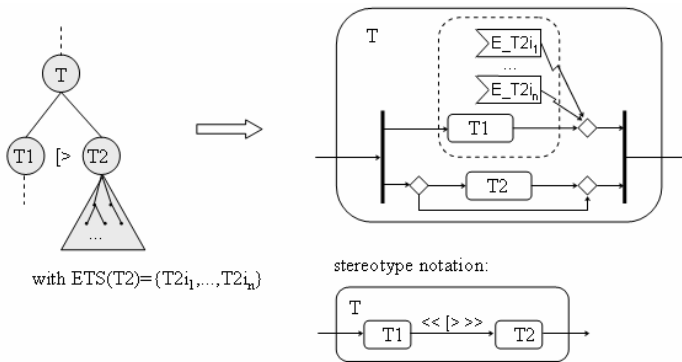


Fig. 7. R5: Iteration in task models and activity diagrams

Iteration is a special kind of temporal relation. It is not specified between two or more different tasks but can be considered as a feature of a task itself. Fig. 7 shows two ways how activity diagrams can express iteration. We prefer version b) because of better readability (particularly in cultures where people read from left to right).

In order to find a mapping from task deactivation to a proper activity diagram fragment we have to go somewhat deeper into the semantics of task models. Assume that task T2 specifies a set of sequences of basic sub-tasks (leaves in a task tree) beginning with either  $T2_{i_1}, T2_{i_2}, \dots$  or  $T2_{i_n}$ . This set is finite and is called enabled task set – ETS (e.g. [19]). Thus,  $ETS(T2) = \{T2_{i_1}, \dots, T2_{i_n}\}$ . T2 deactivates task T1 if one of the basic sub-tasks in  $ETS(T2)$  is accomplished. The execution of T1 is stopped and T is continued by performing the rest of T2.



**Fig. 8.** R6: Deactivation in task models and activity diagrams

To understand the transformation rule depicted in Fig. 8 we first need to look at Fig. 9a) where an activity diagram for modelling basic sub-tasks is presented. Basic sub-task T is mapped to a sequence of a CallOperationAction with the same name and a SendSignalAction  $E\_T$  which notifies that T was performed. The stereotype `<<complex action>>` specifies that no other action can be performed between T and E. In the diagram fragment in Fig. 8 AcceptEventActions for accepting signals sent by actions which correspond to the basic tasks in  $ETS(T2)$  are used in combination with an interruptible region (denoted as dotted box) to describe a possible deactivation of T1. However, to keep the diagrams readable we suggest to use the notation with stereotype `<< [> >>` as shown down right in Fig. 8.

A transformation for the suspense-resume operator is not proposed. It would require a kind of “history-mode” for activities as known, for example, for states in state charts.

### *Transformation of sibling sub-tasks*

Parent nodes with at most two sons are considered only in the transformation rules. However, all sibling sub-tasks of a task at the hierarchical level n have to be mapped to nodes of a corresponding activity diagram at refinement level n. Hence, a multiple application of rules at the same level of the hierarchy is necessary as indicated in

Fig. 9b) for sibling sub-tasks T1, T2, and T3 with temporal constraints  $(T1 \parallel T2) \gg (T3 \gg T1)$ . Here, rule R2 is applied to  $(T1 \parallel T2)$ , R1 to  $(T3 \gg T1)$ , and then rule R1 again to the intermediate results. Take note that a combined application of rules at the same refinement level is possible because the activity diagram fragments in all rules have exactly one incoming control flow and one outgoing control flow.

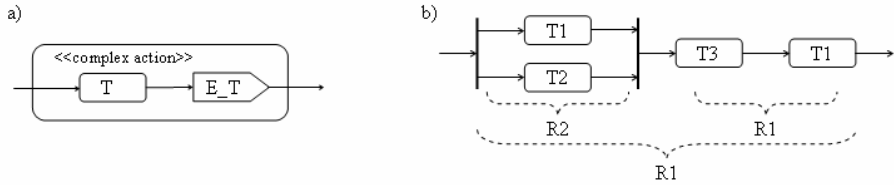


Fig. 9. a) Basic sub-task T as activity diagram, b) Transformation of sibling sub-tasks

### 3.2 Transformation of an Example Task Model

We will now transform the sample task model of Sect.2.1 into an activity diagram to show how transformations work. The task model is about managing incoming items in an abstract company. First, we need to model the begin and end node of the UML activity diagram and then the root task in between these nodes which is shown in Fig. 10. The root task is *Manage Goods Receipt* and is refined by the iterative sub-task *Manage Single Item* (rule R5).

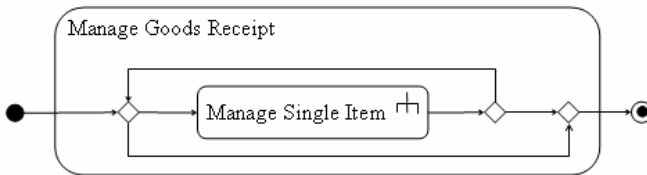


Fig. 10. The root task with an iterative sub-task, begin and end node

Sub-task *Manage Single Item* is divided into three sub-tasks which are related by the enabling operator in the task model. The double application of the transformation rule R1 in Fig. 3 results in the activity diagram shown in Fig. 11.

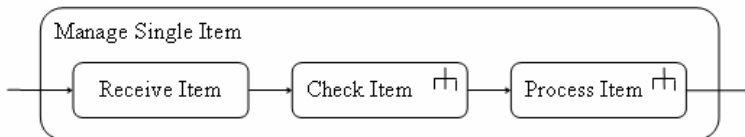
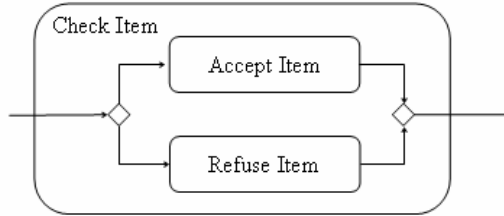
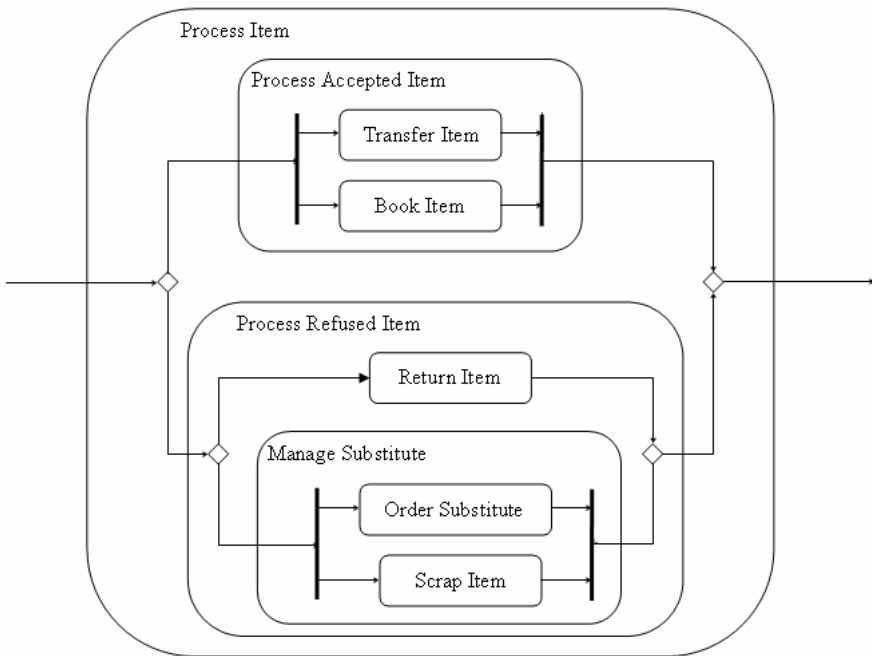


Fig. 11. Refinement of *Manage Single Item* with the enabling transformation

The task *Receive Item* is a leaf in the task model of section 2.1. Thus, a diagram as to be seen in Fig. 9a) has to be created. The refinement of activity *Check Item* is shown in Fig. 12. Sub-tasks *Accept Item* and *Refuse Item* are basic ones in the task model and have to be refined similarly *Receive Item*.



**Fig. 12.** Refinement of *Check Item* with the choice transformation



**Fig. 13.** The result of the transformations in *Process Item*

Now, only the task *Process Item* is left for continuing the transformation process and refining the activity diagram. In Fig. 13, the result of this is shown. There, two choice and concurrency operator transformations are used to get this final result for the activity *Process Item*. To get the whole activity diagram as the result of the transformation process the diagram of Fig. 11 should replace the activity of the same name in Fig. 10. The same has to be done with the diagrams of Fig. 12 and Fig. 13 in Fig. 11.

### 3.3 Handling of Extensions to CTT-Like Task Models

#### 3.3.1 Additional Temporal Operators

Our experiences in case studies raised the need for another kind of iteration. In the example, it could be more comfortable for users if they can manage a second, third or more items independent from the first item. Unlike with the normal iteration, in this kind of iteration one can start the next loop before finishing the first one. We call it *instance iteration* (denoted by  $T^\#$ ). In Fig. 14a), a first idea of a corresponding activity diagram is drawn. There, any number of activity T1 can be started parallel. Unfortunately, the dots between the activities of T1 are not allowed to be used in UML. So we had to search for another solution.

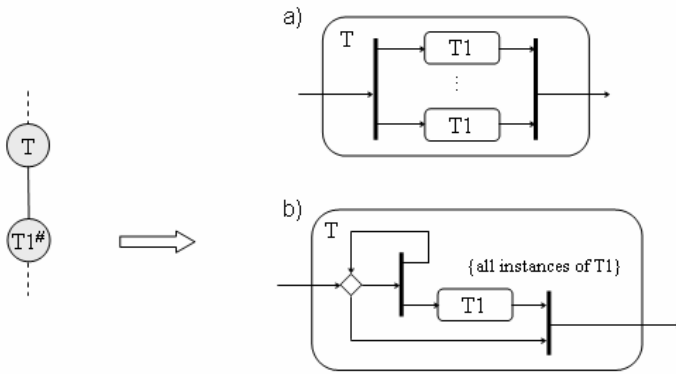
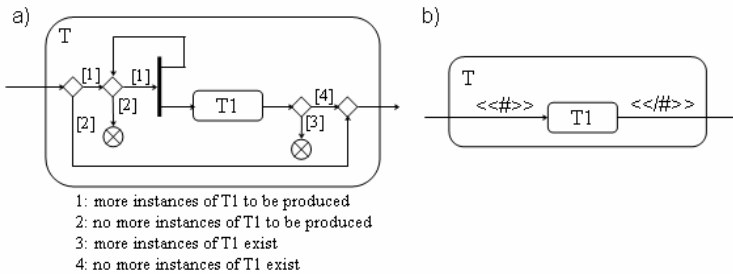


Fig. 14. Instance iteration in task models and activity diagrams

In Fig. 14b), it can be seen how instances of T1 are created. The choice/merge node creates a new instance of T1 if it chooses the arrow in the middle. Then, after the fork node has been passed activity T1 begins and at the same time the token comes back to the choice node. In the same way, any number of new instances of T1 can be created. After sufficient activities of T1 are started the choice node takes the lower arrow. Unfortunately, there is currently no way how the instances of T1 can be caught. In this figure, the functionality of waiting for all the instances finishing is modelled with the discriminator *{all Instances of T1}* associated to the join node.



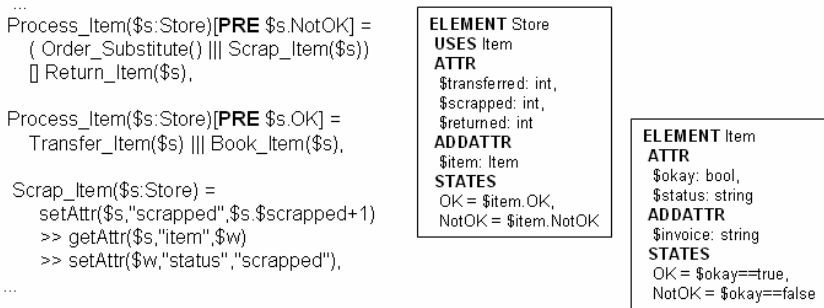
- 1: more instances of T1 to be produced
- 2: no more instances of T1 to be produced
- 3: more instances of T1 exist
- 4: no more instances of T1 exist

Fig. 15. a) An alternative diagram for Fig. 14b), b) instance iteration with stereotypes

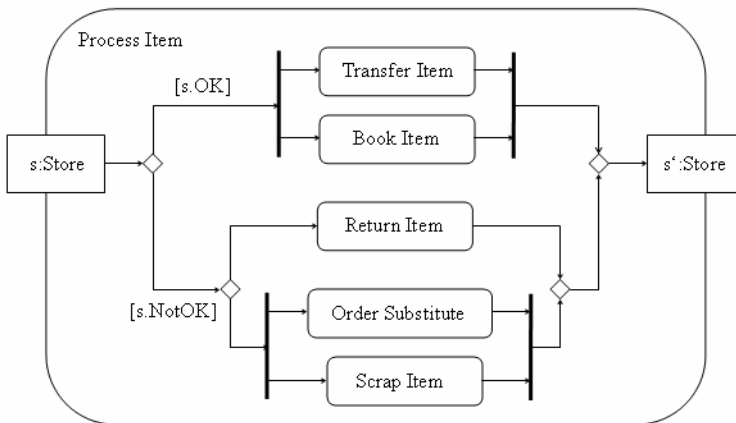
Fig. 15a) depicts an alternative activity diagram for instance iteration. In addition, we suggest stereotypes again to describe instance iteration in a convenient way. This possibility is shown in Fig. 15b). The stereotype <<#>> indicates that any numbers of the following task T1 can be created. With the other stereotype <</#>> it is said that all of the started tasks T1 must be finished before T is finished.

### 3.3.2 Task-Domain Objects and Object Flows

With TaOSpec we developed a specification formalism in our group, which allows to describe tasks as well as task-domain objects in a formal way. In addition, preconditions of sub-tasks and their effects on task-domain objects can be specified. In [8], we have shown that such a hybrid notation often leads to more concise and, possibly, to more natural descriptions than pure temporal notations (like CTT models) or pure state descriptions. In Fig. 16, a more concise TaOSpec fragment, which corresponds to sub-task *Process Item* in Fig. 1 is given (for more details on TaOSpec see e.g. [9]).



**Fig. 16.** Sub-task *Process Item* as TaOSpec model with two relevant domain objects (*Store* and *Item*). The task model is enriched by object flows.



**Fig. 17.** Activity diagram for sub-task *Process Item* enriched by object *s* (instance of *Store*)

Activity diagrams not only allow control flows but also object flows. TaOSpec elements can be mapped to objects. In addition, implicit objects flows in TaOSpec (via parameters in a task model) become explicit object flows in activity diagrams. In Fig. 17, an activity parameter is used to describe a *Store*-object. Guards reflect the pre-conditions specified in Fig. 16. In comparison to Fig. 13 this specification is clearer.

## 4 Model-Based Development of Tool Support

### 4.1 General Development Approach

After several years of individual software development we recently used the MDA approach [17]. Using the technology offered by Eclipse [11] and several related frameworks we specify our metamodels and generate main parts of our tools. In other words: we apply model-based techniques to create model-based development tools.

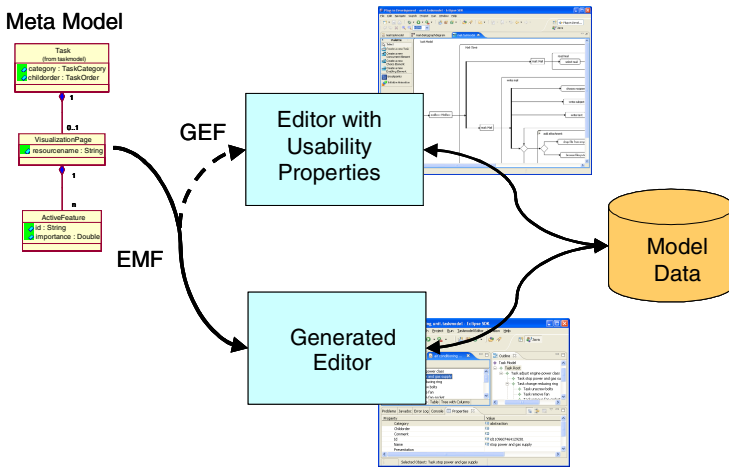


Fig. 18. General approach for model-based development

Based on a meta model and the eclipse modeling framework [12] an editor can be generated that allows the manipulation of corresponding models. In general, such generated EMF-based editors are not very user friendly. For hierarchical models it is acceptable because the model is represented in an appropriate way.

Alternatively, the graphical editing framework [13] offers a technology to develop editors that fulfil the usability requirements better. These editors can work on the same data as the generated editor. In this way, test data can be edited with the generated editor and visualised with the developed one until the full functionality of the user friendly editor is available.

Fig. 18 gives an overview of the general development process. According to the left hand side it is necessary to model the domain of the models. In our case this is a specification for task models.



It is our idea that each task has to be performed by a person or system in a specific role. A task changes the state of certain artifacts. In most cases this will be one object only. Other objects support this process. They are called tools. Both, artifacts and tools are specified in a domain model.

The general idea of hierarchical tasks and temporal relations between tasks are of course true for our models as well.

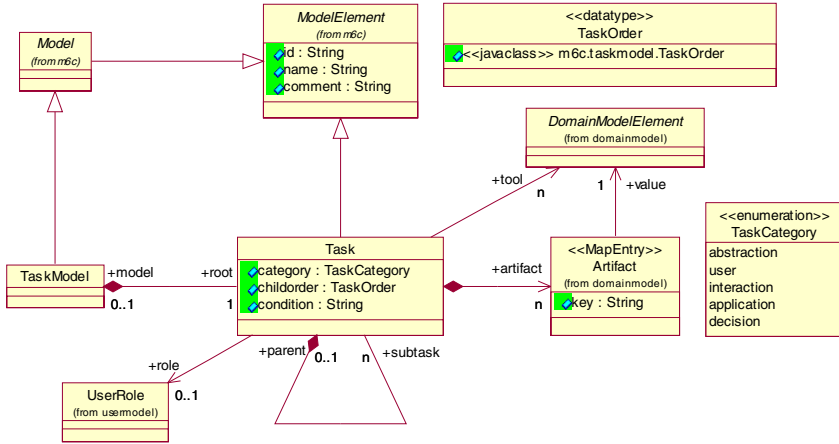


Fig. 19. Meta model for task models

Fig. 19 shows a part of the meta model for hierarchical task models. The parent subtask-relation at the bottom of the class diagram allows the specification of hierarchical task structures. A task model consists of exactly one root task and each task has either a relation to a parent task or is the root task of the model. A task has relations to a user role from a user model and artifacts and tools from a domain model.

A special class *TaskCategory* specifies as enumeration the different types of tasks as they are already known from concurrent task trees [7].

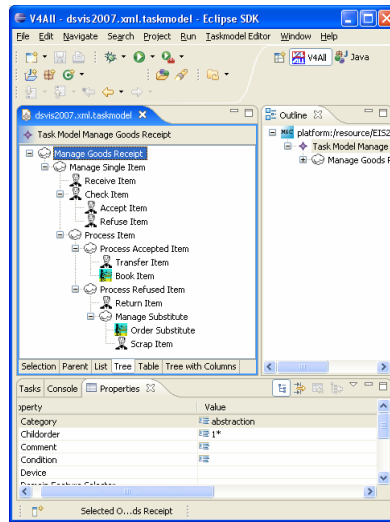
The temporal relations like enabling, order independence, concurrent, choice, suspend-resume and deactivation are represented by a class *TaskOrder*. This class allows the specification of temporal relation of sub-tasks by regular expressions. This gives a little bit more freedom in describing the dynamic behaviour of task models. It allows specifications that are difficult to visualize like ((a >> b >> c) [ ] (b >> c >> d)).

Using EMF [12] the meta model in Fig. 19 is sufficient to generate automatically software for an editor of the corresponding models. This editor of course does not fulfil a lot of usability requirements but it allows specifying models in detail. Fig. 20 gives an impression how the user interface of such a generated editor looks like.

In the central view one can see the tree of model elements, in this case the task hierarchy. New model elements can be inserted via context menu and copied or moved by drag&drop. In the bottom view every attribute of the currently selected model element is shown and can be manipulated via text or combo boxes depending on the corresponding data type in the meta model.

## 4.2 Tool Support for the Transformation from Task Models into Activity Diagrams

Based on the meta-task model and the generated software an own structured editor for activity diagrams was developed using GEF [13]. It is called structured, because it does not allow the drawing of individual nodes and connections like most editors, but allows only the insertion of complex structured model parts like sequences of “enabling tasks”, “choice elements” and “concurrent elements”. More or less only “sub-trees” are allowed to be inserted. The underlying model is the same as for the generated task-model editor. The visual representation is generated automatically from this model and an explicit model to model transformation is not necessary.



**Fig. 20.** Task editor – automatically generated from meta models

In this way a lot of mistakes regarding the creation of activity diagrams can be omitted. This consequence is already known from structured programming that is an improvement over programming with “go to”. Drawing lines can be considered as programming “go to”.

Fig. 21 gives an impression how the user interface of the structured activity diagram editor looks like. On the left hand side one can see the palette of possible operations allowed for a diagram. After selecting one operation and moving the mouse over the diagram certain interaction points will be visible signalling places, where the operation could be performed. After selecting one of these interaction points the editor asks for an integer value, which represents the number of task in a sequence, the number of choices or the number of order independent elements.

According to the answer of the user the corresponding elements are immediately inserted and the diagram is drawn again.

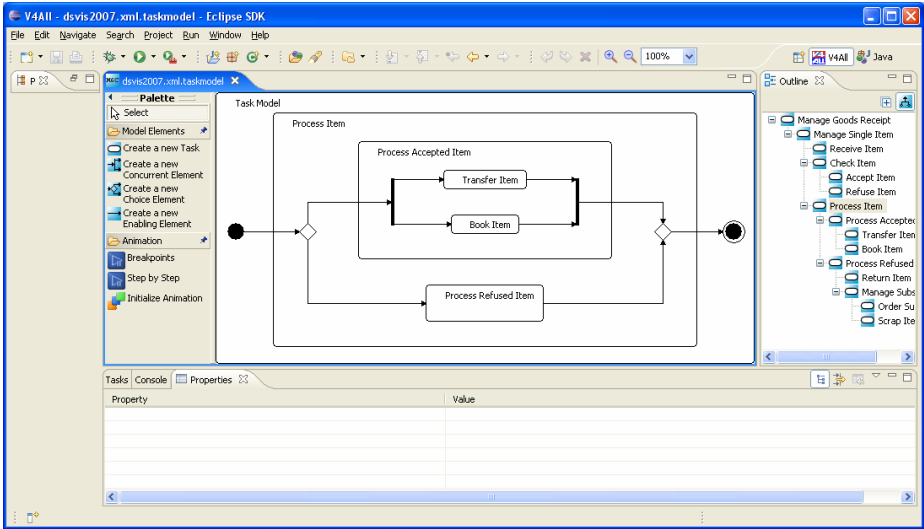


Fig. 21. Structured activity diagram editor

Interactively one can decide how many levels of detail one would like to see. It is also possible to have a look at a special lower level only. This is possible because the task hierarchy is presented as task tree as well. This is contained in the outline view on the right hand side of the user interface. By selecting a tree node the corresponding sub-diagram is shown in the main view.

GEF uses the model-view-controller pattern to ensure consistency between model and its appropriate visualisation. Here, the model is a task model matching the meta model, the view consists of geometrical figures and labels and the controller defines, which interactions are possible and how model changes are reflected in the corresponding view.

### 4.3 Animation of Task Models and Corresponding Activity Diagrams

To validate the behaviour of activity diagrams (or rather their underlying task models) an animation tool has been developed.

Fig. 22 contains an example of an animated activity diagram and the corresponding visualization of the tasks model.

At the current point of execution task *Receive Item* is already performed. It is now the decision to activate *Accept Item* or *Refuse Item*. Both are enabled. *Process Item* is disabled because *Accept Item* or *Refuse Item* has to be performed first.

Animation can be performed with activity diagrams on different levels of detail. Each task containing several subtasks can be collapsed and the user can animate just a part of a complex model. It is possible to automatically proceed the animation until a decision has to be made or to run the animation step by step. In the first mode one can additionally set breakpoints to stop the process at a specific point.

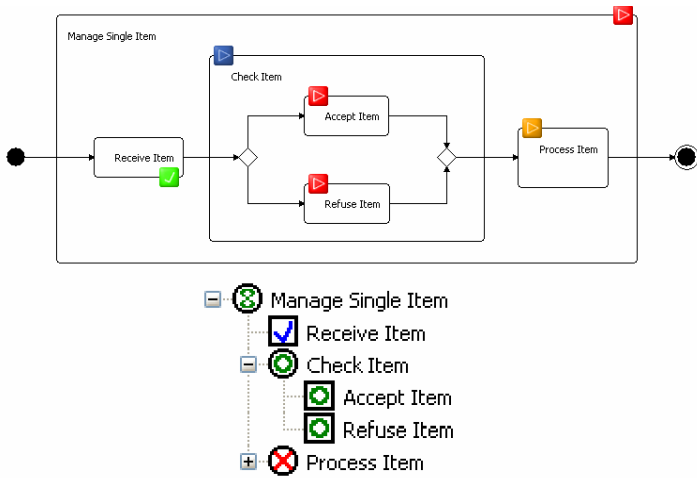


Fig. 22. Activity diagram and task model in an animated mode

## 5 Summary and Future Work

The paper discussed how task modelling concepts can be made more attractive for the software engineering community. It was proposed to present task models as activity diagrams - a notation most developers are familiar with. We suggested a “task-oriented” development of activity diagrams and we defined corresponding mapping rules. Although this restricts the variety of possible activity diagrams we believe that a more systematic methodology helps to come to more reasonable models. Temporal relations available in task models but missing in UML were also represented by stereotypes.

Tool support was suggested that allows to derive and to edit activity diagrams in this structured way. An animation of models helps to evaluate the requirements specifications and to get early feedback from users.

From our point of view structured activity diagrams could play a similar role to activity diagrams as structured programs to programs.

In the future an adequate modelling method has to be elaborated, which allows to develop workflows and task models of current and envisioned working situations as well as system models in an intertwined way. Currently, there exists no satisfying approach in both communities. Activity diagrams in UML 2.0 may be useful to integrate the object-oriented design of interactive systems and the task-based design approach.

## References

1. Annett, J., Duncan, K.D.: Task analysis and training design. *Occupational Psychology* 41 (1967)
2. Bastide, R., Basnyat, S.: Error Patterns: Systematic Investigation of Deviations in Task Models. In: Coninx, K., Luyten, K., Schneider, K.A. (eds.) TAMODIA 2006. LNCS, vol. 4385, pp. 109–121. Springer, Heidelberg (2007)
3. Van den Bergh, J.: High-Level User Interface Models for Model-Driven Design of Context-Sensitive User Interfaces. PhD thesis, Universiteit Hasselt (2006)

4. Bodart, F., Hennebert, A.-M., Leheureux, J.-M., Vanderdonckt, J.: Computer-Aided Window Identification in TRIDENT. In: INTERACT 1995. Chapman-Hall, Boca Raton (1995)
5. Bomsdorf, B.: Ein kohärenter, integrativer Modellrahmen zur aufgabenbasierten Entwicklung interaktiver Systeme. PhD thesis, Universität Paderborn (1999)
6. Bruno, A., Paternò, F., Santoro, C.: Supporting interactive workflow systems through graphical web interfaces and interactive simulators. In: Dix, A., Dittmar, A. (eds.) Proc. of TAMODIA 2005 (2005)
7. CTTE (read: 20.09.06), <http://giove.cnuce.cnr.it/ctte.html>
8. Dittmar, A., Forbrig, P.: The Influence of Improved Task Models on Dialogs. In: Proc. of CADUI 2004 (2004)
9. Dittmar, A., Forbrig, P., Heftberger, S., Stary, C.: Support for Task Modeling -A Constructive Exploration. In: Bastide, R., Palanque, P., Roth, J. (eds.) DSV-IS 2004 and EHCI 2004. LNCS, vol. 3425, pp. 59–76. Springer, Heidelberg (2005)
10. Dittmar, A., Forbrig, P., Reichart, D., Wolff, A.: Linking GUI Elements to Tasks – Supporting an Evolutionary Design Process. In: Dix, A., Dittmar, A. (eds.) Proc. of TAMODIA 2005 (2005)
11. Eclipse Homepage (read: 20.09.06), <http://www.eclipse.org/>
12. Eclipse Modeling Framework Homepage (read: 20.09.06), <http://www.eclipse.org/emf/>
13. Graphical Editing Framework Homepage (read: 21.09.06), <http://www.eclipse.org/gef/>
14. Johnson, P.: Human computer interaction: psychology, task analysis, and software engineering. McGraw-Hill Book Company, New York (1992)
15. Limbourg, Q.: Multi-Path Development of User Interfaces. PhD thesis, Université catholique de Louvain (2004)
16. Limbourg, Q., Pribeanu, C., Vanderdonckt, J.: Towards Uniformed Task Models in a Model-Based Approach. In: Johnson, C. (ed.) DSV-IS 2001. LNCS, vol. 2220, p. 164. Springer, Heidelberg (2001)
17. <http://www.omg.org/mda/> (read: 20.09.06)
18. Jardim Nunes, N., Falcão e Cunha, J.: Towards a UML profile for interaction design: the Wisdom approach. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 101–116. Springer, Heidelberg (2000)
19. Paternò, F.: Model-Based Design and Evaluation of Interactive Applications. Springer, Heidelberg (2000)
20. Puerta, A.: The MECANO Project: Comprehensive and Integrated Support for Model-Based Interface Development. In: Vanderdonckt, J. (ed.) Proc. of CADUI 1996 (1996)
21. Stavness, N., Schneider, K.A.: Supporting Workflow in User Interface Description Languages. In: Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages, AVI 2004 (2004)
22. Szekely, P., Luo, P., Neches, R.: Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In: CHI 1992 (1992)
23. Trætteberg, H.: Modelling Work: Workflow and Task Modelling. In: Proc. of CADUI 1999 (1999)
24. <http://www.uml.org> (read: 20.10.06)
25. Vanderdonckt, J., Puerta, A.: Preface -Introduction to Computer-Aided Design of User Interfaces. In: Proc. of CADUI 1999 (1999)
26. Wilson, S., Johnson, P., Markopoulou, P.: Beyond hacking: A model based design approach to user interface design. In: Proc. of BCS HCI 1993. Cambridge University Press, Cambridge (1993)

## Questions

**Morten Borup Harning:**

*Question: How can you, by simply showing how to represent task models using activity diagrams, convince developers that focusing on task modeling is important?*

Answer: Both the tool support and the extended notation makes it easier to use activity diagrams to represent task models. This is achieved using tool animation; making it easier to specify temporal issues, iteration and so on in a way that is not currently possible. Of course, the tool can be used to specify system oriented designs, but so can CTT.

**Daniel Sinnig:**

*Question: How does the tool enforce the building of activity diagrams that represent task models? Designers don't usually build task diagrams.*

Answer: The tool does not force them but it helps them to understand what is represented in the task model.

**Philippe Palanque:**

*Question: As task models are supposed to be built by Ergonomists or human factors people, why do you think software engineers would use the tool?*

Answer: There are 2 main levels: the analysis level and the design level. Our work is supposed to support the design level. The initial task model coming from analysis of work has to be done first. The idea is to support the Software engineers so that they understand the analysis of work and to exploit it during design.

**Prasun Dewan:**

*Comment: Your activity diagrams describing constraints on concurrent user actions remind me of an idea in concurrent programming called path expressions which describe constraints on procedures that can be executed by concurrent processes. To address the previous question about the usefulness of these diagrams, path expressions provide a more declarative explanation of the constraints than something low level like semaphores. It seems activity diagrams have a similar advantage.*

*Question: You might want to look at path expressions to see if you can enrich activity diagrams. Path expressions can be used to automatically enforce the constraints they describe during application execution. Can activity diagrams do the same?*

Answer: They can be used to automate animations of the constraints.

**Jan Gulliksen:**

*Question: Would you use these diagrams with end users and do you have experience with this?*

Answer: Yes, the simulation tool is aimed at that. And it is very easy for them to understand the models thanks to execution of the models.

# Considering Context and Users in Interactive Systems Analysis

José Creissac Campos<sup>1</sup> and Michael D. Harrison<sup>2</sup>

<sup>1</sup> DI/CCTC, Universidade do Minho

Campus de Gualtar, 4710-057 Braga, Portugal

Jose.Campos@di.uminho.pt

<sup>2</sup> Informatics Research Institute, Newcastle University

Newcastle upon Tyne, NE1 7RU, UK

Michael.Harrison@ncl.ac.uk

**Abstract.** Although the take-up of formal approaches to modelling and reasoning about software has been slow, there has been recent interest and facility in the use of automated reasoning techniques such as model checking [5] on increasingly complex systems. In the case of interactive systems, formal methods can be particularly useful in reasoning about systems that involve complex interactions. These techniques for the analysis of interactive systems typically focus on the device and leave the context of use undocumented. In this paper we look at models that incorporate complexity explicitly, and discuss how they can be used in a formal setting. The paper is concerned particularly with the type of analysis that can be performed with them.

**Keywords:** Interactive systems, modelling, analysis, context.

## 1 Introduction

Because usability is dependent on “specified users” [11], at the limit the usability of a device can only be assessed empirically and ‘in situ’. However, usability analysis techniques can be employed to help the designers and developers to envisage the impact of interactive systems.

Different types of usability analysis methods have been proposed over the years. They can be divided into two general classes. Empirical methods (typically performed with real users – for example, think aloud protocols and questionnaires), and analytic models (usually based on models – for example, heuristic evaluation and cognitive walkthroughs).

Usability cannot be guaranteed in an analytic way. There are simply too many factors involved to make it feasible. Nevertheless, despite some dispute about their real worth [9, 10], analytic methods are being used in practice and evidence indicates that they can play a relevant role in detecting potential usability problems from the outset of design [6].

Performing usability analysis of interactive systems design is a multi-faceted problem. This means that no single analysis method can cover all aspects of usability. For example, Cognitive Walkthrough [12] focuses on how the device supports the users’

work, while Heuristic Evaluation [13] focuses on generic/universal properties of the device. Different methods will be needed at different stages of design and for different tasks.

One specific type of analytic approach is the use of formal (mathematically rigorous) methods of modelling and reasoning. Although take up of formal approaches to modelling and reasoning about software has been slow, recent years have seen an increased interest in the use of automated reasoning techniques such as model checking [5] for the analysis of complex systems. In the case of interactive systems, formal methods can be particularly useful in reasoning about systems with complex interactions. Examples include the analysis of the internal mode structure of devices [4, 8] and the analysis of the menu structures of interactive applications [19].

Consider, for example, performing a Cognitive Walkthrough of a user interface with a complex mode structure. It will be very difficult, if not impossible, to guarantee that all possible systems response will have been considered during the analysis. With model checking, although we cannot achieve the same level of reasoning about cognitive psychology aspects of the interaction, we are able to test properties over all possible behaviours of the system.

The problem with all these techniques is that they focus on the device, occasionally (as in the case of Cognitive Walkthrough) a representation of the user's task, but never on an explicit representation of the context in which the device and user are embedded. Although in practice the analyst or team of analysts brings this contextual understanding to the table, as devices become more dependent on context the need to make assumptions explicit about context becomes more important. This problem becomes more pressing as we move towards ubiquitous computing where device action uses context explicitly, including details like location, user preferences and previous activity.

In this paper we look at the modelling of interactive systems in a formal setting, and what type of analysis can be performed with them. In particular, we look at how we can consider context in interactive systems modelling and analysis from a formal (mathematically rigorous) standpoint. The contribution of the paper is to develop a separable model of context that supports clarity of assumptions in the analysis of the device.

The structure of the paper is as follows. Section 2 discusses the relevance of user and context considerations in the modelling and analysis of interactive systems. Section 3 addresses modelling of devices. Section 4 addresses modelling of assumptions about user behaviour as restrictions on the behaviour of the device. Section 4 addresses the impact of context in the analysis. Section 5 reflects on what has been presented in the paper. Section 6 concludes with some final considerations.

## 2 Devices and Users in Context

According to the ISO 9241-11 standard, usability can be defined as "The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [11]. Analysing this definition, we can see that the factors that have an impact on the usability of a system when trying to achieve a given goal are the actual product (or device) being used, the users using the device to achieve the goal, and the context of the interactive system.



From now on we will use the terms *interactive device* (or simply *device*) and *user(s)* to refer to the interactive product being designed/analysed, and to the human(s) using it, respectively. The term *interactive system* will be used to refer to the combination of both (device and users).

Traditionally, analytic approaches to usability analysis have placed particular emphasis on the device and/or user. So, for example, in heuristic evaluation a team of experts checks a model/prototype against a list of desirable features of interactive devices. It is assumed that the experts will identify appropriate usage considerations. Cognitive walkthroughs attempt to determine if/how a device will support its users in achieving specified goals, from a model of the device. The approach is rooted in the CE+ theory of exploratory learning [16], and, in some ways, this means it overprescribes the assumptions that are made about how the user will behave. In PUMA [1] the model of a (rational) user is built to analyze what the user must know to successfully interact with the device. Again, this means that the assumptions about user behaviour are quite strong. In [4] a model of the device is analysed against all possible user behaviour. Instead of prescribing, from the outset, assumptions about how users will behave, these assumptions are derived during the analysis process. Hence, assumptions about the user are identified that are needed to guarantee specified properties of the overall interactive system.

In summary, context has not been given particular attention, being usually only implicitly considered. Taking account of context is important because it has an effect on the way device actions are interpreted. A key problem associated with ubiquitous systems is that confusions arise because actions are interpreted through implicit assumptions about context. This problem is effectively the mode problem that model checking techniques are particularly well suited to addressing.

Additionally, considerations about the user tend to be either too vague (c.f. Heuristic Evaluation) or overprescribed and therefore in danger of not capturing all relevant behaviours (c.f. Cognitive Walkthroughs or PUMA) – these techniques might overlook workarounds for example. While these approaches can be useful, problems arise when we consider complex systems. This happens because it becomes difficult to identify informally all the assumptions that are being made about (or, more importantly, are relevant to) the user behaviour, and/or because a very prescriptive model of user behaviour might rule out unexpected behaviours that are potentially interesting from an analysis point of view.

As stated above, in this paper we are specifically interested in (formal) analytic approaches. We are particularly interested in seeing how we can build on the work developed in [4, 2] to take into consideration models/assumptions about the users and the context of usage of the systems.

In order to make the discussion more concrete, we will be using as a basis an example described in [4] (but considerably reworked here due to our new focus). We need to be clear about what we mean by context. So we want to discuss the issues associated with context using a very simple example. Rather than look at a ubiquitous system we re-consider the analysis of a mode control panel (MCP). This is a safety critical interactive system that has been analysed using a number of techniques [14]. The important thing about this example is that the context in which the device is embedded is crucial to an understanding of the interactive behaviour of the system. The techniques that are developed here are as important in intelligent and mobile systems

where action inference (qua mode) is based on preferences, or location, or history or other elements that can be described as context. The example addresses the design of the Mode Control Panel (MCP) of an MD-88 aircraft (see figure 1), and was developed using MAL interactors [4, 2].

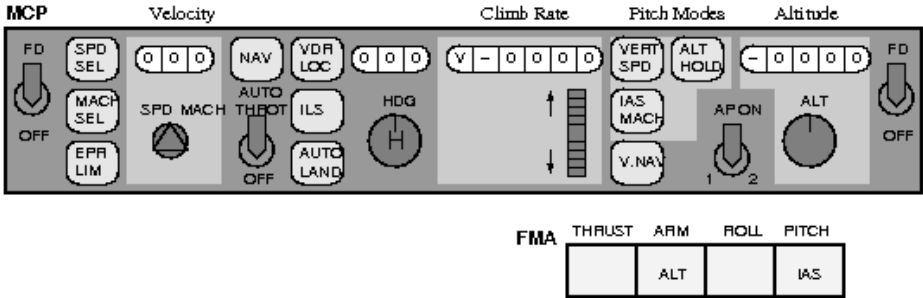


Fig. 1. The MCP panel (areas with lighter background will be modelled)

### 3 Device Model (or, Devices in Context)

Building a behavioural model of the device enables analysis of all the behaviours that are possible to achieve goals. Whether or not these behaviours are cognitively plausible, however, is sometimes left outside the formal analysis process. This is due to the difficulty in adequately formalising the users' cognitive process. This aspect will be further explored in section 2.2. For now we will concentrate on the device model.

#### 3.1 Modelling

In the approach put forward in [4, 2] only the device is modelled explicitly. In the MCP example, the device is the actual MCP. Using MAL interactors we can perform a first modelling approach<sup>1</sup>:

```

interactor MCP
  includes
    dial(ClimbRate) via crDial
    dial(Velocity) via asDial
    dial(Altitude) via ALTDial
  attributes
    [vis] pitchMode: PitchModes
    [vis] ALT: boolean
  actions
    [vis] enterVS enterIAS enterAH enterAC
    toggleALT
  axioms
    [asDial.set(t)] action'=enterIAS

```

<sup>1</sup> For brevity the definitions of some named expressions are not presented here. It is expected that the names used will be self-explanatory. The full model is available at <http://www.di.uminho.pt/ivy/index.php?downloads>

```

[crDial.set(t)] action'=enterVS
[ALTDial.set(t)] ensure_ALT_is_set
[enterVS] pitchMode'=VERT_SPD & ALT'=ALT
[enterIAS] pitchMode'=IAS & ALT'=ALT
[enterAH] pitchMode'=ALT_HLD & ALT'=ALT
[toggleALT] pitchMode'=pitchMode & ALT'=!ALT
[enterAC] pitchMode'=ALT_CAP & !ALT'

```

For a description of the MAL interactors language the reader is directed to [2]. Here the focus is not so much on the particular language being used but in what is being expressed. We will provide enough detail about the models to make their meaning clear. The main point about the language is to know that axioms are written in Modal Action Logic [18].

Returning to the model above, it includes the three dials of interest identified in figure 1, as well as attributes to model the pitch mode and the altitude capture switch (ALT). The pitch mode defines how the MCP influences the aircraft:

- VERT\_SPD (vertical speed pitch mode) – instructs the aircraft to maintain the climb rate set in the MCP;
- IAS (indicated air speed pitch mode) – instructs the aircraft to maintain the velocity set in the MCP;
- ALT\_HLD (altitude hold pitch mode) – instructs the aircraft to maintain the current altitude;
- ALT\_CAP (altitude capture pitch mode) – internal mode used to perform a smooth transition from VERT\_SPD or IAS to ALT\_HLD.

The altitude capture switch, when armed, causes the aircraft to stop climbing when the altitude indicated in the MCP is reached. The available actions are related to selecting the different pitch modes, and setting the values in the dials.

This particular model, however, is of limited interest from a behavioural analysis point of view since it does not consider the semantics of the controlled process. In fact only the logic of the user interface has been modelled. In principle, this can enable us to analyse what are the possible behaviours in the interface. In this case, however, in order for the MCP to have realistic behaviour, we must include in the model information about the process that the MCP is controlling and its constraints (i.e., its context of execution). At the minimum we need to know what the possible responses (behaviours) of the process are. Without that we will not be able to analyse the joint behaviour of device and user (the interactive system).

In this case, the context is a very simple model of the aircraft and its position in airspace:

```

interactor airplane
  attributes
    altitude: Altitude
    climbRate: ClimbRate
    airSpeed: Velocity
    thrust: Thrust
  actions
    fly
  axioms

```

```

# Process behaviour
[fly] (altitude'>=altitude-1 & altitude'<=altitude+1)
      & (altitude'<altitude -> climbRate'<0)
      & (altitude'=altitude -> climbRate'=0)
      & (altitude'>altitude -> climbRate'>0)
      & (airSpeed'>=airSpeed-1 & airSpeed'<=airSpeed+1)
      & (airSpeed'<airSpeed -> thrust'<0)
      & (airSpeed'=airSpeed -> thrust'=0)
      & (airSpeed'>airSpeed -> thrust'>0)
# not enough airspeed means the plane falls/stalls
(airSpeed<minSafeVelocity & altitude>0)->climbRate<0

```

This description is bound to the device model through a number of declarations as described below. Firstly, we must bind the two models architecturally. We do this by inclusion in the MCP of:

```

includes
  airplane via plane

```

Secondly, creating a behavioural binding requires that the following axioms must be included in the MCP:

```

per(enterAC) -> (ALT & nearAltitude)
(ALT & pitchMode!=ALT_CAP & nearAltitude)
      -> obl(enterAC)
pitchMode=VERT_SPD -> plane.climbRate=crDial.needle
pitchMode=IAS -> plane.airSpeed=asDial.needle
pitchMode=ALT_HLD -> plane.climbRate=0
pitchMode=ALT_CAP -> plane.climbRate=1
(pitchMode=ALT_CAP & plane.altitude=ALTDial.needle)
      -> obl(enterAH)

```

What these axioms state is how the process and the device are related. The first two axioms state that action `enterAC` must be performed when the ALT capture is armed and the aircraft is near enough the target altitude, and that only in those conditions can it be performed. The next four axioms state how the different pitch modes in the device affect the process. The last axiom states that action `enterAH` must happen when the target altitude is finally reached.

### 3.2 Analysis

We can now start testing the device. We will be focussing on detecting potential problems with one of the main functions of the MCP: controlling the altitude acquisition procedure. A reasonable assumption is to consider that, whenever the altitude capture is armed, the aircraft will reach the desired altitude (that is, the altitude set in ALT-Dial). This assumption can be expressed as:

```

AG((plane.altitude!=ALTDial.needle & ALT)
  ->
  AF(pitchMode=ALT_HLD
     & plane.altitude=ALTDial.needle))

```

What the formula expresses is that whenever the plane is not at the altitude set in the ALTDial, and the ALT capture is armed, then eventually the plane will be at the desired altitude and the pitch mode will be altitude hold (ALT\_HLD).

A modelling and verification environment (IVY) that is under development<sup>2</sup> has facilitated the analysis of these models using the SMV model checker [5]<sup>3</sup>. With the help of the IVY tool, it is possible to determine that the property above does not hold. The counterexample, produced by NuSMV, shows that the pilot can simply toggle the altitude capture off (see figure 2)<sup>4</sup>.

We can conclude that, in order to guarantee the property, we must at least assume a user that will not toggle the altitude capture off. This is a reasonable expectation on the user behaviour which can be expressed without adding to the model by changing the property to consider only those behaviours where the pilot does not disarm the altitude capture:

```
AG((plane.altitude!=ALTDial.needle & ALT)
  ->
  AF((pitchMode=ALT_HLD
      & plane.altitude=ALTDial.needle)
     | action=toggleALT))
```

Now, either the plane reaches the desired altitude/pitch mode or the altitude capture is turned off.

This new formulation of the property still does not hold. The counterexample now shows a pilot that keeps adjusting vertical speed. Clearly this is a possible but, in the current context, unlikely behaviour. Once again we need to redefine the property in order to consider only those behaviours where this situation does not happen. There is a limit to the extent to which this process can continue because:

- the property to prove is made opaque through more and more assumptions about the user;
- there are assumptions that can become very hard to encode this way;
- there is no clear separation between the property that was proved and the assumptions that were needed.

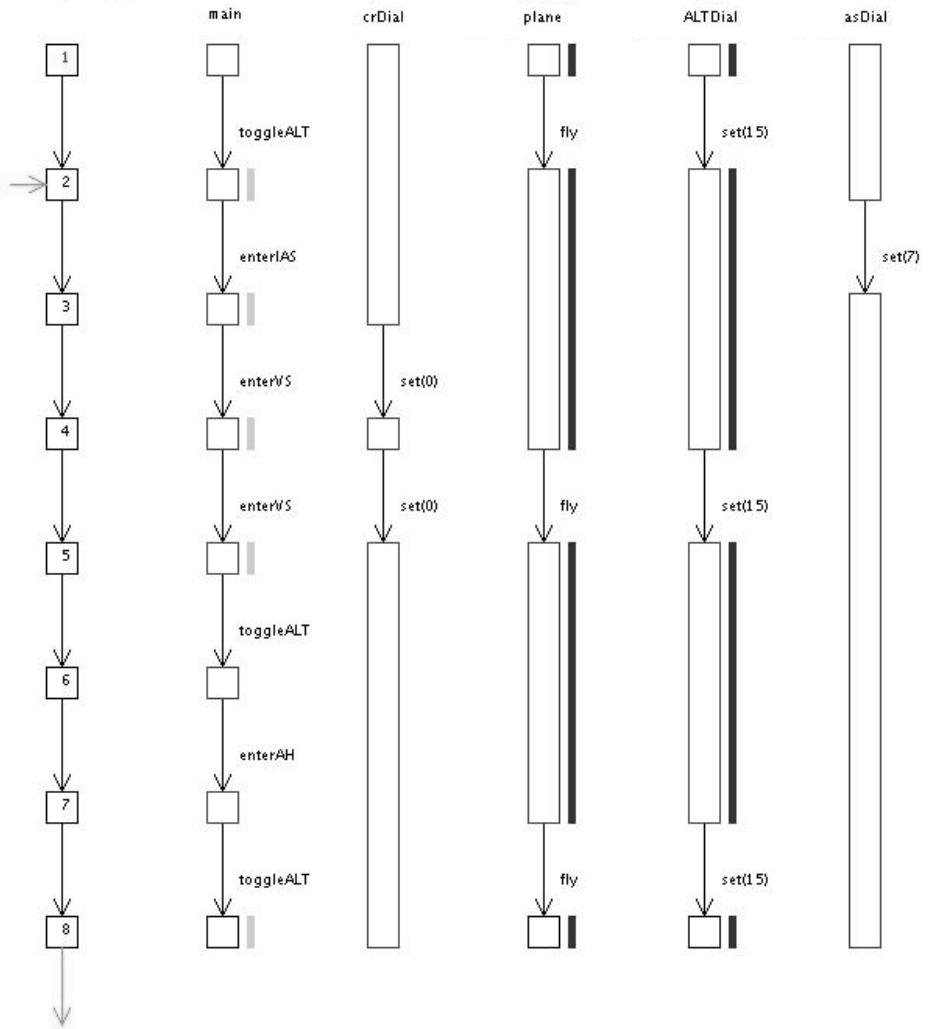
To avoid these problems, we will now explore encoding the assumptions about user behaviour as constraints on the possible user behaviours. Remember that up to now we were considering all possible behaviours that the device supported, regardless of their cognitive plausibility. This new approach will be dealt with in the next section.

---

<sup>2</sup> See <http://www.di.uminho.pt/ivy>

<sup>3</sup> To be precise, two versions of SMV are currently being maintained and developed: Cadence SMV, by Cadence labs, and NuSMV. In the current context are using NuSMV.

<sup>4</sup> We present here a graphical representation of the traces produced by NuSMV. This representation is shown at the level of abstraction of the MAL interactors model (hence the presence of actions associated with state transitions). Each column represents the behaviour of a single interactor (except for the first column which acts as a global index to the states produced by the model checker). States (represented by rectangles) can be annotated with information on their attributes (not in this particular case) and/or markers identifying specific state properties. Transitions are labeled with the action that triggers them. The trace representations in this paper have been produced by the trace visualizer component of the IVY tool.



**Fig. 2.** Counter example for the first property (the dark coloured lines identify states where  $plane.altitude < ALTDial.needle$ ; the light coloured lines identify states where the ALT capture is armed)

### 4 On User and Other User Related Models

Several authors have proposed the use of different types of models to address the issue of considering users during formal verification of interactive systems. Two examples are the work on Programmable User Modelling Analysis (PUMA) [1], and work by Rushby [17]. In the case of PUMA, the objective is to model a rational user. As already explained, this can become too prescriptive, considering that we want to explore unexpected interactions.

In the case of Rushby's work, assumptions about how the users will behave are encoded in the device model from the outset. The danger here is that no clear separation between the device and user assumptions is enforced by the modelling approach. Hence assumptions might be made that go unnoticed during the analysis.

We adopt an approach similar to the latter except for a significant difference. We do not create the model (make assumptions about user behaviour) beforehand. Instead, we obtain the user model as a by-product of the verification process, identifying the assumptions that are needed for the interactive system to verify the property or properties under consideration. This means that even when the property is finally verified, an analysis must be performed of the needed assumptions in order to see if they are acceptable. This way, the results are less prone to tainting by hidden assumptions made about the users' behaviour during the modelling process.

## 4.1 Modelling

We will now consider a user model that constrains the pilot not to behave as described in the previous section. The approach to encoding assumptions about user behaviour is to strengthen the pre-conditions on the actions the user might execute.

The only danger in doing this is that the action whose pre-conditions are being strengthened can also be used by the device itself. In that case the axioms would restrict not only user behaviour, but also the device's behaviour. This problem can be avoided by defining distinct user-side, and device-side actions with the same semantics, but different modality annotations.

For example, in the case of the `toggleALT` action we would be defining two replacement actions:

- `toggleALT_user` – action for the user to toggle the altitude capture on and off;
- `toggleALT_dev` – action for the device to toggle the altitude capture on and off.

The first would be marked as user selectable, while the second would not. Alternatively we could use a parameter in `toggleALT` to specify whether the actions were being caused by the user or by the device, and strengthen the axioms for the user only. In this case, however, using different modalities would not be possible since we would only have one action.

In the current case `toggleALT` is only performed by the users so we do not need to make the above distinction.

We start by setting up the user interactor. It simply creates a binding (by inclusion to the MCP model):

```
interactor user
  includes
    MCP via ui
```

Next we introduce the assumptions as restrictions on user behaviour. Since we want to model restrictions, the axioms take the form of permission axioms over the action of the user:

- Assumption n. 1 – the pilot will not toggle the altitude capture off. The axiom states that the altitude toggle action is only permitted when the altitude capture is off. This restricts the behaviours of interest to those where the user never switches the altitude capture off. Note that this does not interfere with the internal behaviour of the device. The device uses the enterAC action to switch the capture off when approaching the target altitude.

```
per(ui.toggleALT) -> !ui.ALT
```

- Assumption n. 2 – the pilot will be wise enough not to set inappropriate climb rates. The three following axioms state that, when the altitude capture is armed, the user will only set climb rates that are appropriate for the goal at hand (negative if the aircraft is above the target altitude; positive if the aircraft is below the target altitude; and zero when the aircraft is at the target altitude).

```
per(ui.crDial.set(-1)) ->
  (!ui.ALT | ui.plane.altitude>ui.ALTDial.needle)
per(ui.crDial.set(0)) ->
  (!ui.ALT | ui.plane.altitude=ui.ALTDial.needle)
per(ui.crDial.set(1)) ->
  (!ui.ALT | ui.plane.altitude<ui.ALTDial.needle)
```

Our model is now three tiered. At the core there is the context in which the device is embedded and in which the interaction takes place, in this case the aircraft itself. Then there is the device (the MCP). Finally at the top level there is a model of user assumptions.

## 4.2 Analysis

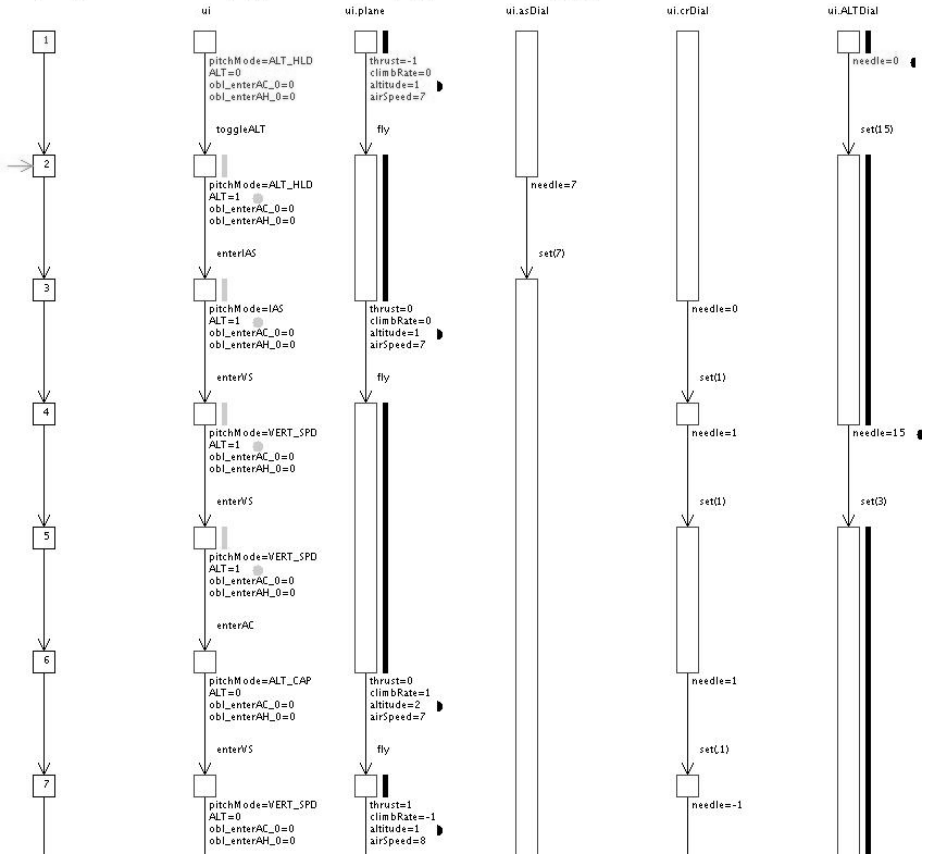
We can now test the system under these two user assumptions. Considering the user model, the property becomes:

```
AG((ui.plane.altitude!=ui.ALTDial.needle & ui.ALT)
  -> AF(ui.pitchMode=ALT_HLD
    & ui.plane.altitude= ui.ALTDial.needle))
```

In the context of these two assumptions the property still does not hold. This time the counter example points out that, during the intermediate ALT\_CAP pitch mode, changes to the vertical speed will cause a change in pitch mode when the altitude capture is no longer armed. This behaviour effectively ‘kills the altitude capture’: the aircraft will be flying in VERT\_SPD pitch mode with the altitude capture disarmed (see state 7 in figure 3).

We could keep adding constraints to the behaviour of the user, and we would find out that the only possibility to prove the property is to consider that the user does not make changes to the values set in the MCP while the plane is in ALT\_CAP mode. This seems an unreasonable assumption, and in fact instances of this problem have been reported to the Aviation Safety Report System (ASRS) [14].





**Fig. 3.** Partial view of the counter-example for the model with user assumptions (from state 3 to state 4 the action `set(1)` in `crDial` causes no problem, from 6 to state 7 the altitude capture is no longer armed and the `ALT_CAP` pitch mode is lost)

## 5 Impact of Context in the Analysis

In reality, there is a problem with the analysis above. We are referring directly to `plane.altitude` at the user level in the second assumption which is an attribute of the aircraft, not an attribute of the device. On the face of it axioms in the user model should only refer to attributes of the interactive device annotated with an appropriate modality. The problem is that in our model there is no information about current altitude being provided through the device that mediates the context to the user.

There are two possible solutions to this:

- If we are designing the device we might consider including the needed information on the display.
- If we are analysing an existing device (as is the case), or designing it as part of a larger system, we must analyse whether the information is

already present in some other part of the system, not included in the current model, and consider how to represent this in the model.

Of course the results of the analysis are completely dependent on the quality of the model used. However, developing separate models for the different levels of analysis involved helps in identifying potential flaws in the models.

In any case, we can also explore the use of contextual information, and whether the needed information is present in the environment.

### 5.1 Context

Context is understood as the characteristics of the environment that have a bearing on the interactive system (see figure 4). The system (S) is to be understood as the combination of device (D) and user(s) (U). The device is formed by the application’s functional core (L) and its user interface (I). Analysing the context can be relevant at a number of levels:

- We might want to analyse whether including some piece of information in the device is really needed – if the information is clearly present in the context of use then including it in the device might create unnecessary user interface clutter.
- We might want to analyse a situation of (partial) system failure, and whether the user will be able to overcome it by resorting to contextual information.
- We might be interested in identifying problems related to different perceptions being obtained from the information gathered through the context and its representation in the device’s user interface.
- We might also be interested in the effect that (changes in) the context of usage might have on interaction with the device. It is not the same to use a system under high or low workload conditions. For example, under high workload conditions it is unlikely that the pilot will be able to adequately process the information about vertical speed obtained from the environment.

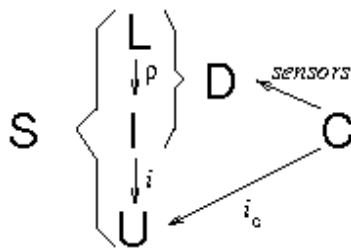


Fig. 4. Context

Context is present at many levels: physical environment, user capability and preferences and so on. Different levels “see” context differently – these may be thought of as interpretation functions (probably partial functions because the levels do not necessarily consider the same subsets of the context, and do not necessarily interpret it in

the same way). These different interpretations of context can be used to express how information about context is processed at different levels.

## 5.2 Context in the MCP

Returning to the MCP, the altitude of the plane is part of the context of the MCP (device). In this case, we can say (assuming a 'large' aircraft) that the pilot has no (or little) context regarding altitude or velocity. He may have information about vertical speed (derived from the aircraft's tilt and thrust). However it is likely that the user perception of this context information is quite low and can be discarded except for extreme circumstances. However, in those extreme circumstances the workload inside the aircraft's cockpit will probably be high. Hence, it is unlikely that the pilot will be able to gain accurate context information. In that case, unless the device provides information on the altitude, the axioms for the first set of assumptions on section 3 cannot be accepted as they have been written.

Even if we consider that contextual information about the altitude is available (because we are talking about a small aircraft), we still have to analyse what information is available. There is the problem of the definition of the information that is perceived by the pilot. It is unlikely that the pilots will be able to compare the altitude displayed in the MCP with their perception of the altitude of the aircraft. It is necessary to be cautious about what should and should not be part of the context of the user (and how) because this will have a strong impact on the quality of the analysis.

All things considered, it is conservative to assume that the user will not be able to gain accurate enough information regarding altitude from the context of use to be able to compare it with the value set in the ALTDial dial. This means that we must find a way to reformulate assumption number two. As the situation stands, even considering a user that does not use the MCP while in ALT\_CAP mode is not enough to conclude that the system is predictable regarding altitude acquisition.

We could simply assume that the pilot would not change the climb rate whenever the altitude capture is armed (or even consider that the MCP would not allow it to happen). These constraints, however, are clearly too strong. The alternative then would be to expand the interface to include information about the current altitude of the aircraft.

We note that while in this case the analysis of contextual information on the user side meant that not enough information was available to users, due to the specific conditions inside a cockpit, in mobile and ubiquitous environments contextual information will most probably play a more relevant role. In this type of system action inference (qua mode) is based on preferences, or location, or history or other elements that can be described as context.

## 6 Discussion

As stated in section 2, we chose to introduce the issues associated with context by means of a simple example. This was done so that we could be clear about the different concepts involved. This section reflects on what was learnt, and discusses the relevance of context in a larger setting.

## 6.1 Relevance of Context

Figure 4 identifies different aspects that must be considered when analysing an interactive system. The setting of the Activity to be carried out by the system is critical to this analysis. Typical approaches to the analysis of interactive systems that address the interaction between user and interface might or might not take the Activity into consideration (for example, a task model), and might or might not take the Logic of the device into consideration (depending on the modelling detail). What we have argued is that Context is also a relevant factor in this analysis process.

In our example, the aircraft was the context for the MCP and was both being influenced by the MCP, and influencing its behaviour. Hence, context will interact with the device: it can both influence the device's behaviour and be influenced by it.

More importantly, the context will also influence the user. Not only what the user knows (as was discussed in relation to the MCP), but even the user's goals, and how he or she tries to achieve them. Hence, context will also influence the activities the system supports.

## 6.2 Different Models/Different Analysis

The analysis of the MCP was introduced as a means of illustrating the ideas being put forward regarding both the need to take into account context when performing analysis of interactive system models, and the possibility of deriving information about needed assumptions over user behaviour from that same analysis. It has illustrated a particular style of analysis based on behavioural aspects of the system, specifically related to the mode structure of the device.

Besides mode related issues we can also think of analysing the menu structure of a device, or its support for specific user tasks. Using an approach based on a number of different models, each relating to a specific type of analysis means that it becomes easier to take into consideration different combinations of these factors. For example, we could add to our model a user task model and analyse whether the device, with the given user assumptions, supported that specific task in a given context.

Another (non-mutually exclusive) possibility is to consider the analysis of representational issues of the interface. In fact, it is not sufficient to say that some piece of information is available at the user interface, it is also necessary to consider if the representation being used to present the information is adequate.

Again, the notion of context becomes relevant. In [7] a model of user beliefs about the device's state is analysed against a model of the actual device's state. The objective of that analysis was to assess the quality of the user interface with respect to how it conveyed information about the device. In a contextually rich setting, however, the user will be exposed to more stimuli than those provided by the device, and unless the context of use is considered, the correspondence between the model of user beliefs and reality will be limited.

## 6.3 Information Resources

Focussing on context not only helps make analysis more accurate by more thoroughly identifying what information users have available, it also raises new issues. Task models might take contextual information into consideration to express how users will

adapt to different situations. It becomes relevant to consider how context changes the beliefs the user has about the device, but also how the device conveys information about the context, and whether the information the user receives via the device, and the information the user receives directly are consistent.

The goal of this focus on context is to identify relevant information that the user needs to successfully interact with the system. In the example we were mainly interested in understanding whether the user would have enough information to keep the climb rate of the aircraft at an appropriate level. However, we could also consider what information was needed for the user to take specific actions. For example, if instead of being automatic, the transition to the ALT\_CAP pitch mode was to be performed by the pilot, we could be interested in analysing whether enough information was being provided so that the pilot could make the decision to activate that pitch mode at (and only at) the appropriate time.

This information can come from the device or from the context of use. In [3] an approach is discussed that uses the notion of (information) resources to facilitate the analysis of whether enough information is provided to inform user actions. The resources considered therein related to the device only. The approach can easily be extended to consider contextual information, and to include not only resources for action but also resources as a means of supporting the definition of user assumptions. Hence the notion of information resource can act as a unifying approach that helps in considering all types of information available to the user in the same framework.

## 7 Conclusion

Several authors have looked at the applicability of automated reasoning tools to interactive systems analysis and their usability characteristics. Approaches such as Paternò's [15] or Thimbleby's [19] have focused heavily on the device. They have shown that it is possible to reason about characteristics of the dialog supported by the device. For example, in [19] it is shown how a formal analysis of the menu structure of a mobile phone could contribute to a simpler and faster dialogue.

When analysing an interactive device, we must take into consideration the characteristics of its users to avoid analysing behaviours that are irrelevant from a cognitive perspective, or consider design that, although ideal according to some formal criterion, are not cognitively adequate. When building a formal model we are necessarily restricting the domain of analysis, and in that process relevant aspects might be left out of the model. This is particularly relevant of interactive systems, where cognitive aspects are important but difficult to capture. Taking the user into consideration during the analysis helps in reducing that effect.

Approaches aimed at building complex architectures that attempt to model the user cognitive processes are clearly inadequate from a verification standpoint. In PUMA [1], a more contained approach is attempted: modelling the behaviour of a rational user. Even so, the authors agree that creating models suitable for automated reasoning is a time consuming process. It should also be noted that the analysis is then performed against those behaviours that are considered rational only. An alternative is to consider, not a model of the user but a model of the work. In [3] information derived from the task model for the device is used to drive the analysis. This enables analysis

of whether the device supports the intended tasks, but restricts the analysis to those behaviours that are considered in the task model.

A more flexible approach is to consider assumptions of user behaviour instead of a full blown model of user behaviour or work. These assumptions act as snippets of user behaviour that are found relevant for the analysis in question. Two approaches that follow this approach are work by Campos and Harrison [4] and by Rushby [17]. In the first case assumptions are derived from the analysis process (i.e., nothing is assumed to start with) and the analysis drives which assumptions are needed in order to guarantee some property. The assumptions are encoded into the property under verification. In second approach, assumptions are encoded into the model from the outset. That is, during model development.

The advantage of producing a separate model of context is that (1) it separates the description of the device from those concerns that influence the use of the device (2) it makes clear the contextual assumptions that are being made that can be used as part of the rationale for the design. Issues of context will become more important with the trend towards ambient systems where user context (for example location, task, history, preferences) may be used by the system to infer what action the user should make.

The example given here hints at many of these issues. This paper sets forth an agenda for more explicit specifications of context that can provide basic assumptions for rationale for the design of implicit action and its analysis.

**Acknowledgments.** This work was carried out in the context of the IVY project, supported by FCT (the Portuguese Foundation for Science and Technology) and FEDER (the European Regional Development Fund) under contract POSC/EIA/26646/2004.

## References

1. Butterworth, R., Blandford, A., Duke, D., Young, R.M.: Formal user models and methods for reasoning about interactive behaviour. In: Siddiqi, J., Roast, C. (eds.) *Formal Aspects of the Human-Computer Interaction*, pp. 176–192. SHU Press (1998)
2. Campos, J.C.: *Automated Deduction and Usability Reasoning*. DPhil thesis, Department of Computer Science, University of York (September 1999)
3. Campos, J.C., Doherty, G.J.: Supporting resource-based analysis of task information needs. In: Gilroy, S.W., Harrison, M.D. (eds.) *DSV-IS 2005*. LNCS, vol. 3941, pp. 188–200. Springer, Heidelberg (2006)
4. Campos, J.C., Harrison, M.D.: Model Checking Interactor Specifications. *Automated Software Engineering* 8(3), 275–310 (2001)
5. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: *Model Checking*. MIT Press, Cambridge (1999)
6. Desurvire, H.W., Kondziela, J.M., Atwood, M.E.: What is gained and lost when using evaluation methods other than empirical testing. In: Monk, A., Diaper, D., Harrison, M.D. (eds.) *People and Computers VII — Proceedings of HCI 1992*. British Computer Society Conference Series, pp. 89–102. Cambridge University Press, Cambridge (1992)
7. Doherty, G.J., Campos, J.C., Harrison, M.D.: Representational Reasoning and Verification. *Formal Aspects of Computing* 12(4), 260–277 (2000)
8. Gow, J., Thimbleby, H., Cairns, P.: Automatic Critiques of Interface Modes. In: Gilroy, S.W., Harrison, M.D. (eds.) *DSV-IS 2005*. LNCS, vol. 3941, pp. 201–212. Springer, Heidelberg (2006)

9. Gray, W., Salzman, M.: Damaged merchandise? A review of experiments that compare usability evaluation methods. *Human Computer Interaction* 13(3), 203–261 (1998)
10. Hartson, H.R., Andre, T.S., Williges, R.C.: Criteria for Evaluating Usability Evaluation Methods. *International Journal of Human-Computer Interaction* 1(15), 145–181 (2003)
11. ISO: International Standard ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on Usability, International Organization for Standardisation, Geneva (1998)
12. Lewis, C., Polson, P., Wharton, C., Rieman, J.: Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In: *CHI 1990 Proceedings*, pp. 235–242. ACM Press, New York (1990)
13. Nielsen, J., Molich, R.: Heuristic evaluation of user interfaces. In: *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 249–256. ACM Press, New York (1990)
14. Palmer, E.: Oops, it didn't arm – a case study of two automation surprises. In: Jensen, R.S., Rakovan, L.A. (eds.) *Proceedings of the 8th International Symposium on Aviation Psychology*, pp. 227–232. Ohio State University (1995)
15. Paternò, F.D.: A Method for Formal Specification and Verification of Interactive Systems. D.Phil thesis, Department of Computer Science, University of York (1996)
16. Polson, P., Lewis, C.: Theory-Based Design for Easily Learned Interfaces. *Human-Computer Interaction* 5, 191–220 (1990)
17. Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and Systems Safety* 75(2), 167–177 (2002)
18. Ryan, M., Fiadeiro, J., Maibaum, T.: Sharing actions and attributes in modal action logic. In: Ito, T., Meyer, A.R. (eds.) *TACS 1991*. LNCS, vol. 526, pp. 569–593. Springer, Heidelberg (1991)
19. Thimbleby, H.: User Interface Design with Matrix Algebra. *ACM Transactions on Computer-Human Interaction* 11(2), 181–236 (2004)

# XSED – XML-Based Description of Status–Event Components and Systems

Alan Dix<sup>1</sup>, Jair Leite<sup>2</sup>, and Adrian Friday<sup>1</sup>

<sup>1</sup> Computing Department, Lancaster University, Infolab21,  
South Drive, LA1 4WA, UK

<sup>2</sup> Departamento de Informática e Matemática Aplicada,  
Universidade Federal do Rio Grande do Norte, Lagoa Nova,  
59078-970, Natal, RN, Brazil

alan@hcibook.com, jair@dimap.ufrn.br, adrian@comp.lancs.ac.uk  
<http://www.hcibook.com/alan/papers/EIS-DSVIS-XSED-2007/>

**Abstract.** Most user interfaces and ubiquitous systems are built around event-based paradigms. Previous work has argued that interfaces, especially those heavily depending on context or continuous data from sensors, should also give attention to status phenomena – that is continuously available signals and state. Focusing on both status and event phenomena has advantages in terms of adequacy of description and efficiency of execution. This paper describes a collection of XML-based specification notations (called XSED) for describing, implementing and optimising systems that take account of this dual status–event nature of the real world. These notations cover individual components, system configuration, and separated temporal annotations. Our work also presents a implementation to generate Status-Event Components that can run in a stand-alone test environment. They can also be wrapped into a Java Bean to interoperate with other software infrastructure, particularly the ECT platform.

**Keywords:** Status–event analysis, reflective dialogue notation, ubiquitous computing infrastructure, XML, temporal properties.

## 1 Introduction

This paper describes a collection of XML-based specification notations for describing, implementing and optimising status–event based systems. The notations are collectively called XSED (*pron. exceed*) – XML Status–Event Description.

User interfaces are nearly universally programmed using an event-based paradigm. This undoubtedly matches the underlying computational mechanism and is thus necessarily the way the low-level implementation deals with execution. However, this event-based paradigm is also evident in the way in which interfaces are described at a higher-level and this is more problematic. This purely event-oriented view of interfaces has been critiqued for a number of years and status–event analysis proposes a view of interaction that treats status phenomena (those that have some form of persistent value) on an equal footing with event phenomena [1,2]. For example, when



dragging a window the window location and mouse location are both status phenomena and the relationship between them should be described in terms of a continuous relationship over time.

Arguably the status-oriented interactions in a traditional GUI (principally mouse dragging and freehand drawing) are to a large extent the exception to the rule of more event-focused interaction (button click, key press). However, in ubiquitous or pervasive environment the reverse is often the case. Sensors tend to monitor status phenomena such as temperature, pressure, sound level. At a low level these sensor values are translated into discrete data samples at particular moments, but unlike the moment of a mouse click, the particular times of the samples are not special times, merely convenient ones to report at. So at a descriptive level it is inappropriate to regard them as ‘events’ even if they are implemented as such at low-level. Furthermore the data rates may be very high, perhaps thousands of samples per second, so that at a practical level not taking into account their nature as status phenomena can be critical for internal resource usage and external performance and behaviour.

In this paper we will examine some of these issues and present a collection of XML-based specification notations, XSED, for embedding full status–event processing within event architectures used in ubiquitous and mobile computing. Notations are included for individual components (encoded as an extension to the W3C XML finite state machine specification), configuration of multiple components and annotations of specifications for runtime optimisation. The notations together allow local and global analysis and run-time introspection. The specifications are transformed into Java code for execution and can be wrapped as Java Bean components for execution within the ECT infrastructure [3].

## 2 Status–Event Analysis

### 2.1 What Is It?

Status–event analysis is concerned with the issues that arise when you take into account the distinction between status and event phenomena. The distinction is quite simple:

**events** – things that *happen* at a particular moment: mouse click, alarm clock rings, thunder clap

**status** – things that *are* or in other words always have some value that could be sampled: screen contents, mouse location, temperature, current time or weather

Note that the word ‘status’ is used rather than ‘state’ because of the connotations of internal state in computer systems. Whilst this internal state is an example of a status, status also includes things like the temperature, patterns of reflected light, average walking speed of a crowd.

Note too that status phenomena may be continuous (temperature) or discrete (is the light on) – the critical thing is their temporal continuity. Figure 1 demonstrates this. Status phenomenon labelled (1) has a continuously varying value over time, but the status phenomenon (2) has a number of discrete values, but still at any moment has a well defined value (except possibly at moments of transition). In contrast the event phenomena (3) and (4) occur only at specific times. The two event phenomena (3) and (4) are also shown to demonstrate that event phenomena may be periodic (3) or irregular (4).

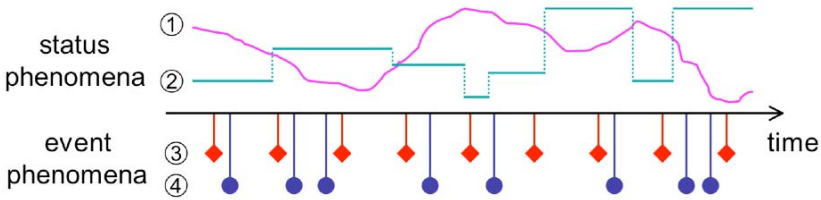


Fig. 1. Status and event phenomena over time

Status–event analysis has proved a useful way to look at interactive systems because it is able to describe phenomena that occur in human–computer interactions, in human–human interactions, in human interactions with the natural world, and in internal computational processes. For example, one way in which an active agent can discover when a status phenomena has changed is to poll it; this occurs internally in a computer, but also at a human level when you glance at your watch periodically in order not to miss an appointment. As an internal computational paradigm it has also been used in commercial development (see section 2.3 below).

Status–event analysis draws its analytic power not just from establishing distinctions, but also from the inter-relationships between status and event phenomena, like the polling behaviour above. Clearly events may change status phenomena (e.g. turning on a light) and a change in status may be noticed by an agent and become a *status change event*. Events may give rise to other events as in event-based systems, but also a status may depend on one or more other status, for example the window location tracking the mouse location – a *status–status mapping*.

## 2.2 Does It Matter?

Whilst there are clearly real distinctions between classes of phenomena, is it important that these are reflected in specification and implementation of systems? In fact there are a number of reasons why it is important to explicitly encode both status and event phenomena in specifications.

**Purity and Capture.** The first reason is just that it is right! This is not just a matter of theoretical purity, but of practical importance. The most costly mistakes in any development process are those made at requirements capture. Describing things in the way in which they naturally are is more likely to lead to correctly formulated requirements. Whilst later more formal manipulations (whether by hand or automated such as compilers) can safely manipulate this, but the initial human capture wants to be as natural as possible. For example, compare a status-oriented description: "Record a meeting when Alison and Brian are both in the room"; with an event-oriented one: "Record a meeting when there has been an 'Alison Enters' event followed by events not including 'Alison Leaves' followed by 'Brian Enters' OR a 'Brian Enters' followed by events not including 'Brian Leaves' followed by 'Alison Enters'". Which is easier to understand and more likely to be expressed correctly?

**Premature Commitment.** Even if one captures a required behaviour correctly the conversion of status–status mappings to event behaviours usually involves some form of 'how' description of the order in which lower level events interact in order to give higher

level behaviour. That is a form of premature commitment. An example of this occurred recently in the development of a visualisation system. A slider controlled a visualisation parameter. This was encoded by making each slider change event alter the underlying data structures, creating a storm of data updates and screen repaints – (prematurely committed) event-based specification of what is really a status–status mapping.

**Performance and Correctness.** We have seen how the lack of explicit status–status mappings can lead to interaction failure! If the system ‘knew’ the relationship between slider value and visualisation appearance, it could infer that updates to the internal data structures are only required when a repaint is about to happen. In general, this lack of explicit status knowledge can lead to both local computational resource problems and also excessive network load in distributed systems. In a sensor-rich system with high-data-rate sensors this is critical. Typically this is managed on an ad hoc basis by throttling sensors based on assumed required feed rate. However, this does not allow for dynamic intervention by the infrastructure if the system does not behave in the desired fashion, for example, to modify the rate of event sources.

An example of this occurred in ‘Can you see me now’ a ubiquitous/mobile game [4]. The location of each player was shown on a small map, but during play the locations of players lagged further and further behind their actual locations. The reason for this turned out to be that the GPS sensors were returning data faster than it was being processed. The resulting queue of unprocessed events grew during the game! Clearly what was wanted was not that for every GPS reading there was a corresponding change on the map (an event relationship), but instead that the location on the map continually reflected, as nearly as possible, the current GPS location (a status–status mapping).

The phrase “as nearly as possible” above is important as any status–status mapping inevitably has delays, which mean it is rarely completely accurate. For simple mappings this simply means small lags between different status–status phenomena. However, if different status phenomena have different lags then incorrect inferences can be made about their relationships [5]. For example, on a hot summer day if the house gets warmer but sensors in the hotter part of the house have longer lags than the cooler parts, then a climate control system may set fans to channel air the wrong way. In such cases explicit encoding of the status–status mapping would not remove the inherent problem of sensing delays, but would make the interdependency apparent and allow setting of parameters such as expected/required jitter between sources, forms of generalised ‘debounce’ etc.

### 2.3 Existing Status–Event Systems/Notations

A notation was used in [2] for status–event analysis, in particular allowing the specification of interstitial behaviours of interfaces – the status–status relationships that occur in-between major events, which are often what gives to an interface its sense of dynamic feel. This was targeted purely at specification and theoretical analysis although is potentially not far from an executable form. Some other specification notations, whilst not based on status–event analysis, embody aspects of status phenomena. Wüthrich made use of cybernetic systems theory with equations close to those in continuous mathematics to describe hybrid event/status systems [6] and different forms of hybrid Petri Nets have been used by different authors [7,8]. Also there is a whole sub-field of formal methods dedicated to hybrid systems specification although principally focused

on mixing continuous external real world behaviour with discrete computational behaviour [9].

Further theoretical work on event propagation in mixed status–event systems showed the importance of a strong distinction between data flow direction and initiative [10]. Often status–status mappings are better represented at a lower level by demand-driven rather than data-driven event propagation. The Qbit component infrastructure in the commercial onCue system were constructed to enable this flexibility [11]. Each Qbit may have ‘nodes’ (like Java Bean properties) of various kinds. Of the unidirectional nodes, there are familiar *get* and *set* nodes where a value is output or input under external control, *listen* nodes where a value can be output to a listener under internal control and finally *supply* nodes that allow the Qbit to request a value from an unknown external source. The last, that naturally completes the space of (single directional) node types, is particularly important as it allows demand-driven data flows with external connections.

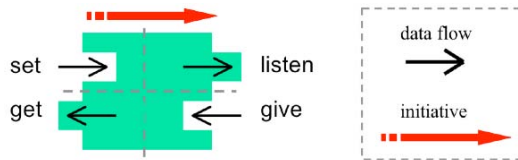


Fig. 2. Qbit nodes

Note however, that the Qbit component still does not represent status phenomena explicitly; instead it aims to make event representations easier and more flexible. The onCue product it supported was a form of context-aware internet toolbar, so shared many features with more physical sensor-based systems. Another critical feature of the Qbit framework that we have preserved in XSED, is *external binding* – because of the symmetry of the Qbit I/O model it is possible to wire up Qbits without 'telling' the Qbit what it is connected to. In comparison the listener model used for Java Beans requires each Bean to 'remember' what has asked to be told about changes.

### 3 The XSED Notation

In order to better represent both kinds of phenomena for ubiquitous interactions we have defined and implemented an executable description notation XSED that explicitly encodes status and event phenomena. The description notation XSED includes four key elements:

1. Individual software components – descriptions that include both status and event input and output, and specification of the mappings between input and output
2. Configuration – showing, without reference to specific components, how several components fit together architecturally to make a larger component
3. Binding – filling the component 'holes' in a configuration description with specific components
4. Annotation – additional timing and other information referring to the configuration links that can improve the performance of a system.

The separation of components allows a level of reuse and 'plug and play' between components. For example, an infra-red motion sensor may be used by both a burglar alarm system and to adjust the lighting when people leave a room. The separation of binding from configuration allows the same flexibility for the system as a whole: the alarm could use ultrasound instead of infra-red motion detection. The separate annotation allows varying levels of designer or automated analysis to inform optimisations of the system. For example, a temperature sensor may be able to deliver thousands of readings a second, but we may only require the temperature once per second. The knowledge of the appropriate sensor rate depends on the particular set of components, their configuration and particular external constraints in and information (e.g. the maximum rate at which temperatures change in the environment). This information does not belong in individual components, nor the configuration, nor the binding. In addition, separating performance-oriented annotation allows a level of plug-and-play for analytic tools as it gives a way for a static or dynamic analysis of a system to be fed into its (re)construction.

The concrete syntax is in XML for structural description with embedded JavaScript for computational elements. While this choice can be debated it follows successful XML-based notations such as XUL [12] and could easily be generated as intermediate form by other forms of graphical or textual notation. The XML notation can be used for design-time analysis, executed through an interpreter directly, or transformed into Java for integration with other software.

## 4 Individual Components

The individual components in XSED may represent individual UI widgets, sensors interfaces, or more computational processing. The notation follows [2] in declaring explicit status input and output as well as event input and output. One aim is to make a description that is easy for a reflective infrastructure to analyse hence we have chosen initially to encode the internal state of each component as a finite state machine rather than to have arbitrary variables in the state as in [2]. Other aspects of the notation (configuration, binding and annotation) do not depend on this decision. So it is independent, with limited repercussions. We can therefore revisit this decision later if the capabilities of the FSM are too restrictive, but it initially allows easier analysis. The use of a FSM also parallels the early Cambridge Event architecture to allow comparison between solely event-based and status–event descriptions. For concrete syntax we use XML extending the W3C draft standard for finite state machines [13].

### 4.1 XML Specification

Figure 3 show the top level syntax of a single Status-Event component (see also web listings 1 and 2). A single SE component has initial input and output declarations each of which may be a status or event. This may be followed by default status–status mappings giving output status in terms of input status. The states also contain status–status mappings (the defaults mean that these can be incomplete otherwise every output status must be given a value for every state). Finally the transitions describe the effects of events in each state. Each transition has a single input event that triggers the transition and a condition. As well as causing a change of state may also cause output events to fire.

```

xmlspec ::= input output defaults state* transition*
input ::= ( status | event )*
output ::= ( status | event )*
defaults ::= status-out*
state ::= id {start} status-out*
transition ::= state-ids event-in {condition} event-out*

```

**Fig. 3.** Overall structure of XSED specification

The status–status mappings (status-out) and event outputs have values given by expressions and the transitions conditional is a boolean expression. These can only access appropriate input status/events. In the case of output status, only the input status can be used as there are no events active. In the case of transition conditions and event outputs the value (if there is one) of the triggering event can also be used in the expressions.

## 4.2 Transforming and Executing the Specification

The XML specification is parsed into an internal Java structure. This follows a four stage process:

1. *Parsing* – the XML specification is read into an internal DOM structure using standard XML parsing
2. *Marshalling* – the XML DOM is transformed into a bespoke internal structure that still represents the specification, but does so in dedicated terms specialised methods etc. Note that this level of representation would be shared with any alternative concrete syntax. This level of representation is also suitable for static analysis.
3. *Building* – the specification is used to construct data structures and Java code suitable for execution. The components generated can optionally be wrapped as a Java Bean suitable for embedding in other execution environments, notably ECT [3]. Note, some elements of the specification are retained at runtime in the component schema to allow runtime reflection used during configuration linkage.
4. *Running* – the generated Java code is compiled and placed in suitable folders, Jar files, etc. for deployment either in EQUIP or in stand-alone test environment.

The component generated from the XML description during the *build* phase implements a generic interface that is neutral as to the precise notation used. This is so that we can experiment with different kinds of status–event ‘savvy’ notations such as augmented process algebras. Figure 4 shows this interface. It provides methods to get the value of a specific named status of a status–event component and to fire a named event with a value. It is also possible to get the names and types (schema) of input and output status and events. The last method sets the environment to which the component interact with. Note that the `getStatus` method is for the *output* status and the `fireEvent` method is for *input* events.

```

public interface SEComponent {
    Object getStatus(String name);
    void fireEvent(String name, Object value);
    public Schema getSchema();
    public void setEnvironment(SEEnvironment environment);
}

```

**Fig. 4.** Abstract status–event component

The remaining two methods are (i) a reflection method `getSchema` that retrieves the names, types etc. of the inputs and outputs to allow dynamic binding and (ii) `setEnvironment` that gives the component a handle into the environment in which it operates. This Java environment object acts as a form of single callback where the component goes when it needs to access status input or to fire an event output.

The Java interface for `setEnvironment` is shown in Figure 5. As is evident this is exactly the same as the basic part of a component. That is for most purposes a component and the environment in which it acts are identical. This is not just an accident of the Java implementation but reflects the underlying semantics – one of the strengths of status– event descriptions is that it offers a symmetric description of interactions across a component boundary.

```

public interface SEEnvironment {
    Object getStatus(String name);
    void fireEvent(String name, Object value);
}

```

**Fig. 5.** Abstract status–event environment component

## 5 Configuration

As noted we separate out the configuration of components, what links to what, from the individual component specifications. This allows components to be reused in different contexts. Furthermore, while this configuration will typically be designed with particular components in mind, it is defined only in terms of schemas of expected components. This means the configuration can also be reused with different variants of components, or with a different set of components with similar interrelationships.

At an abstract level the configuration specification is similar to the architectural specifications in Abowd et al [14], that is a collection of named component slots with typed input/output nodes and linkage between them. However, Abowd et al., in common with most configuration notations, do not fully abstract over the components and instead frame the architecture specification over specific components. In addition, the typing on our components includes their status/event nature.

In the XML configuration file, each component lists its inputs and outputs each with a status/event tag and type (see web listing 3 for an example). A `<links>` section specifies the connections between the components. Static checking verifies whether output nodes always connect to appropriately types input nodes (including their status/event nature). In addition, static checking verifies that every status input has exactly one incoming link, whilst other forms of input/output node can have one, several or no connections.

The other unusual aspect of the configuration is the way in which it is packaged as a component in its own right. Instead of specifying inputs and outputs of the configuration as a whole, a 'world' component is added within the configuration. This looks exactly like all other components and represents the external environment of the set of components. This takes advantage of the fact, noted previously, that the status–event semantics are symmetric with respect to the environment – the environment of a component looks similar to the component as the component looks to the environment. When the configuration is bound with specific components it then becomes a single component that can be placed elsewhere. The interface of this aggregate component is precisely the dual of the world component – inputs to the aggregate component are effectively outputs of the environment and vice versa.

## 6 Binding

The actual binding of configuration to components is currently entirely within code at runtime. In order to link the SE components in a configuration, small proxy environments are produced for each component linked into the configuration. When a component request an input status, it asks its environment proxy, which then looks up the relevant source status in the internal representation of the component linkage. The relevant output status and component (linked to the requested input status) is then obtained. Similarly when an output event is fired this is passed to the proxy environment, which then finds the relevant input events on other components and fires these.

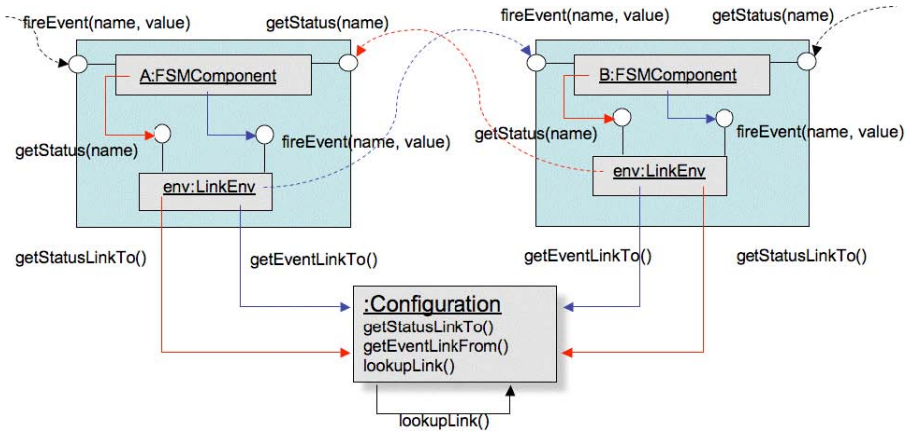


Fig. 6. Linking components

The link to the outside environment is produced using a special pair of dual SE components with the 'world' schema. These have no internal computation but simply reflect the inputs of one as the outputs of the other and vice versa. One end of the dual is bound into the configuration as the 'world' component and, once there, functions precisely like all other components within the configuration including having its own proxy environment. The other end of the dual pair then becomes the external view of the



aggregate component built from the configured components. Its inputs and outputs are then effectively the inputs and outputs of the configuration as a whole. This use of the dual radically simplifies the semantics of component aggregation.

## 7 Annotation

In the annotations description, unique link identifiers refer to links in a configuration file and specify properties of those links that can then be used to optimise the runtime behaviour of the system (see also web listing 4 ). The annotations include:

*Initiative* – Whether status links should be demand driven or data driven. Demand-driven status is the default behaviour where status inputs are requested when used. In contrast data-driven status is more similar to event propagation, where changes in status are pushed through the system. The latter is preferred if changes of status are rare compared to accesses.

*Time* – The timeliness of a link. A value of 1000 (milliseconds) means that data used can be up to 1 second 'out of date'. For a status link this would mean that if a status input is re-requested within 1 second of the last access the previous accessed value can be used. For an event link this means that multiple events within 1 second can be buffered and passed on together. For distributed system this can reduce overheads. This is similar to techniques used previously in the GtK (Getting-to-Know) notification server [15].

*Last-or-all* – When a sequence of events are fired whether all of them are important or just the last. In fact, when it is just the last, this is normally a sign that the event is a status change event. When this is combined with a timeliness annotation then multiple events within the specified time window can be suppressed and only the last one passed on.

*Synchronisation* – The timeliness annotations can mean that events and status change are not passed on in the same temporal order as they are produced. A synchronisation annotation over a collection of links specifies that order must be preserved over those links. For example, if an event is fired on one of the synchronised links then up-to-date status must be obtained for each of the synchronised status links no matter whether there is pre-accessed status of acceptable timeliness.

Note that these annotations may change the semantics as well as performance of the system. The production of the annotations, whether by a human designer or an automated global analysis, must ensure that this change in low-level semantics either does not change the higher-level semantics, or 'does not matter'. Such choices are always made in systems involving status phenomena as sampling rates are chosen depending on sensor, network or computational capacity or perceived required timeliness. However, normally these decisions are embedded deeply within the code or sensor choices, the separate annotation surfaces these decisions and separates these pragmatic decisions from the 'wiring' up of the components themselves.

As noted annotation is deliberately separated from the configuration as it will depend on the precise combination of components and the context in which they will be used. The separate annotation also means that the analysis tools (or had analysis) is separated from the use of the products of that analysis for optimisation during runtime.

## 8 Applying SE Components in Status/Event Infrastructures

We want to apply the generated Java Beans SE Components into existing distributed and ubiquitous infrastructures. We have chosen the ECT platform [3] because it supports events and states applying the concept of tuple spaces. In order to understand the requirements to support the status-event model and the advantages it can provide we present several computing architectures to deal with events and states.

### 8.1 Existing Status/Event Architectures

Over the last decade or so many researchers have attempted to design elegant and effective programming abstractions for building distributed systems. Space prohibits a full exploration here, but they can be loosely categorised as event based or state based.

#### 8.1.1 Event Based Architectures

The Cambridge Event Architecture (CEA) is an example of an event based system [16]. In the context of their work with ‘Active Badges’, which allowed the tracking of electronic badge wearers throughout their research lab, the event architecture added the facility to build monitors that composed raw events together (e.g. Person A in room X, fire alarm activated etc.) to construct higher level information about the state of the world (e.g. fire alarm activated then person A left the building).

CEA was constructed using finite state machines composed of directed acyclic graphs of states (‘beads’), representing start states, transitional states and final (accepting) states. Arcs could be standard (transition once when an event occurs) or spawning (create a new bead each time this transition occurs) – a spawning arc could be used to count every time a person left the building after the fire alarm, for example. Arcs may also be parameterised, which acts as a placeholder for information extracted from the state associated with each event (e.g. the badge holder’s name). Handlers can be added to accepting states to trigger notifications to the applications deploying the monitors.

CEA provided an elegant declarative approach for specifying monitors and handlers. However, as the authors acknowledged in their paper, the order in which events occurred was sometimes hard to determine in the distributed case (meaning state machines would not transition correctly), moreover, it was not possible to represent the timely nature of events, nor whether events occurred within a certain time of each other – which can lead to unexpected generation of notifications. Most importantly for this discussion, while the system captures state internally (e.g. sensor identity, badge id) and can make this available to handlers, the status of the world must be actively reconstructed from monitoring events; if the monitor is offline when the event occurs, there is no facility to recover it.

In classic distributed systems, processes communicate with each other using virtual channels described by bindings and formed from pairs of endpoints (e.g. the well known BSD 4.3 Sockets API). Elvin, originating from DSTC [17], is an event broker that decouples application components in a distributed system. Producers form a connection to a broker and emit events (notifications) consisting of one or more typed name value pairs. Consumers connect to the Elvin broker and subscribe to particular

notifications matching a given regular expression. Subscriptions may express criteria for the name, type, value and conjunction of fields of interest within an event. The broker optimises the flow of matching events from producers to consumers based on the set of subscriptions it tracks. Events are propagated at best effort pace via the broker. A key advantage of this approach is that consumers may subscribe to events at any time, allowing for easy introspection of the internal communication between applications. Like CEA however, there is no persistence in the system so status cannot be reconstructed until the appropriate events are observed first hand by the consumer.

Brokers can be federated (also using the subscription language) to create larger distributed applications. The API lends itself to the creation applications based on the publication of content, such news tickers, chat applications and diagnostic monitors.

### 8.1.2 State Based Architectures

As a total contrast we also briefly consider state driven architectures. The classic example of such an approach is the canonical work by Gelernter [18] – Gelernter observed that coordinating the allocation of distributed computations in massively parallel computer architectures was I/O bound; much of the expected gains in computational throughput lost in the inter processor communication to coordinate the distribution of tasks to processors. The innovation in his approach was to build a computational model based around the generation and consumption of state in the form of typed tuples in an entity known as a ‘tuple space’. A computational enhancement to existing programming languages (known as LINDA)<sup>1</sup> provided operations for adding, removing and observing content in the space. The task of allocating tasks to processors was turned from a producer driven model in which jobs were allocated to idle processors, to a consumer driven one in which idle processors pulled tuples (computations or part-computations) from the tuple-space and returned results to the space upon completion.

Since tuples persist in the tuple-space, producers and consumers do not have to be synchronously available to communicate – this is known as spatial and temporal decoupling. This feature of the paradigm has caused its adoption in mobile computing for dealing with loosely coupled distributed systems where parts of the application are seldom able to communicate synchronously [19,20].

As the tuple space paradigm has been used to build interactive systems it has become apparent that in order to support pacity interactions one must rapidly detect changes to the content of the tuple-space; something the original API was never designed for. Researchers have since augmented the standard API with additional operations (e.g. the EventHeap [21]) most notably to offer event notifications when tuples are added or removed [20]. Note that these operations are ‘change of state’ notifications on the tuple space, and do not support events between applications. If tuples reflect sensor values in the world then the tuple space may give us a historical view of status, but tuples do not necessarily reflect the current state of the world and it is left for the application writer to determine which information is the most current.

---

<sup>1</sup> Many researchers have since extended and explored alternate coordination based approaches; the interested reader is directed to Papadopoulos & Arbab [22].

### 8.1.3 Hybrid Event-State Architectures

The Equator Equip platform is<sup>2</sup> an example of a middleware that has drawn inspiration from the tuple-space concepts but, in addition, it includes support for passing events between applications. Equip ‘dataspaces’ contain typed objects that may contain state (as with the tuples in a tuple space). Different object types may be handled in different ways – objects that are of type ‘event’ trigger event notifications to a client application when they match an object representing a subscription to that type of event belonging to the application. This is, to our knowledge, the first system that attempts to offer an explicit separation between the use of the dataspace as a repository of shared information and the use of events for representing transient information (for example for tracking a user input device). By the taxonomy proposed by Papadopoulos and Arbab [22], Equip is a purely data-driven coordination model – all processes communicate via the dataspace; events are exchanged through the dataspace as specially typed objects.

The Equator Component Toolkit [3] (ECT) is a collection of java bean ‘components’ that provides a library of tools for constructing interactive ubiquitous computing applications. By linking ECT components together using a graphical editor designers can create limited ‘interactive workflows’ that are triggered by user interaction and interact with users in return via a range of physical prototyping and visualisation tools (such as Phidgets, MOTES, webcams etc). As components are dragged into the editor and as properties of the components are linked together to form a directed acyclic graphs, these get transformed into underlying objects, events and subscriptions in the Equip dataspace. Links between the properties of ECT components are stored as tuples in the shared Equip dataspace – note that, in contrast with channel based coordination models such as Reo [23], these links are application data as far as the dataspace is concerned and are not first class entities.

Distributed applications can be built by linking dataspaces together in client-server relationships or as synchronised peers. In equip, when two dataspaces link, historic events are explicitly regenerated by the system to bring both peers into exact synchronisation (a late joining client will see all events they would’ve seen had they been connected). When they disconnect, objects in a dataspace belonging to another peer are garbage collected. This behaviour has particularly interesting implications for interactive applications; the replay of historic state and events can appear as a fast motion replay of past activity, which is often meaningless or confusing for the user. Moreover, when a dataspace in a distributed application disconnects (potentially just because of a glitch in communications) and the data is garbage collected, the ECT components and connections between them that are represented are removed from the system (the application is partially destroyed). More importantly, the system would require a notion of which objects in the system represent status in the world and what the constraints are for their production and consumption to be able to optimise the flow of information between peers and application components. It is this issue we aim to explicitly address in XSED.

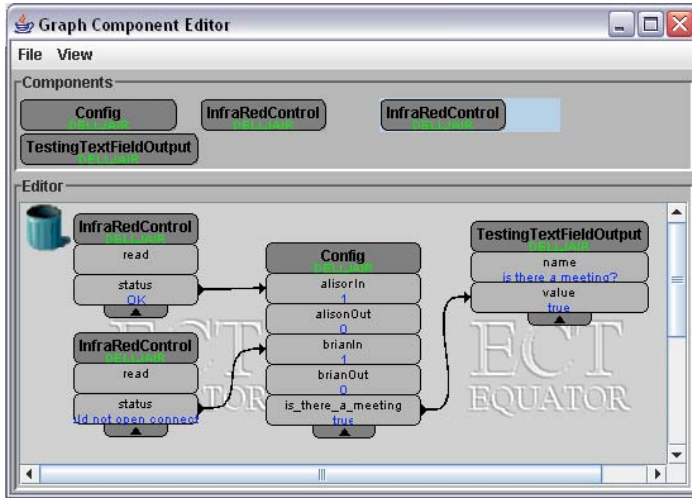
## 8.2 Generating SE Components to ECT Platform

Running the SE Components in the ECT platform [3] will enable us to use sensors, actuators and other components that have existing drivers/wrappers for ECT. In

---

<sup>2</sup> <http://equip.sourceforge.net/>

common with most ubicomp infrastructures, ECT is entirely event driven and this is achieved through listeners on Bean properties. Figure 9 show a SE Configuration Component (namely Config) running in ECT platform. The component is linked to three others ECT component (not generated by XSED) to illustrate our approach.



**Fig. 9.** A SE Configuration Component (Config) running in ECT platform

SE components are transformed into Beans using a wrapper class generated from the schema. For each input and output, both status and event, a Java slot is provided, but these are expected to be used differently depending on the type of node:

- (i) event input – when the slot is set, the appropriate `fireEvent` is invoked.
- (ii) status input – when the slot is set nothing happens immediately except the Bean variable being set, and when the component requires the status input (either when processing an event or when a status output is required), the variable is accessed.
- (iii) event output – when the component fires the event the listeners for the relevant slot are called.
- (iv) status output – when the `getName` method is called for a slot, the corresponding status output is requested from the component (which may require accessing status input).

The 'wiring' in (i) and (iv) is directly coded into generated code for the Bean, but (ii) and (iii) require an environment for the component as the SE component simply 'asks' the environment for input status and tells it when an output event is fired. A proxy environment object is therefore also generated that turns requests from the component for status input into accesses on the internal Bean variables and when told that an output event has fired turns this into an invocation of the Bean change listeners. Figure 10 summarises these connections when an `FSMComponent` is wrapped.

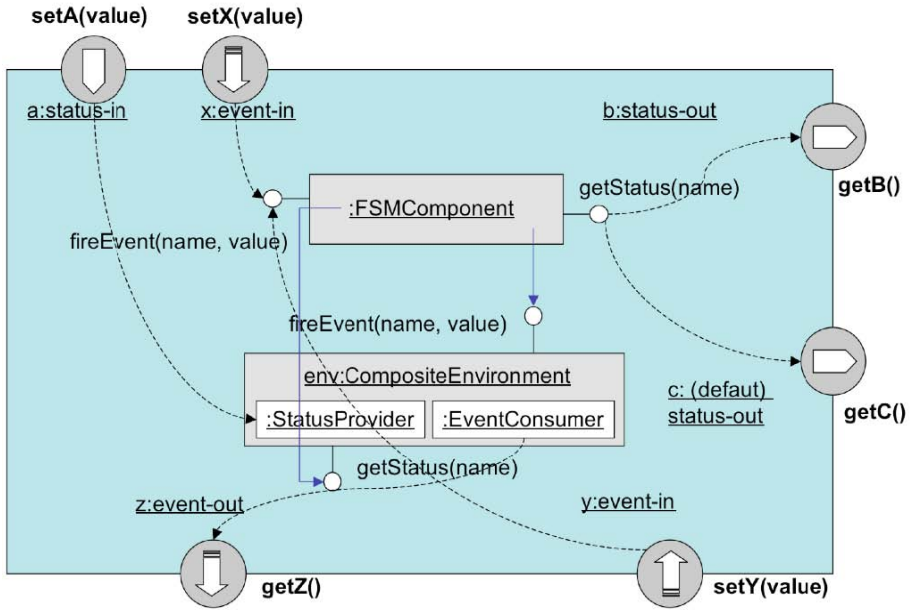


Fig. 10.

Unfortunately, in order to map Status–Event components into a Java Bean we have to effectively lose most of the distinction between Status and Event at the Bean level, both are properties; the differences between the properties are purely in the internal 'wiring' and in the expected way in which those properties will be accessed externally. While these wrapped Beans still provide explicit documentation of the status/event distinctions and also, to some extent, a more natural way of specifying the various status–event relations, it does lose the opportunities for more global reasoning and optimisation. In particular, we cannot throttle unneeded raw events or status-change events. Happily, the entire system formed by binding components with a configuration then forms a new component. So this compound component can also be wrapped into a Java Bean meaning that internally it can make use of the full richness of the SE environment including throttling.

## 9 Summary

We have seen how XSED allows descriptions of systems that include both status and event phenomena to be included naturally and without having to prematurely transform the status into discrete events. The notation separates components, configuration, binding and annotation allowing reuse and flexibility, but also allowing global analysis (by hand as now, or in future automated) to feed into optimisation of the execution over the infrastructure. We also saw how the symmetric treatment of input and output allowed the external environment of configurations of components to be treated as a component alongside others. The transformation onto Java has created efficient implementations of XSED components and systems and Bean wrappers allow these to be embedded within existing infrastructure, notably ECT.

Note that at a low level XSED specifications are still implemented as discrete events – this is the nature of computers. The crucial thing is that the specifications themselves do not assume any particular discretisation of status phenomena into lower-level system events. For the analyst/designer this means they describe what they wish to be true, not how to implement it. At a system level this means appropriate mappings onto discrete events can be made based on analysis not accident. The difference between XSED and more event-based notations is thus rather like between arrays and pointers in C-style languages, or even between high-level programming languages and assembler.

Future work on the underlying notation includes: refinement to allow status-change events (such as when  $\text{temp} > 100^{\circ}\text{C}$ ); alternative basic component specifications (e.g. process algebra based); ways of naming components (e.g. URIs) to allow binding to be controlled through XML files and additional annotations. In addition we plan more extensive case studies including distributed examples where the efficiency advantages can be fully appreciated.

## References

1. Dix, A.: Status and events: static and dynamic properties of interactive systems. In: Proc. of the Eurographics Seminar: Formal Methods in Computer Graphics. Marina di Carrara, Italy (1991)
2. Dix, A., Abowd, G.: Modelling status and event behaviour of interactive systems. *Software Engineering Journal* 11(6), 334–346 (1996)
3. Greenhalgh, C., Izadi, S., Mathrick, J., Humble, J., Taylor, I.: A Toolkit to Support Rapid Construction of UbiComp Environments. In: Proceedings of UbiSys 2004 - System Support for Ubiquitous Computing Workshop, Nottingham, UK (2004), <http://ubisys.cs.uiuc.edu/2004/program.html>
4. Flintham, M., Benford, S., Anastasi, R., Hemmings, T., Crabtree, A., Greenhalgh, C., Tandavanitj, N., Adams, M., Row-Farr, J.: Where on-line meets on the streets: experiences with mobile mixed reality games. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI 2003, Ft. Lauderdale, Florida, USA, April 05 - 10, 2003, pp. 569–576. ACM Press, New York (2003)
5. Dix, A., Abowd, G.: Delays and Temporal Incoherence Due to Mediated Status-Status Mappings (part of report on Workshop on Temporal Aspects of Usability, Glasgow, 1995). *SIGCHI Bulletin* 28(2), 47–49 (1996)
6. Wüthrich, C.: An analysis and model of 3D interaction methods and devices for virtual reality. In: Duke, D., Puerta, A. (eds.) *DSV-IS 1999*, pp. 18–29. Springer, Heidelberg (1999)
7. Massink, M., Duke, D., Smith, S.: Towards hybrid interface specification for virtual environments. In: Duke, D., Puerta, A. (eds.) *DSV-IS 1999*, pp. 30–51. Springer, Heidelberg (1999)
8. Willans, J.S., Harrison, M.D.: Verifying the behaviour of virtual environment world objects. In: Palanque, P., Paternó, F. (eds.) *DSV-IS 2000*. LNCS, vol. 1946, pp. 65–77. Springer, Heidelberg (2001)
9. Grossman, R., et al. (eds.): *HS 1991 and HS 1992*. LNCS, vol. 736. Springer, Heidelberg (1993)
10. Dix, A.: Finding Out -event discovery using status-event analysis. In: *Formal Aspects of Human Computer Interaction – FAHCI 1998*, Sheffield (1998)

11. Dix, A., Beale, R., Wood, A.: Architectures to make Simple Visualisations using Simple Systems. In: Proc. of Advanced Visual Interfaces - AVI 2000, pp. 51–60. ACM Press, New York (2000)
12. Goodger, B., Hickson, I., Hyatt, D., Waterson, C.: XML User Interface Language (XUL) 1.0. Mozilla.org. (2001) (December 2006),  
<http://www.mozilla.org/projects/xul/xul.html>
13. Nicol, G.: XTND - XML Transition Network Definition. W3C Note (November 21, 2000),  
<http://www.w3.org/TR/xtnd/>
14. Abowd, G., Allen, R., Garlan, D.: Using style to understand descriptions of software architecture. In: Notkin, D. (ed.) Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering, SIGSOFT 1993, Los Angeles, California, United States, December 08 - 10, 1993, pp. 9–20. ACM Press, New York (1993)
15. Ramduny, D., Dix, A.: Impedance Matching: When You Need to Know What. In: Faulkner, X., Finlay, J., Détienne, F. (eds.) HCI 2002, pp. 121–137. Springer, Heidelberg (2002)
16. Bacon, J., Moody, K., Bates, J., Chaoying, M., McNeil, A., Seidel, O., Spiteri, M.: Generic support for distributed applications. IEEE Computer 33(3), 68–76 (2000)
17. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: Proc. 18th Annual ACM Symposium on Principles of Distributed Computing (PODC 1999), pp. 53–61. ACM Press, New York (1999)
18. Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 255–263 (1985)
19. Picco, G.P., Murphy, A.L., Roman, G.: LIME: Linda meets mobility. In: Proceedings of the 21st international Conference on Software Engineering, Los Angeles, California, United States, May 16 - 22, 1999, pp. 368–377. IEEE Computer Society Press, Los Alamitos (1999)
20. Wade, S.P.W.: An Investigation into the use of the Tuple Space Paradigm in Mobile Computing Environments, PhD Thesis, Lancaster University (1999)
21. Ponnekanti, S.R., Johanson, B., Kiciman, E., Fox, A.: Portability, extensibility and robustness in iROS. In: 1<sup>st</sup> IEEE Pervasive Computing and Communications Conference (PerCom 2003), 23-26 March 2003, pp. 11–19 (2003)
22. Papadopoulos, G.A., Arbab, F.: Coordination models and languages. In: Centrum voor Wiskunde en Informatica. Advances in Computers, vol. 46. CWI Press (1998)
23. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. 14(3), 329–366



# Identifying Phenotypes and Genotypes: A Case Study Evaluating an In-Car Navigation System

Georgios Papatzanis<sup>1</sup>, Paul Curzon<sup>1</sup>, and Ann Blandford<sup>2</sup>

<sup>1</sup> Department of Computer Science, Queen Mary, University of London,  
Mile End Road, London, E1 4NS, UK

gp@dcs.qmul.ac.uk, pc@dcs.qmul.ac.uk

<sup>2</sup> UCL Interaction Centre, University College London,  
Remax House, 31-32 Alfred Place, London WC1E 7DP, UK  
A.Blandford@ucl.ac.uk

**Abstract.** There are a range of different usability evaluation methods: both analytical and empirical. The appropriate choice is not always clear, especially for new technologies. In-car navigation systems are an example of how multimodal technologies are increasingly becoming part of our everyday life. Their usability is important, as badly designed systems can induce errors resulting in situations where driving safety may be compromised. In this paper we use a study on the usability of a navigation device when the user is setting set up an itinerary to investigate the scope of different classes of approach. Four analytical and one empirical techniques were used to evaluate the usability of the device. We analyse the results produced by the two classes of approach – analytical versus empirical – and compare them in terms of their diversity and the insight they provide to the analyst in respect to the overall usability of the system and its potential improvement. Results suggest a link between genotypes and the analytical class of approach and phenotypes in the empirical class of approach. We also illustrate how the classes of approach complement each other, providing a greater insight into the usability of a system.

**Keywords:** Usability evaluation, UEMs, In-car navigation, Cognitive Walkthrough, UAN, EMU, Design Criteria, Phenotypes, Genotypes.

## 1 Introduction

Various techniques can be used for the evaluation of interactive systems. Techniques are classified according to their approaches in conducting evaluation: *analytically* when a simulation is performed by an expert/analyst to predict the behaviour of the user and detect potential problems, without the involvement of users; *empirically* when the system is tested by users while their performance and problems are recorded.

The aim of this paper is to report on a study where both analytical and empirical approaches were employed to evaluate an in-car navigation device. In this study we concentrated solely on tasks related to the programming of the device (destination entry) before the user starts driving the car. We look at the results from a qualitative perspective; we do not seek to establish efficiency counts (number of usability problems)

for different techniques or approaches. Instead we analyse the results produced by the two classes of approach – analytical and empirical – and compare them in terms of their diversity and the insight they provide to the analyst in respect of the overall usability of the system and its potential improvement. We investigate the variance of results between the classes of approach and explore the association of genotypes and phenotypes with the empirical and analytical classes of approach respectively.

## 2 Background

Car navigation systems are designed to guide the driver through a generated route toward a selected destination. Drivers are interacting with such devices when programming the destination point and customising the route (touch screens, dials, voice input) and whilst driving when receiving instructions from the device (maps, visual cues, voice instructions). As a result navigation systems can cause driver distraction (a) when entering information in the device and (b) when following the driving instructions issued by the system. The different modes of interaction with the device have varying effects on the driving performance.

The usability of navigation devices is a contributing factor to the overall safety of car driving. Nowakowski et al. [19] carried out heuristic analysis and user testing on navigation systems and exposed a set of recurring usability problems. Nowakowski *et al.* identified problems in both destination entry and guidance modes: (a) layout and labelling of the control menus; audio and visual feedback; order of entry of destination information, and (b) starting guidance and ending; display design and readability; voice guidance and timing; rerouting. In this paper we examine aspects of the device related to the preparation of a route, before the device commences with the navigational instructions to the car driver.

Various case studies are reported in the literature with regard to the evaluation of usability methods ([18], [1], [12], [4], [6]). Comparisons between methods have been carried out in terms of problem count, scope, validity, evaluator effect, etc. Wright and Monk [22] also carried out case studies reporting on the difference of usability evaluation results obtained between users or usability experts and the system designers when applying cooperative evaluation to the same system.

In this study we take a different perspective and make a comparison between analytical and empirical classes of approach on two discrete dimensions. The first dimension considers usability problems identified and the insight they provide to the analyst into the usability of a system. Secondly, we look at the usability issues in terms of phenotypes – overt and observable manifestations of an incorrectly performed action – and the contrasting genotype – the underlying likely cause which eventually can account for the phenotype [8] [9]. Phenotypes describe observable behaviour, while genotypes are concerned with the interpretation of such behaviour.

## 3 Method

The case study was executed in two discrete parts: analytical and empirical evaluation, followed by an analysis of the results comparing the two classes of

approach. In each, the usability of the selected application was assessed against a predefined scenario and set of tasks (see Table 1). The scenario and tasks were based on the activities carried out by the driver prior to driving to the destination, i.e., preparing an itinerary on the navigation device. Such tasks take place in the car while stationary. This set of tasks enabled us to assess a wide range of primary functions that are frequently used in such devices.

**Table 1.** Sample tasks used for the evaluation

*Task 1: Program the device to reach the city centre of Leeds.*

*Task 2: Program the device to reach the following restaurant before the final destination.*

World Service  
Newdigate House  
Castle Gate  
Nottingham  
NG1 6AF

*Task 3: Check the route to see if you are using the M621. If you do, program the device to avoid this part of the route.*

In the analytical part of the study we applied a series of analytical methods in the evaluation of the navigation system. The first author of the paper carried out the analytical evaluations of the system. The personal judgement and experience of an analyst, may have a significant impact on the results (known as the evaluator effect [6] or craft skill). Nevertheless, in this study we focus more on the types of problems reported by each class of approach, rather than contrasting problem counts. We compare the results as identified by the different classes of approach, empirical vs. analytical, rather than comparing the different sets of issues within each class of approach. As a result, the evaluator effect has minimal impact on our comparison. Furthermore, two usability experts independently reanalysed the results with respect to genotypes and phenotypes.

Four methods were chosen for the analytical part of the study, employing a diverse approach to evaluation. In the subsequent sections, we describe these techniques and identify various issues pertaining both to their applicability and their effectiveness as identified during the study. The methods selected are characterised by a varying degree of formality, with each advocating a different approach to user interface evaluation. Each method has its own potential merits in the evaluation of this device. Cognitive Walkthrough [20] was selected as it is most suitable for walk-up-and-use interfaces. EMU (Evaluating Multi-Modal Usability) [10] is a technique specifically implemented for multimodal systems, thus appropriate for this type of device. UAN (User Action Notation) [5] provides an extensive notation, incorporating temporal issues and patterns for the specification of the interface. Leveson's design guidelines [13] were selected because of their focus on error detection and analysis. The diversity of these techniques gives us an increased capacity for the detection of usability issues, giving a wide range to compare against those found empirically.

In the second part of the study we carried out an empirical evaluation of the device, using the same scenario and tasks as in the first part of the study. The empirical evaluation was carried out in a usability laboratory, as the context (being in a car) is not relevant for the set of tasks selected for this study. We focused our attention on the usability issues that drivers encounter in the use of such devices, employing an exploratory approach.

### 3.1 Car Navigation System

Navigation systems are increasingly becoming standard equipment in motor vehicles. Their usability is an important factor, as badly designed systems can induce errors resulting in situations where driving safety is compromised. Although manufacturers suggest that users must read the entire manual before operating such navigation systems, it is often the case that they are used by drivers as walk-up-and-use devices.



Fig. 1. (a) Main menu & (b) House number entry

The navigational device selected for this study utilises the TomTom Navigator 5 application running on an HP iPAQ handheld computer. The user can manipulate the application through the user interface displayed on the touch screen of the device. The device offers visual and voice instructions to the user in order to guide them through the itinerary. The system offers the functionality usually found in navigational devices, such as looking up and navigating to an address or point of interest, re-routing and generating different routes to a selected destination. The system is accessed via a touch-screen based interface comprising a set of menus and data entry screens (see Fig. 1).

## 4 Analytical Techniques

In this section, we briefly outline each analytical method used, describing the empirical study methodology in the next section.

## 4.1 Cognitive Walkthrough

Cognitive Walkthrough (CW) is an informal inspection methodology for systematically evaluating features of an interface in the context of the exploratory theory CE+ [14] [20]. Wharton *et al.* [20] present CW as a theoretically structured evaluation process that follows the application of a set of questions asked about each step in the task, derived from the underlying theory, and attempting to focus the attention of the analyst on the CE+ claims. The questions are preceded by a task analysis and the selection of the appropriate sequence of user actions to successfully perform a task (preparatory phase). During the execution of the method (analysis phase), the analyst simulates the execution of the sequence of user actions and assesses the ease of learning of the design, by using the questions as summarised and exemplified in Table 2.

**Table 2.** Cognitive Walkthrough extract

*Task:* Enter house number

*Question 1:* Will the users try to achieve the right effect?

No. The system requires information not known by the driver.

*Question 2:* Will the user notice the correct action is available?

Probably not. The driver should select 'done' on this screen, in order to avoid inputting a house number.

*Question 3:* Will the user associate the correct action with the effect trying to be achieved?

No. The driver might attempt to enter a random number to skip this screen.

*Question 4:* If the correct action is performed, will the user see that progress is being made towards the solution of the task?

Not really. Once the selection is made the system automatically starts calculating the route without any further confirmation. The markers and labels on the map are indiscernible or non-existent and cannot confirm the route that the driver has been trying to build up.

In this extract the user is asked to enter a house number as part of the destination input, although such information is not provided in the use scenario. Although this information is not required by the system, there is no clear way to skip this step.

In this study CW reported a series of issues relating to feedback, consistency of design, labels, task structure, and user interface navigation. The bulk of the issues identified by the technique are attributed to the analyst's craft skill, rather than the technique itself. Nevertheless, the technique led the analyst to engage deeply with the system in order to arrive at these results.

## 4.2 UAN (User Action Notation)

UAN [5] [7] is a behaviour-based notation specifying user actions, computer feedback and interface internal state at the same time. UAN is primarily a shorthand way to represent steps (such as "mouse down") that a user would take to perform a task on a given user interface, existing or under development.

The notation is semi-formal in that it makes use of visually *onomatopoeic* symbols, for example  $Mv$  represents a “mouse down” action, while  $M^{\wedge}$  represents a “mouse up” action. The goal of UAN is to represent simple and complex user tasks in a notation, that is easy to read and write, but one that is more formal, clear, precise, and unambiguous than English prose. As it is not overly formal it is assumed that designers can learn the notation without major problems.

**Table 3.** Extract from UAN specification of the user interface

TASK: Enter street number			
USER ACTIONS	INTERFACE FEEDBACK	INTERFACE STATE	CONNECTION TO COMPUTATION
&~ [number']* $Mv$	number'!	key selected = number'	put number'in field
$M^{\wedge}$	number'-!	key selected = null	
~ [Done] $Mv$	Done!		
$M^{\wedge}$	Done-!		If field isNull then number = default else selected number = field number; go to map screen

In the extract shown in Table 3, we describe the interaction with the user interface in the house entry dialogue. During the interaction the user selects the appropriate number (~ [number']\*) using the virtual numerical keyboard, while the ‘done’ button is used to complete the task. From the specification we can easily distinguish the feedback (number'!) provided at each step of the interaction, as the system updates (key selected = number') its variables and displays (put number'in field) the relevant information.

Although UAN is not necessarily a suitable technique for identifying usability problems, the specification of the interface enforces the analyst to deconstruct the user interface and identify issues hidden in its design. In the UAN analysis we identified mainly issues related to feedback, design and labelling of the interface.

### 4.3 EMU – Evaluation Multi-modal Usability

*EMU (Evaluation Multi-modal Usability)* [10] is a methodology developed to address specifically the evaluation of usability of multi-modal systems. The scope of the methodology extends to issues related to user knowledge and use in context, with a special focus on the issues concerned with the physical relationship between the user and the device [1]. Multimodal interfaces can enhance the user’s understanding, as they provide more communication channels between the user and the system.

**Table 4.** EMU stages

<p><i>Stage 1. Define the task that is to be analysed</i></p> <p><i>Stage 2. Modality lists</i></p> <p><i>Stage 3. Define the user, system and environment variables</i></p> <p><i>Stage 4. Profiles compared to modality listings</i></p> <p><i>Stage 5. Interaction modality listing</i></p> <p><i>Stage 6. Add in clashes, etc.</i></p> <p><i>Stage 7. Assess the use of modalities</i></p> <p><i>Stage 8. Final report.</i></p>
---

EMU methodology presents a novel approach to the evaluation of multimodal systems. It presents a comprehensive taxonomy, underpinned by a new theory on multimodality, tightly coupled with a notational representation and a structured step-by-step approach for its application. In this evaluation, we applied the methodology as described in the EMU tutorial [10]. The methodology is executed in several stages (see Table 4) in order to identify the various modalities (see Table 5) of the interaction and any usability issues resulting from these modalities.

**Table 5.** Extract from EMU analysis

<p><b>Display</b></p> <p>[UE hap-sym-dis] *user types the house number *</p> <p>[SR hap-sym-dis] *system records house number *</p> <p>[SE vis-sym-dis] *system flashes pressed buttons*</p> <p>[UR vis-sym-dis] *user sees pressed button*</p> <p><b>and</b></p> <p>[SE vis-lex-cont] *number appears on house number field*</p> <p>[UR vis-lex-cont] *user reads house number field*</p> <p>precon: UE [hap-sym-dis] *user types numbers*</p> <p><i>key</i></p> <p><i>SE: System Expressive (expressed by the system)</i></p> <p><i>SR: System Receptive (received by the system)</i></p> <p><i>UE: User Expressive (expressed by the user)</i></p> <p><i>UR: User Receptive (received by the user)</i></p> <p><i>hap: haptic, vis: visual, lex: lexical,</i></p> <p><i>sym: symbolic, dis: discrete, cont: continuous</i></p>
--

Table 5 gives us an extract of the EMU analysis for the house entry dialogue of the system as shown in Fig.1 (b). The user enters the information ([UE hap-sym-dis]) into the system using the touch screen display. As the system receives the information ([SR hap-sym-dis]), the appropriate visual feedback ([SE vis-sym-dis]) is received by the user ([UR vis-sym-dis]) for each button pressed. At the same time the user can read the information ([UR vis-lex-cont]) provided to the system, as it is shown in the relevant display ([SE vis-lex-cont]).

Due to the nature of the tasks under evaluation, there were only a very limited number of modality clashes identified as part of EMU analysis. Nevertheless, the analysis gave the analyst the opportunity to examine the system from a different perspective, resulting in an extensive set of usability problems, with a wider scope not solely related to multimodal issues, but also labelling, interface design, and interface navigation issues.

#### 4.4 Design Guidelines

The use of design guidelines (DG) or design criteria has been a common practice for the evaluation of user interfaces. The conformance of the interface design to an appropriate set of guidelines can improve the usability of an application. In the HCI literature one can find different sets of guidelines to suit different domains and applications [15]. Guidelines can be used for helping designers resolve design problems, or for the evaluation of an interface.

The design guidelines of Nielsen and Molich [17] have been widely used in the HCI community in order to improve the usability of interactive systems. This method, heuristic evaluation [17], is suitable for quick and relatively easy evaluation. The analyst carries out a systematic inspection of the interface to identify usability problems against a set of guidelines, also known as heuristics.

In the field of safety-critical systems, the analyst seeks to identify high-risk tasks and potentially safety-critical user errors through system hazard analysis. Various sets of guidelines for detecting design flaws, which might cause errors leading to safety-critical situations, can be found in the literature (e.g., [11] [13]).

Jaffe [11] and Leveson [13] have created sets of guidelines for the design of safety-critical systems. In this study, we used a subset of the Human-Machine Interface (HMI) Guidelines [13]. These guidelines are based partly on an underlying mathematical model, but to a greater extent on the experience of the authors in the design and evaluation of safety-critical systems used in cockpits. As a result these guidelines are greatly influenced by issues pertinent to the particular domain.

Although these guidelines were not intended for the usability evaluation of interactive systems, we applied them in a similar way that an analyst would apply guidelines in Heuristic Evaluation [16]. Every screen of the system in the task sequence was assessed against a subset of the Design Guidelines. During the evaluation we only used a restricted subset as many of them were either domain-specific or irrelevant to our device. This reduced the number of times that the analyst had to traverse through the list of guidelines during the evaluation session.



**Table 6.** Extract from the subset of Design Guidelines used for the evaluation of the interface

<i>Design Guidelines</i>
1. Design for error tolerance: (a) make errors observable, (b) provide time to reverse them, and (c) provide compensating actions
2. Design to stereotypes and cultural norms
3. Provide adequate feedback to keep operators in the loop.
4. Provide facilities for operators to experiment, to update their mental models, and to learn about the system. Design to enhance the operator's ability to make decisions and to intervene when required in emergencies.
5. Do not overload the operator with too much information. Provide ways for the operator to get additional information that the designer did not foresee would be needed in a particular situation.
6. Design to aid the operator, not take over.

Applying this technique in the house entry dialogue (Fig. 1 (a)), as shown before with other techniques, we identified several issues that violated the design guidelines. Table 7 gives extracts from the analysis detailing some of the problems and the associated guidelines that have been violated.

**Table 7.** Extract from the Design Guidelines analysis of the system

<i>(Guideline 1)</i> If the users change their mind or realise they needed a different postcode, it is impossible to return to the previous page to rectify their action. The user will have to cancel the interaction and start again from Step 1.
<i>(Guideline 2)</i> It is not possible on this page to confirm that the right selection has been made in the previous page. An instant flashing message is displayed to the user when the page loads, but it can be easily missed.
<i>(Guideline 3)</i> There is no label associated with the arrow button.

DG were drafted to be used for the design of safety-critical systems. In this study, we identified a range of usability problems in the process of analysis – labelling, navigation, feedback, as well as issues relating to error recovery which are specifically targeted by the method.

## 5 Empirical Study

An empirical study can potentially give important insights regarding issues of context of use that analytical methods might fail to capture. We also investigated issues concerning whether analytic and empirical methods can be combined into a composite method for comprehensive coverage of problems in such environments. It has previously been suggested [2] that the combination of empirical and theoretical

analysis can provide a greater insight than the individual approaches into the issues of such an application area.

Eight users participated in the experiment, including both male and female members of the academic community. The empirical study was split into two parts. All participants participated in both parts of the experiment. The first part was a training session where users were expected to follow a given scenario and carry out a set of three tasks (see table 8). During this part of the trial, the users were allowed to ask questions of the experimenter. The goal of this session was to allow the participants to familiarise themselves with the device before continuing to the main trial session. Participants were provided with a sheet containing the scenario and tasks and a print-out containing a set of screens from the device.

**Table 8.** Tasks used for training session

<p><i>Task 1: Program the device to reach the Berkeley Hotel, Brighton.</i></p> <p><i>Task 2: Program the device to reach the following restaurant before the final destination.</i></p> <p style="text-align: center;">IKEA Croydon Volta Way Croydon CR0 4UZ</p> <p><i>Task 3: Check your route to see if you are using A22. If you are, program the device to avoid this part of the route.</i></p>
--

During the second part, users followed a different set of (similar) tasks in a new scenario. At this stage the experimenter did not interfere with the tasks. In the second part of the empirical study we used the task list that was also used with the analytical techniques of the study (see Table 1). In both sessions of the experiment we used TomTom Navigator 5 software running on an iPAQ handheld computer connected to a TomTom GPS device via Bluetooth, as described in previous sections.

During the experimental trials we collected video and audio data from the interaction between the user and the system using a video camera. We also captured a video stream of the information shown on the screen of the iPAQ device. The video data for each participant were synchronised and merged before we started a thorough analysis of the interaction.

Firstly, we started with the transcription of the sequence of actions that each user followed in order to achieve the tasks as set out in the experiment trials. Each interaction step was recorded and matched against the current state of the interaction device. Having completed this process, we analysed the data, in order to understand the problems that the users encountered during the interaction and how their sequence of actions compare to the sequence of actions required to successfully complete the tasks. We grouped together relevant sequence events and identified repeating patterns between users in their interactions.

## 6 Results from Analytical and Empirical Evaluations

In this study we examined several parts of the system over three different tasks selected for evaluation. We identified a set of over 50 usability problems attributed to one or more techniques.

Although the analytical and empirical techniques managed to identify a similar number of issues during the analysis, the types of issues varied significantly. Each *class of approach* identified a distinct set of issues, while only a few usability problems were identified by both classes of approach.

We overview briefly below the two subsets of usability problems – analytical and empirical – and where these subsets intersect. For the purposes of the analysis we give for illustration (Table 9) a representative sample of usability problems collated during the analytical and empirical study.

**Table 9.** Extract of usability problems list

		CW	UAN	EMU	DG	EMP
1.	No way to edit the route from view menu			☒	☒	☒
2.	No clear way to bypass house number entry	☒		☒	☒	☒
3.	Invalid input through address entry					☒
4.	Wrong mode					☒
5.	Inappropriate headings in menus	☒	☒	☒	☒	
6.	Inconsistent colour scheme within menus	☒			☒	

The first two usability problems identified in the Table 9 were captured by both analytical and empirical techniques.

### 1. *No way to edit the route from view menu*

The design of the menu structure prohibited the users from making any changes to the generated route from the set of menus used to view the route. As a result, users frustratingly navigated through the system menus to locate the appropriate functionality.

### 2. *No clear way to bypass house number entry*

The second task of the user trial involved programming the navigation device to reach a restaurant. The address provided to the user in the context of this trial did not include a street number. Nevertheless, the system asks for this piece of information (see Fig.1 (b)) as part of the address, without an obvious way to bypass it.

Analytical techniques identified the issue in the study, offering design recommendations to resolve it. In the empirical study, as users were not aware of a street number for the restaurant, they employed various strategies, as there was no evident way to skip this step. Some users entered a random number, while some others chose the ‘crossing menu’ functionality in order to complete their task.

The next two usability problems (number 3 & 4) identified were captured by the empirical evaluation only.

### 3. *Invalid input through address entry*

The system offers different interactive dialogues to the user to input the information about the destination in terms of an address, postcode, point of interest, city centre, etc. Users repeatedly attempted to input through the address dialogue, information other than that asked for at the time by the device. Although the first screen of this dialogue asks for the city name of the destination, users tried to enter the postcode, name of the restaurant, street name, etc. Apparently users did not identify the specific dialogue for postcode entry, subsequently trying to communicate the postcode to the system through this menu, since another option was not readily available. This led to confusion, decreasing significantly the usability of the system.

### 4. *Wrong mode*

Another issue identified in the empirical study only refers to the user being in the wrong mode. The system incorrectly captured the intentions of the user without the user realising. This was identified as the user attempted to carry out the 2<sup>nd</sup> task, i.e., setting up an intermediate destination in the itinerary, through the address dialogue. More specifically, the user input part of the name (N-O-T-T) of the stopover town (Nottingham) and the system automatically updated the appropriate list, according to the user input. As it was being updated, Nottingham momentarily appeared on top of the list, before it went to second place in order to be replaced by Notting Hill. Nevertheless, the user selected the first item on the list, having not realised that the list had changed before the selection was made.

Under these circumstances the user arrived at the next screen, 'Travel via/Address/Street entry', under the illusion that Nottingham was selected in the previous screen. As a result the user was unsuccessful in locating the street or the restaurant on the list, as the wrong city was selected.

The last two usability problems identified that we discuss here were captured only by analytical class of approach:

### 5. *Inappropriate headings in menus*

The lack of appropriate headings throughout the application was picked up by all analytical techniques applied in this study. Titles are necessary as they provide orientation and constant feedback to the user. Missing or inappropriately used headings decrease significantly the usability of a system.

### 6. *Inconsistent colour scheme within menus*

Colour schemes can be used to group together similar functions, while at the same time offering the sense of continuity to the user when progressing through the task. In this system the colour scheme is used inconsistently, resulting in inappropriate feedback and sense of confusion by the user. This was picked up by DG as it violated the respective guideline, while it was also identified in the process of the CW.

## 7 Analysis of the Results

The types of issues captured by analytical and empirical techniques vary significantly. Some usability problems were identified by both classes of approach (analytical and empirical), but many were identified only by one or the other. In the previous section

we presented a set of usability problems representing these categories and as tokens of the usability problems identified.

One important aspect that emerges when looking at the results is the variability between the coverage of results reported by analytical and empirical approaches. There is only a small overlap on the issues identified by the two approaches. The vast majority of usability problems were independently identified by one class of approach only.

Under closer investigation we also observe that the type of problems detected by the approaches is significant. While the analytical techniques identified mainly usability problems that might create difficulties to the users, the empirical data demonstrated specific instances of user behaviour where users experienced such difficulties. The usability problems reported by the empirical approach are associated with the manifestations of user errors, while the usability problems reported by the analytical approach correspond to the underlying cause of such manifestations. This correspondence thus relates to the phenotype – observable manifestations of an incorrectly performed action – and the contrasting genotype – the underlying likely cause [8] [9].

**Table 10.** Extract of reanalysis of usability problems

		Analytical	Empirical	Expert 1	Expert 2
1.	No way to edit the route from view menu	☒	☒	<i>genotype</i>	<i>genotype</i>
2.	No clear way to bypass house number entry	☒	☒	<i>genotype</i>	<i>genotype</i>
3.	Invalid input through address entry		☒	<i>phenotype</i>	<i>phenotype</i>
4.	Wrong mode		☒	<i>phenotype</i>	<i>phenotype</i>
5.	Inappropriate headings in menus	☒		<i>genotype</i>	<i>genotype</i>
6.	Inconsistent colour scheme within menus	☒		<i>genotype</i>	<i>genotype</i>

In order to investigate the association of genotypes and phenotypes with their respective classes of approach – empirical and analytical, the first author and a further two usability experts independently assessed the issues identified in the study in terms of genotypes and phenotypes. The experts did not have any prior knowledge of the results or their association to any technique or class of approach. They were provided with the complete list of issues, as identified by both classes, and were instructed to assign each issue as a phenotype or as a genotype. The experts were able to match the majority (over 95%) of the issues to the type of error, as we had hypothesised with the correlation between genotypes, phenotypes and their respective classes of approach. Table 10 gives the reanalysis of the usability problems presented in Section 6. More specifically, the issues identified by the empirical class of approach were assigned as phenotypes, whereas the issues identified by the analytical class of approach were assigned as genotypes. In the extract presented in Table 10, the problems captured by

both classes of approach were classified as genotypes by the experts. Further work is needed to investigate the overlap cases.

Matching phenotypes to their respective genotypes during the analysis of the results in the study turned out to be a difficult feat. Although we were able to identify several manifestations of user difficulties, we were unable to directly pinpoint the underlying cause; we could only theorise about possible candidates. For example, a phenotype identified in the study was issue three from Table 9. As explained in Section 6, the user attempted to make an invalid entry through the address dialogue. There are several genotypes that can be potentially associated with this issue, such as inappropriate headings, inconsistent interface design, grouping of functions, etc. Although some of them could be perspective candidates it is not easy to establish a link between them. The lack of the specific underlying causes prevents us from making design recommendations in order to remove such user difficulties, identified as phenotypes. Such relationships, between genotypes and phenotypes could eventually be established through further experimental studies examining the appearance (or not) of the phenotypes, once a genotype has been removed from the system. However this would be a very time-consuming approach.

Usability evaluation methods are used in order to improve the usability of a system. This is done through the identification of usability problems and a set of design recommendations, which are subsequently applied to improve the usability of the system under investigation. We have seen in this study that the empirical study mainly focused on the identification of phenotypes, which does not lead directly to the improvement of a system, as it does not provide causal explanations needed in many cases as a precursor for recommendations on how to do so. Nevertheless, the phenotypes also serve their purpose as they are reminders to designers and developers of the difficulties or problems encountered by the users and their satisfaction while using the system.

Although an empirical approach can identify in a system difficulty of use or usability problems, it does not readily identify or resolve the underlying causes of the issues identified. An alternative approach should be followed for the identification of the genotypes. As demonstrated in this study, the analytical approaches fare well in this task. The coverage of results collated by the analytical techniques used in this study concentrates mainly on the genotypes. Furthermore an explicit part of some of the techniques – such as EMU and CW – is the identification of design recommendations that can be used for eradicating the genotypes from the system under evaluation.

Nevertheless, this does not reduce the value of the empirical approach. Wixon [21] argues that the evaluation of a system is best accomplished within the context of use for each system, advocating a more exploratory approach, through the use of case studies and user involvement. Furniss [3] also argues that demonstrating user problems (phenotypes) is a significant step for persuading design teams and relevant stakeholders to introduce changes to systems. In contrast, expert reports (describing genotypes) can be more easily dismissed. Thus, the use of phenotypes might be used for persuading the appropriate stakeholders as needed, while genotypes can help the design times understand better the underlying causes and offer more appropriate solutions.

As illustrated above neither of the two approaches can serve as a panacea for evaluating an interactive system. Using an analytical or empirical approach can only

have a limited effect on the overall usability of the system. Each approach offers different insights and power to the analyst and the synergy of a combined approach can provide a more complete approach to usability evaluation.

## 8 Conclusion

In this study we set out to compare different evaluation techniques by evaluating the usability of a car navigation device. Our efforts were concentrated on the aspects of the device relating to the preparation of a route, before the device commences with the navigational instructions to the driver of the car.

In the previous sections we examined the analytical and empirical techniques that were employed during the study. Each technique employed in this study offers a different perspective into the usability evaluation of interactive systems and identified different sets of issues. In this study we focused on the kind of usability problems reported from each class of approach. According to the results of the study, the analytical class of approach is most powerful as a way of identifying genotypes, while the empirical class of approach is best at identifying phenotypes. These results support the argument that a combination of analytical and empirical approaches can offer a richer insight into the usability of the system and give the usability practitioner greater argumentative power, as their findings complement each other.

The combinatory use of the complementary approaches described above still remains a challenge for the analyst. The association of phenotypes with their respective genotypes is a difficult task, but necessary in the process of increasing the usability of a system, when adopting such an approach. Further work needs to be carried out into making this process easier for the analyst to undertake. Taxonomies identifying domain specific genotypes and phenotypes could eventually assist the analyst relating observational behaviour to underlying cause, resulting in a deeper insight into the usability of a system.

In order to assess further the scope of each technique and approach in a dynamic environment, we are carrying out another study where the tasks selected are representative of the user experience while driving and taking instructions from a navigation device. This future study will give us further insight into the appropriateness of the methods when using such devices in a constantly changing environment and where the goals of the users are not preconceived as is the case in this study.

**Acknowledgments.** The work described in this paper has been funded by the EPSRC Human Error Modelling project (GR/S67494/01 and GR/S67500/01). We are grateful to all participants in our studies.

## References

1. Blandford, A.E., Hyde, J.K., Connell, I., Green, T.R.G.: Scoping Analytical Usability Evaluation Methods: a Case Study. Journal publication (submitted, 2006)
2. Curzon, P., Blandford, A., Butterworth, R., Bhogal, R.: Interaction Design Issues for Car Navigation Systems. In: 16th British HCI Conference, BCS (2002)

3. Furniss, D., Blandford, A., Curzon, P.: Usability Work in Professional Website Design: Insights from Practitioners Perspectives. In: Law, E., Hvannberg, E., Cockton, G. (eds.) *Maturing Usability: Quality in Software, Interaction and Value*. Springer, Heidelberg (forthcoming)
4. Gray, W.D., Salzman, M.C.: Damaged Merchandise? A Review of Experiments That Compare Usability Evaluation Methods 13(3), 203–261 (1998)
5. Hartson, H.R., Gray, P.D.: Temporal Aspects of Tasks in the User Action Notation. *Human-Computer Interaction* 7(1), 1–45 (1992)
6. Hertzum, M., Jacobsen, N.E.: The Evaluator Effect: A Chilling Fact About Usability Evaluation Methods 15(1), 183–204 (2003)
7. Hix, D., Hartson, H.R.: *Developing User Interfaces: Ensuring Usability Through Product and Process*. John Wiley and Sons, Chichester (1993)
8. Hollnagel, E.: The phenotype of erroneous actions. *International Journal of Man-Machine Studies* 39(1), 1–32 (1993)
9. Hollnagel, E.: *Cognitive Reliability and Error Analysis Method*. Elsevier Science Ltd., Oxford (1998)
10. Hyde, J.K.: *Multi-Modal Usability Evaluation*, PhD thesis. Middlesex University (2002)
11. Jaffe, M.S., Leveson, N.G., Heimdahl, M.P.E., Melhart, B.E.: Software Requirements Analysis for Real-Time Process-Control Systems. *IEEE Trans. on Software Engineering* 17(3), 241–258 (1991)
12. Jeffries, R., Miller, J.R., Wharton, C., Uyeda, K.: User interface evaluation in the real world: a comparison of four techniques. In: *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*. ACM Press, New Orleans (1991)
13. Leveson, N.G.: *Safeware: System Safety and Computers*. Addison-Wesley, Reading (1995)
14. Lewis, C., Polson, P.G., Wharton, C., Rieman, J.: Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces. In: *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*. ACM Press, Seattle (1990)
15. Newman, W.M., Lamming, M.G., Lamming, M.: *Interactive System Design*, p. 468. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (1995)
16. Nielsen, J.: Heuristic Evaluation. In: Nielsen, J., Mack, R.L. (eds.) *Usability Inspection Methods*. John Wiley & Sons, New York (1994)
17. Nielsen, J., Molich, R.: Heuristic evaluation of user interfaces. In: *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*. ACM Press, Seattle (1990)
18. Nørgaard, M., Hornbæk, K.: What do usability evaluators do in practice?: an explorative study of think-aloud testing. In: *Proceedings of the 6th ACM conference on Designing Interactive systems*. ACM Press, University Park (2006)
19. Nowakowski, C., Green, P., Tsimhoni, O.: Common Automotive Navigation System Usability Problems and a Standard Test Protocol to Identify Them. In: *ITS-America 2003 Annual Meeting*. Intelligent Transportation Society of America, Washington (2003)
20. Wharton, C., Rieman, J., Lewis, C., Polson, P.: The cognitive walkthrough method: a practitioner's guide. In: *Usability inspection methods*, pp. 105–140. John Wiley & Sons, Inc., Chichester (1994)
21. Wixon, D.: Evaluating usability methods: why the current literature fails the practitioner. *Interactions* 10(4), 28–34 (2003)
22. Wright, P.C., Monk, A.F.: A cost-effective evaluation method for use by designers 35(6), 891–912 (1991)



# Factoring User Experience into the Design of Ambient and Mobile Systems

Michael D. Harrison, Christian Kray, Zhiyu Sun, and Huqiu Zhang

Informatics Research Institute, Newcastle University, NE1 7RU, UK

Michael.Harrison@ncl.ac.uk

**Abstract.** The engineering of ubiquitous computing systems provides important challenges. Not least among these is the need to understand how to implement designs that create a required experience for users. The paper explores a particular class of such systems for built environments. In particular it is concerned with the capture of experience requirements and production of prototypes that create experience. The aim is to develop methods and tools for such environments to enable the creation of particular sorts of experience in users. An approach that combines the use of scenarios, personae and snapshots with the use of prototypes and models is described. The technique aims to elicit an understanding of the required experience of the system and then create a design that satisfies the requirements.

## 1 Introduction

While a wide variety of experimental ubiquitous computing systems have been developed in recent years, relatively little effort has been aimed at the problems of engineering the interaction of these systems. This paper addresses a class of such systems that involve public displays, hand held devices and location sensors. The systems that are of interest may be used to deploy services to users of built environments (office, leisure complex, hospital, airport or museum). Such systems enhance the user's experience of the environment by offering information about it and the services available within it. The systems envisaged here are always on in the background, and provide services to the user according to their context and location.

The success of these systems depends on a number of factors, including software and hardware reliability and usability. The user's experience of these systems is particularly important but what experiencing a system in a particular way might mean is difficult to express and then to implement in a system. Examples of experience in a built environment might include: *place* (feeling that you know where things are); *absence of anxiety*; *safety* or *security*.

These experiences can be valuable in making the environment more attractive to users. They can also enhance in users an awareness of issues such as safety or security and therefore modify their behavior. Weiser and Brown [27] in early discussions of ubiquity highlighted the importance of experience when they used the term "calm technology". Their vision of ubiquitous systems was that it would create an experience akin to lack of anxiety and feeling in control. Forlizzi and Battarbee [10] go

further and make distinctions between types of experience relating to “fluent”, “cognitive” and “expressive” interactions. The issue to be addressed is how to elicit, model and implement these experience requirements. A particular concern is what role that formal modeling could play in such a process.

Many factors affect the experience that users have of built environments. These include the texture and physical characteristics of the environment and where information displays are situated. The paper uses the same example throughout which is based on an airport. In each space within the airport there is a public display that displays messages about flights that are relevant to passengers that occupy the space at any one time. Each passenger carries a mobile phone and receives messages on this phone that are specifically relevant to their flight and location. The design deliberately adopts a simple set of techniques for deploying information to users. It should be noted from the outset that many schemes are feasible for combining public displays with private information, see for example [13,16]. The scheme used here is illustrative of a range that could equally be addressed by the techniques described. A prototype is described as well as a formal model used to explore experience requirements and the creation of designs producing the required experience for users.

The structure of the paper is as follows. Section 2 introduces the main issues associated with experience relevant to the paper. Section 3 discusses issues of experience elicitation and proposes a set of feasible experience properties of the example. Section 4 comments on experience articulation. It identifies the problems associated with expressing an experience requirement so that an engineer can use it to produce a design. Section 5 discusses experience prototyping. It describes prototypes that were developed as a basis for exploring features of the airport environment. Section 6 describes a specific model of the airport system. It discusses the role that modelling and analysis techniques might play.

## 2 Factoring in Experience

Before building a system that creates a given experience it is necessary to understand what experience is appropriate. It is then necessary to express the experience in a form that can be used by designers and engineers. It is only possible to be sure of the experience that is created in a design when the system is in-situ in its proposed setting. However it is usually infeasible to explore the role of a prototype system in this way, particularly when failure of the system might have safety or commercial consequences. A prototype running in a busy airport will have unacceptable consequences if it fails. It may have safety or commercial consequences if crucial information is not provided clearly in a timely way. At the same time, deploying a system that is close to product when many downstream design commitments have already been made will be expensive to redesign. Exploring and assessing usability and experience of prototypes, however close to product, in its target environment is therefore unlikely to be acceptable or cost effective. Techniques are required to enable early system evaluation.

Once an experience has been understood, it should be expressed in a form that supports construction of an environment that creates the experience. This paper addresses a number of questions. How are the experience requirements for such a system established? How are they articulated so that a system can be designed to implement them?

How can models or prototypes be used to check whether the required experiences are created in the design before committing to a final implementation?

The paper explores available methods for experience elicitation, noting the role that scenarios play not only in capturing features of an experience but also providing a basis for visualizing what a proposed new design would be like in the context of that experience. The paper also explores the role that snapshot experiences play in deriving properties that can be applied to models of the proposed design. Snapshot experiences can also be used to inspire or to construct further scenarios that can also be explored as a basis for visualization.

### 3 Experience Elicitation

McCarthy and Wright [22] and Bannon [2] have argued that while GUIs lead to an emphasis on technology as tools, systems such as those described in this paper require thought about how people live with the technology. This change has also been described as a shift from understanding use to understanding presence [15]. Existing methods of user-centered design do not help engineers understand which designs are likely to lead to feelings of resistance, engagement, identification, disorientation, and dislocation amongst users.

Experience can be understood through a variety of mechanisms. It can be understood through narrative by:

- Asking people to tell stories about experiences they have had of an existing system.
- Exploring alternative worlds in which the experience would have been different.

Many authors (for example [14]) discuss the use of scenarios based on these narratives. Personae are also used as a filter for understanding the scope of experience requirements.

Scenarios alone are not sufficient to provide clear experience requirements for which the route to implementation is clear. They may be biased towards the current system. They may lead unacceptably to a proposed solution that fails to capitalize on the opportunities that the new technology affords and is instead an incremental development of the old one. The collection of scenarios and personae are unlikely to be sufficiently inclusive to provide a complete picture of the experience of the system. However, scenarios provide rich descriptions that are extremely valuable in the experience elicitation process. At the same time scenarios provide a medium that can later be used with proposed paper designs or prototypes to “visualize” effectively what the design requirements are.

Other techniques are required to complement scenario orientated techniques. It is necessary to augment some of the limitations of scenarios to obtain a richer understanding of what experience is required. Cultural probes provide an orthogonal perspective [12]. They can be used to elicit snapshot experiences. These are understood as fragments of experience provided by users that can be used to help understand how users experience an existing system. The aim is that these snapshots should be used to establish what is required of a new design. Eliciting snapshots involves subjects

collecting material: photographs, notes, sound recordings, that they believe capture important features of their environment. These snippets may make sense as part of a story. The information gleaned may help understand characteristics of the current system that cut across a range of scenarios. In the example (see Section 1) the purpose of the ambient and mobile system is to notify passengers about the status of their flights, wherever they are in their passenger journey. Passengers might be asked to identify snapshot experiences of the existing airport environment. They may be invited to take photographs or make audio-video recordings and to produce commentaries or annotations of these snapshots explaining why the snapshots are important. The following are plausible examples:

- **S1:** photographs of the main display board with comments such as:
  - “I like to be in a seat in which I can see this display board”;
  - “I wish that the display board would tell me something about my flight - it disturbs me when it simply says wait in lounge”;
  - “How can I be sure that it is up-to-date?”;
- **S2:** photographs of signposts to the departure gate with annotations such as: “I wish I had better information about how far it was and whether there were likely to be any delays on the way”;
- **S3:** tape recordings of helpful announcements and tape recordings of unhelpful announcements, with annotations such as “These announcements do not happen often enough and announcements for other flights distract me”;

This information requires organization to ensure that subsets of facilities are not neglected. Snapshot experiences may be used to trigger further narratives. The analyst might enquire of a user who has generated a snapshot: “Can you think of situations where this particular feature has been important?” By these means they may inspire a scenario that would not otherwise have been gathered. They can also be converted into properties that the new design should satisfy. Hence the comment relating to S1: “How can I be sure it is up-to-date” could lead to a number of properties:

- **P1:** when the passenger moves into the location then flight status information is presented to the passenger's hand-held device within 30 seconds
- **P2:** information on public displays should reflect the current state of the system within a time granularity of 30 seconds

In future sections these properties are considered in more detail.

## 4 Experience Articulation

As discussed in the previous section, scenarios and snapshots together capture experience characteristics of the system. The question is how this information can be used along with prototypes and models to produce an implementation that can create the desired experience. Experience provides an imprecise basis for implementation requirements. It becomes necessary to explore the proposed system experimentally: “I will know what it is when I have got it”. Buchenau and Suri [6] describe a process of probing using scenarios and approximate prototypes. For example their method might involve asking people to carry dummy devices around with them to visualize how it

would feel. Their approach (“experience centred design”) enables imagination of the experience that users would have with the design. The quality and detail tends to vary: from “mocking up”, using prototypes that simply look like the proposed device but have no function, to more detailed prototypes that are closer to the final system. The design intended to create the experience emerges through a process of iteration. Articulation of the required experience is encapsulated in the design that emerges from the process. To explore and to visualize the proposed design effectively it is important that prototypes can be developed with agility. It should be possible to try out ideas and to dispose of prototypes that are not effective. It should be possible to use a context that is close to the proposed target environment. These early prototypes help envision the role of the “to-be-developed” artefact within the user’s activity. Prototypes can also be used to “probe” or to explore how valid and representative the scenarios are. This can be used as a basis for generating a discussion about alternative or additional scenarios.

Snapshot experiences can be a valuable aid to analysts. They can form the basis for properties that the system should satisfy. The conversion from snapshots to properties relies on the experience and practice of the analyst. Such properties should be independent of specific implementation details. Whereas scenarios can be explored with prototypes, properties require the means to explore the design exhaustively. This can be done, as in heuristic evaluation, through the expertise of a team of analysts exploring a description of the design systematically. It can also be automated through model checking as will be discussed in a later section. The same model that is appropriate for experience requirements checking can be used to analyze other properties that relate to the integrity and correctness of the system.

- **P3:** when the passenger enters a new location, the sensor detects the passenger’s presence and the next message received concerns flight information and updates the passenger’s hand-held device with information relevant to the passenger’s position and stage in the embarkation process.
- **P4:** when the passenger moves into a new location then if the passenger is the first from that flight to enter, public displays in the location are updated to include this flight information
- **P5:** when the last passenger on a particular flight in the location leaves it then the public display is updated to remove this flight information

## 5 A Stimulus for Experience Recognition

The physical characteristics of the environment in which the proposed system is embedded are critical to an understanding of the experience that the system will create for users. These characteristics might include the texture of the environment, ambient light and color, the positioning of public displays, the activities that passengers are engaged in (for example pushing luggage trolleys) and how this intrudes on their ability to use mobile phones and look at public displays. Given the potential cost of premature commitment to systems in the target environment how can scenarios and snapshot experiences be used earlier in the development process as a means of understanding the experience that users might have with a proposed design?

## 5.1 The Role of Scenarios

Walkthrough techniques such as cognitive walkthrough [18] can be applied to a proposed design in the early stages of the design development. These techniques require sufficient detailed scenario narratives to make it possible to analyze individual actions. In the context of the airport, analyzing actions would involve assessing how effectively the displays and mobile phones resource the actions described in the scenario. Similarly, walkthrough techniques may be used to explore the experience of a proposed system if the analyst can use the scenario to visualize in sufficient detail what the system would “feel like” in its proposed setting. The problem with this approach is that it depends on the imagination of the analyst – what would it really feel like for a particular persona [14], perhaps a frequent flyer who is nevertheless an anxious traveler, to be involved in this story with the proposed design embedded in a given airport. The advantage of using such visualization techniques is that they can be used at very early design stages. A further development would be to ask potential users to visualize scenarios in the context of a description of the proposed design, perhaps using mock-ups of the displays or very approximate, perhaps non-functional, artifacts to help them visualize the scenario in the proposed target environment [6]. Here they would imagine the scenario, perhaps sitting in a meeting room, but would be able to hold or see some of the proposed artifacts that are designed to be embedded in the intended environment. Such a visualization approach is not concerned with the details of the actions involved in the scenarios, rather it would provide an impression of aspects that require further analysis.

Providing an environment in which a “passenger-to-be” can envisage the experience of the proposed technology would involve transplanting working prototypes either to a different context or to simulate the proposed context. For example, some of the features associated with the proposed system are similar to a system designed to provide office commuters with train departure information. To explore this analogy a large display was sited in a common area in the office, and a database was created containing information about workers’ railway tickets. A blue-tooth sensor detected the presence of enabled mobile phones in the common area. Relevant information about the departure times of the next few trains was displayed for those people who were in the common room who had railway tickets and were registered with enabled phones. Particular train information was removed from the display when the last commuter for whom the train was relevant left the common room. The system was developed using publish subscribe middleware, by scraping the train destination information from [www.livedepartureboards.co.uk](http://www.livedepartureboards.co.uk). It was then possible to explore how users would experience this environment by configuring their mobile phones appropriately for the trains for which they had tickets and exploring how well the system worked in various situations. The question that such an activity raises is whether much can be learned about the experience of office workers using this system that can be transferred to the airport environment.

In reality the two contexts are very different and therefore the experience is likely to be very different. Office workers move out of their workspace to the common area for a cup of coffee or specifically to see whether their preferred train is currently on time. For an air traveler the primary purpose of being in the airport is to travel. They may be working at their laptops or making phone calls but these are secondary activities. Only the most general usability considerations can be addressed at issues associated with the stability of the display and the way the display is updated.



DESTINATION	TIMETABLE	EXPECTED	OPERATOR
Manchester Airport	11:08	Starts here	First TransPennine Express
London Kings Cross	11:30	Starts here	GNER
Edinburgh	11:40	On time	Virgin Trains
Penzance via Leeds	11:40	11:55	Virgin Trains
London Kings Cross	11:57	11:58	GNER
Glasgow Central	12:05	On time	GNER
Manchester Airport	12:08	Starts here	First TransPennine Express
Plymouth via Doncaster	12:19	Starts here	Virgin Trains
London Kings Cross	12:34	On time	GNER
Bournemouth via Leeds	12:40	No report	Virgin Trains

**Fig. 1.** The real train departure display

It is clear that prototyping a similar system (the train information system) in a different setting is not likely to provide much useful information about the experience of the airport. Another possible solution is to explore a simulated environment for the prototype system. A virtual environment was created that bore some resemblance to the office space within a CAVE environment (an alternative that was not explored was to consider the virtual environment on a desk-top to stimulate the experience of the office system with the public display). The departure information was displayed in a virtual room using a virtual display located on one of the walls in the room. The basis of the proposed target system: the sensor software, the use of the publish-subscribe middleware, was the same as the implemented system but it provided a virtual display, and a virtual sensor was triggered by the presence of a real mobile phone in the virtual common room (Figure 2).

There were a number of problems with this approach. Though it had the effect of creating some of the features of the proposed real world it lacked textural realism. In reality common rooms contain people as well as the bustle and noise of these people and their activities. These issues could be crucial to an assessment of the appropriate experience. The CAVE environment made it possible for potential users to explore the virtual space and to see the display from various angles as they would if they were in the real world. To achieve this exploration “natural” mechanisms for navigation around the space are required. A wand was used for navigation in the prototype. In practice this was not satisfactory because it adds encumbrance to the user, potentially interfering with their use of the mobile phone. An alternative approach, currently under exploration is to use body movement as a means of navigation. Another problem with these techniques is that they can provoke nausea in the subject. Simulation sickness can entirely affect the experience of the user in the virtual environment in such a way that its value as a means of exploring experience requirements is compromised.

An alternative approach that would be effective to overcome some of these problems and create an improved simulation of the experience that a user would have is described in [26]. Here “immersive video” is used as a means of exploring features of the design

of a system. Their approach uses a video of the existing environment that has been treated using computer enhancement to create the artifacts (for example the public displays) that are proposed. The film represents the scenario. At stages in the scenario the appropriate triggers are generated to modify the subject’s mobile phone. The advantage of this technique is that it provides a richer environment with better atmospheric texture including ambient sound and the movement of other people. The approach is called immersive video because all three sides of the CAVE contain video perspectives, though the film is not stereoscopic. The problem with the approach is that the exploration is limited to a fixed sequence. Users have some interaction capabilities with the immersive video and they have more limited means to explore the virtual world that has been created. The filmed scenario constrains where they can move.

A combination of these approaches promises to provide useful feedback on user experience before deployment of the completed system.

### 5.2 The Role of the Snapshots

The information that is gathered through snapshot experiences can be used by the analyst to elicit further scenarios. Hence a snapshot can be used as basis for visualizing the experience that the proposed design would create. Alternatively, as illustrated through S1, the comments that are associated with the snapshots can be used as a basis for discovering properties that the design should satisfy such as P1-P2. These properties can be used systematically but informally in the way that usability heuristics [24] are used. Usability inspection techniques typically involve a team of analysts

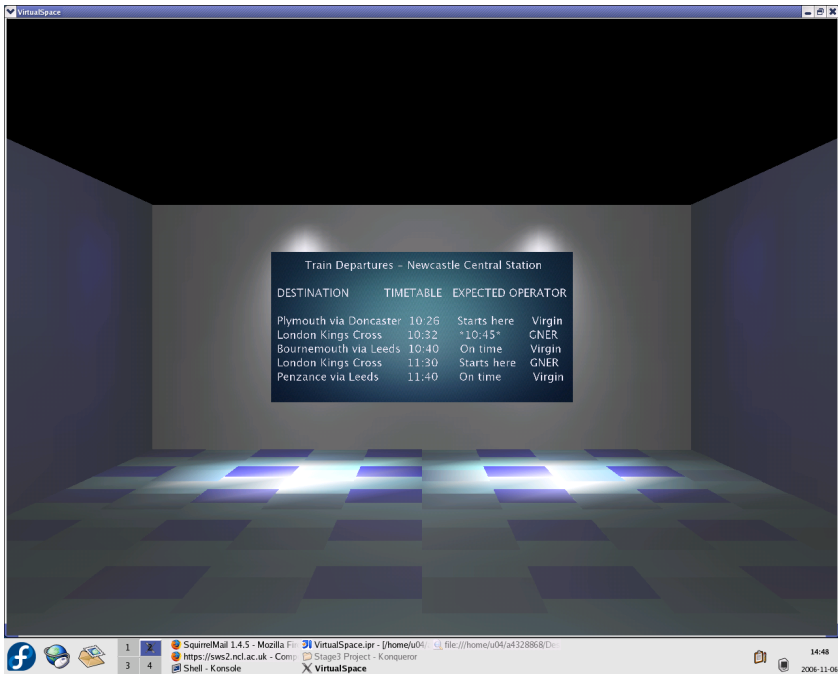


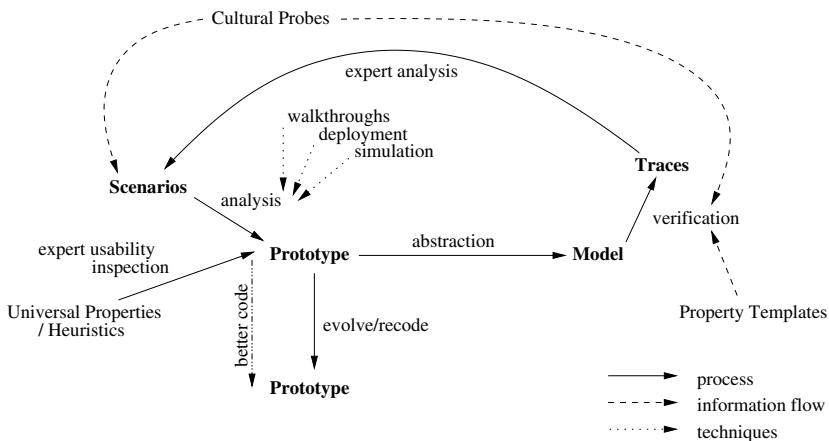
Fig. 2. Virtual display of train departure information



to question the representation of the design. Alternatively, these properties may be made formal and applied to a model of the proposed design as a further stage of the analysis. While satisfaction of the properties is the goal of this formal modeling, counter-examples where the properties fail may provide valuable information that can be used as the basis for further scenarios.

## 6 Modeling the System

Snapshot experiences can be converted into properties to be used as a complement to scenario driven design. Instead of visualizing the design through the scenario, the model of the system is checked to ensure that the property holds. The approach comes full circle when sequences of states of the model that are generated as a result of properties not being true are themselves used as the basis for further scenarios. Campos, Harrison and Loer [7,20] have explored techniques for using properties to analyze models of interactive systems in the context of usability analysis, in particular the mode complexity of the design. They use model checking techniques to discover whether a property is true in general of the proposed model or to find counter examples that do not satisfy these properties. Model checkers typically generate sequences of states of the model as counter examples. Domain experts can use a bare sequence of states to create a plausible narrative to form the basis for a scenario. This scenario can then be used to visualize the design as described in Section 5.1. This process that combines models, prototypes, snapshot experiences, properties, traces and scenarios is depicted in Figure 3. The figure reflects an iterative process in which models and prototypes are developed in parallel keeping models and prototypes consistent.



**Fig. 3.** The formal process of experience requirements exploration

It is possible to enter the diagram from a usability perspective or from a system modeling perspective. Traces are used by specialists to construct scenarios on one side of the diagram and these scenarios are evaluated using prototypes. On the other side of the diagram properties are derived from snapshot experiences and these

properties are used to check models of the system. The diagram suggests the variety of evaluation techniques that can be applied to the scenarios and the prototypes.

It is envisaged that a variety of models may be developed to check the properties of the system. The airport model described in this section reflects preoccupations surrounding properties P1-P5 which were in turn based on snapshot experience S1. They focus on timing related properties. An alternative model could have been created to explore possible physical paths in the environment. This model could have been used to analyze properties relating to the snapshot experience S1: "I like to be in a seat in which I can see this display board" and to the snapshot experience S2: "I wish I had better information about how far it was and whether there were likely to be any delays". Loer and Harrison [19] include location in a model of a plant involving pipes, pumps and valves. They use this model to explore the control of the plant by a hand-held PDA and the potential confusions that arise as a result of location. It is envisaged that a similar model to [19] which employs SMV [23] could be used to model locational characteristics of the airport.

Alternatively a model could be developed to address stochastic properties of the proposed design to address further properties using techniques such as those described by [9, 17]. Examples of properties that might be explored using such models are:

- **P6:** any service that is offered to a subscriber will only be offered if there is a high probability that there is enough time to do something about the service
- **P7:** the message is most likely to be the next message

P6 may have arisen as a result of a comment: "What is the use of being told about a service if there is no time to benefit from the service before the flight departs". P7 on the other hand could be a property generated by the engineer as a compromise, recognizing that the user requirement that it should be guaranteed to be the next message cannot be satisfied in practice.

It is envisaged that generic models could be developed to make the process of construction of models easier. The airport model shares generic characteristics with other ubiquitous systems designs to deploy information about services in built environments (for example systems involving rooms, public displays and sensors). Such an approach is already being used in the analysis of publish-subscribe protocols [3, 11]. The properties may also be based on property templates that are generic forms of frequently occurring snapshot experiences. These templates could be supported in a way that is similar to that described in [20] in the context of usability properties. The challenge is to develop a model at an appropriate level without unnecessarily biasing the design consideration. The models should allow a proper formulation and application of appropriate properties.

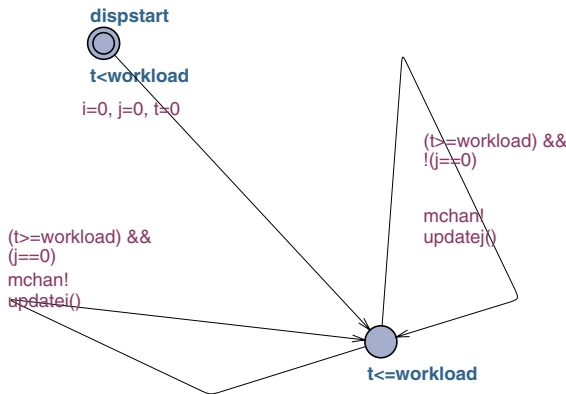
Traces, providing the basis for scenarios, are important in the investigation of experience requirements. However some properties, for example those that relate to quantifiable aspects of the design cannot produce meaningful scenarios. Consider the following:

- **P8:** no matter how many services a user is subscribed to, the flight information service will be dispatched both to the user's device and to the local display within a defined time interval.

## 6.1 Characteristics of the Airport Model

The model captures the timing behavior of the airport system. It follows previous work [21] on timing aspects of scheduling in a dynamic control system (a paint shop) using `uppaal` [5] to model the interactive system. The airport model contains a process that describes the activity within a room, including the mechanism for sensing the arrival and departure of passengers. This process updates the room based display to show flight information for those passengers that are in the room. A further process describes the passenger that receives specific messages relating to flight and location in the airport. The passenger moves from room to room. There is also a process that dispatches messages regularly. In what follows a more detailed description of the system will be given.

In the `uppaal` diagrams that follow, circles represent states. States can be named (for example `dispstart`) and can be identified as initial states by two concentric circles. Arcs between states represent possible transitions that can occur. Transitions may be guarded. Hence in Figure 4 one transition from the un-named state can only occur if the clock  $t$  is greater than or equal to a value defined by the constant `workload` and the variable  $j$  is non zero. An arc can specify a communication. Hence `mchan!` is an output signal representing the next message to be sent to waiting processes. This transition can only proceed if there is a process waiting to receive using `mchan?`. A transition can also specify that the state is to be updated.



**Fig. 4.** The dispatcher process

Hence in the arc from `dispstart`,  $i=0$ ,  $j=0$ ,  $t=0$  specifies that variables  $i$  and  $j$  are set to 0 and the clock  $t$  is also set to 0. Finally, functions may be used to specify more complex updates. In general, for reasons of space, these functions will not be described in detail. In the case of the process of Figure 4, `updatei()` and `updatej()` are functions that among other things update  $i$  and  $j$  respectively.

The dispatcher (Figure 4) is critical to the timing characteristics of the design, and alternatives be explored by the designers to create a system that satisfies the required properties. This would involve adjusting the rate and the order of distribution of messages. Alternative dispatchers taking account of passenger arrival volumes should also

be considered. The example in figure 4 distributes messages in strict order. Messages relevant to flight and location are sent in sequence. The next message is sent every time interval. The rate of distribution (the variable `workload`) can be adjusted to assess the properties of different rates of distribution. In this process the variable `i` (describing the flight number) is updated when `j` (the location value) returns to zero.

Two types of process receive information from the dispatcher. The sensor process (Figure 5) combines the behavior of the public display with the room sensor. The passenger process (Figure 6) describes the passenger and the relevant behavior of the passenger's mobile phone and the mobile device respectively. In the model that was analyzed a sensor was instantiated for each room of the fictional airport (entry hall, queue1, queue 2, check in, main hall, gate). The aim was to ensure that these processes model the key interaction characteristics that are required of the proposed system design insofar as they relate to the properties P1-P5.

The sensor process (Figure 5) describes the key interaction features of:

- the public display located in the room
- the sensor that recognizes the entry and exit of passengers – this assumes an interaction between the sensor and the passenger device

The sensor communicates by means of three channels.

- It receives messages that have been distributed to it from the dispatcher by means of the channel `mchan`.
- It receives requests from the passengers' hand held devices (via `arrive`) where they arrive in the room that relates to the sensor
- It receives requests from the passengers' handheld devices (via `depart`) when they leave the sensor's room.

When the sensor receives a message from the dispatcher, the function `read()` checks the tags on the message and if the location tag coincides with the location of the sensor then the display is updated. Of course a realistic implementation of this system would update a flight information array for display each time a relevant message is received. The array updating mechanism is not of interest to interaction analysis. When the sensor receives a message from the `arrive` channel this signals the entry of a passenger. The array `present[]` keeps a count of the number of passengers present for a particular flight and is incremented with the arriving passenger's flight number. When the sensor receives a message from the `depart` channel then the array is decremented using the departing passenger's flight number. If the result of this is that there are no passengers for a particular flight left in the room then the flight information is removed from the display. In the event that the last passenger moves out of the space the display is cleared. When the passenger is newly arrived in the space then the array `present` is incremented and so next time a message arrives about this flight the information will be displayed for the first time.

The passenger process (Figure 6) describes the activity of the passenger and the key features of their mobile phone. This activity has a number of characteristics:

- The passenger is given a specific path to follow. This is defined in the array `path`.
- The process notifies the room sensor that it has `arrived`. The passenger ticket is updated to point to the current location.

- The passenger moves to a state where it receives messages from the dispatcher via `mchan`. If the received message is tagged with the passenger's current location and the passenger's flight number then the mobile phone display is updated.

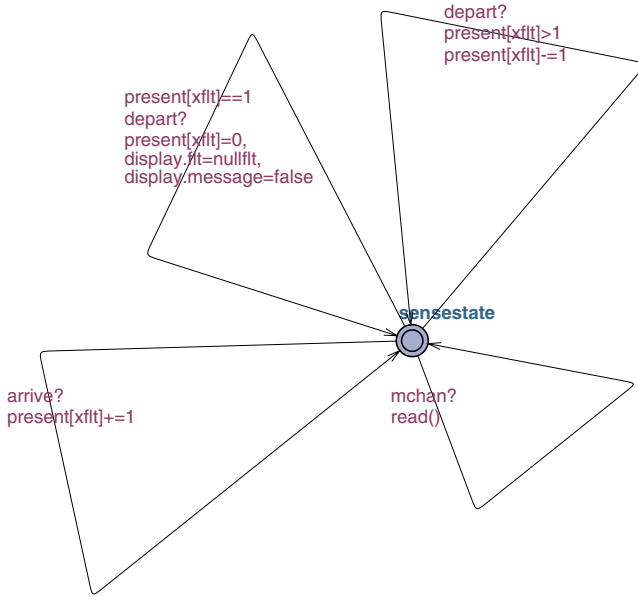


Fig. 5. The sensor process

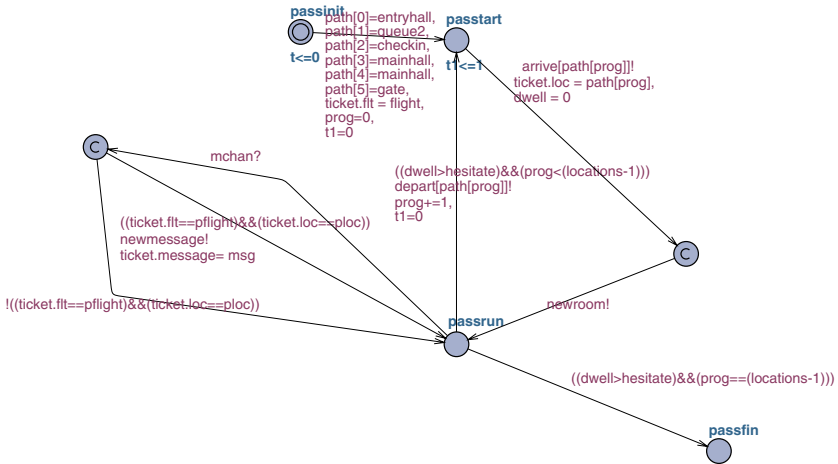


Fig. 6. The passenger process

This completes the description of the model. The next stage is to prove properties of the model.

### 6.2 Checking the Properties

The model captures those features of the airport system that relate to properties P1-P5. Space only permits a limited description of the analysis of the system.

P1 requires that “when the passenger moves into a new location in the airport then flight status information is presented to the passenger’s hand-held device within 30 seconds.” In fact P2 is a property that can be checked in the same way but relates to the sensor rather than the passenger. It must be updated within a period of delay after a passenger arrives. P1 can be characterized as proving that from the point that the passenger enters the location (regardless of flight number) the relevant message will be received by the passenger. Two transitions are of interest in the passenger process (Figure 6). The first occurs as the passenger moves into the new location and the second occurs when the passenger receives a message from the dispatcher that matches the flight number and location of the passenger. This property is checked by introducing an observer process (Figure 7) and adding a communication (*newroom*) in the passenger process (Figure 6) to signal arrival in the new location and similarly a communication (*newmessage*) to signal receiving a relevant message. If the message does not arrive while the passenger is in a location (this time is determined by the variable *dwell*) then the observer will deadlock. Given that *dwell* is the required time interval and the processes accurately reflect temporal aspects of message distributions, deadlock checking can be used to check P1 and P2. When appropriate diagnostics are switched on deadlock generates a trace that can then be further analyzed to work out why the system does not satisfy the properties.

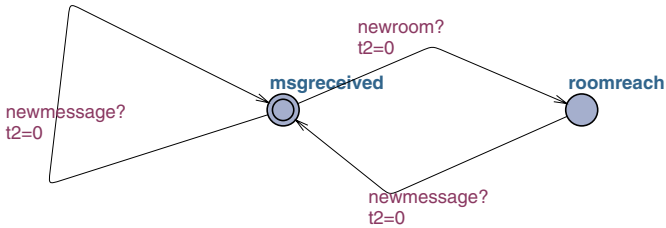


Fig. 7. The observer

In practice the generalized deadlock property is very compute intensive and on a no-frills specification PC the uppaal system (UPPAAL 4.0.0 (rev. 1900), May 2006 see <http://www.uppaal.com/>) ran out of memory after three hours execution. Alternative, more specific properties relevant to P1 and P2 were checked within a minute. For example `A[] (o1.roomreach imply (o1.t2 < maxdelay))` was checked for different values of *maxdelay*. This property holds true as long as passenger 1 (this passenger’s observer is *o1*) receives a message within *maxdelay* after entering any new room. The airport system that was used for analysis contained two instantiations of the passenger process. As a further elaboration, to check that the passenger received

regular updates while occupying a particular room, `A[] (o1.msgreceived imply (o1.t2<maxdelay))` was checked. This property checks whether subsequent messages that are received, while the passenger 1 is in a particular space, arrive at intervals of more than `maxdelay`. This property failed for an appropriate value of `maxdelay` though successful because the passenger had completed its path through the airport and terminated in `passfin`. While the observer `o1` continued to wait expecting a further signal from the passenger to say it had received another message the observer's local clock `t2` exceeded the `maxdelay` limit.

## 7 Conclusions

The models illustrated in this paper, taken together with the prototypes that were developed to explore some of the concepts in a virtual environment, have enabled the exploration of experience properties. These techniques can provide early warning of ways in which the system will not create the experience that is the purpose of the design. The model can be used to demonstrate that experience properties (derived from snapshots) are satisfied or fail to be satisfied in specific situations. These situations can be used as a basis for scenarios, used creatively to give early valuable feedback. The paper aims to set these formal processes in the context of other engineering processes that provide early visualization of the design either relying on the user's (or analyst's) imagination or using prototypes in simulated contexts that capture some of the texture of the proposed target environment.

While the proposed approach focuses on the individual's relationship with their environment, it is clear that social aspects of experience are crucial to the success of a design. In the context of the method proposed here these social aspects are captured through the probing of individuals, however it would be envisaged that social modeling would provide additional clarity about how human human interactions contribute to experience and can be supported by the ambient systems. Further techniques would be appropriate for identifying such requirements as discussed in [4]. These considerations are left for future work.

If ubiquitous computing is to become a robust feature of everyday life then engineering techniques such as those described here are required. These techniques will be particularly valuable if it becomes possible to develop generic models for classes of ambient and mobile systems in the style discussed in the context of analysis of publish subscribe systems [3, 11]. In the same way template properties should be developed that frequently occur in experience evaluations and can be instantiated to the specific circumstances of the system being developed. Finally it is to be envisaged that models and prototypes can be developed in synchrony using the style hinted at in [25] and thereby provide coordination between formal models and agile prototypes [1].

## Acknowledgement

We thank Jose Creissac Campos for valuable input during the preparation of this paper.

## References

- [1] Agile working group. The agile manifesto (2004), <http://agilemanifesto.org>
- [2] Bannon, L.: A human-centred perspective on interaction design. In: Pirhonen, A., Isomäki, H., Roast, C., Saariluoma, P. (eds.) *Future Interaction Design*, pp. 31–52. Springer, Heidelberg (2005)
- [3] Baresi, L., Ghezzi, C., Zanolin, L.: Modeling and validation of publish / subscribe architectures. In: Beydeda, S., Gruhn, V. (eds.) *Testing Commercial-off-the-shelf Components and Systems*, pp. 273–292. Springer, Heidelberg (2005)
- [4] Battarbee, K., Koskinen, I.: Co-experience: user experience as interaction. *CoDesign* 1(1), 5–18 (2005)
- [5] Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In: Bernardo, M., Corradini, F. (eds.) *SFM-RT 2004*. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
- [6] Buchenau, M., Suri, J.: Experience prototyping. In: *Proceedings Designing Interactive Systems (DIS 2000)*, pp. 424–433. ACM Press, New York (2000)
- [7] Campos, J.C., Harrison, M.D.: Model checking interactor specifications. *Automated Software Engineering* 8, 275–310 (2001)
- [8] De Nicola, R., Latella, D., Massink, M.: Formal modelling and quantitative analysis of KLAIM-based mobile systems. In: Haddad, H., Liebrock, L., Omicini, A., Wainwright, R., Palakal, M., Wilds, M., Clausen, H. (eds.) *Applied Computing 2005: Proceedings of the 20th Annual ACM Symposium on Applied Computing*, pp. 428–435 (2005)
- [9] Doherty, G., Massink, M., Faconti, G.: Using hybrid automata to support human factors analysis in a critical system. *Journal of Formal Methods in System Design* 19(2), 143–164 (2001)
- [10] Forlizzi, J., Battarbee, K.: Understanding experience in interactive systems. In: *Designing Interactive Systems (DIS 2004)*, pp. 261–268. ACM Press, Cambridge (2004)
- [11] Garlan, D., Khersonsky, S., Kim, J.: Model checking publish-subscribe systems. In: *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN 2003)*, Portland, Oregon (2003)
- [12] Gaver, W., Dunne, T., Pacenti, E.: Design: cultural probes. *ACM Interactions* 6(1), 21–29 (1999)
- [13] Gilroy, S.W., Olivier, P.L., Cao, H., Jackson, D.G., Kray, C., Lin, D.: Cross Board: Crossmodal Access of Dense Public Displays. In: *International Workshop on Multimodal and Pervasive Services (MAPS 2006)*, Lyon, France (2006)
- [14] Grudin, J., Pruitt, J.: Personas, participatory design and product development: an infrastructure for engagement. In: *Proceedings PDC 2002*, pp. 144–161 (2002)
- [15] Halnass, L., Redstrom, J.: From use to presence: on the expressions and aesthetics of everyday computational things. *ACM Transactions on Computer-Human Interaction* 9(2), 106–124 (2002)
- [16] Kray, C., Kortuem, G., Krueger, A.: Adaptive navigation support with public displays. In: Amant, St., R., Riedl, J., Jameson, A. (eds.) *Proceedings of IUI 2005*, pp. 326–328. ACM Press, New York (2005)
- [17] Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Field, T., Harrison, P.G., Bradley, J., Harder, U. (eds.) *TOOLS 2002*. LNCS, vol. 2324. Springer, Heidelberg (2002)
- [18] Lewis, C., Polson, P., Wharton, C., Rieman, J.: Testing a walkthrough methodology for theory based design of walk-up-and-use interfaces. In: Chew, Whiteside (eds.) *ACM-CHI 1990*, pp. 235–242. Addison-Wesley, Reading (1990)



- [19] Loer, K., Harrison, M.D.: Analysing user confusion in context aware mobile applications. In: Costabile, M.F., Paternó, F. (eds.) INTERACT 2005. LNCS, vol. 3585, pp. 184–197. Springer, Heidelberg (2005)
- [20] Loer, K., Harrison, M.: An integrated framework for the analysis of dependable interactive systems (ifadis): its tool support and evaluation. *Automated Software Engineering* 13(4), 469–496 (2006)
- [21] Loer, K., Hildebrandt, M., Harrison, M.D.: Analysing dynamic function scheduling decisions. In: Johnson, C., Palanque, P. (eds.) *Human Error, Safety and Systems Development*, pp. 45–60. Kluwer Academic, Dordrecht (2004)
- [22] McCarthy, J., Wright, P.C.: *Technology as Experience*. MIT Press, Cambridge (2004)
- [23] McMillan, K.: *Symbolic model checking*. Kluwer, Dordrecht (1993)
- [24] Nielsen, J.: Finding usability problems through heuristic evaluation. In: *Proc. of ACM CHI 1992 Conference on Human Factors in Computing Systems*, pp. 249–256. ACM, New York (1992)
- [25] Niu, N., Easterbrook, S.: On the use of model checking in verification of evolving agile software frameworks: an exploratory case study. In: *MSVEIS 2005*, pp. 115–117 (2005)
- [26] Singh, P., Ha, H.N., Kwang, Z., Olivier, P., Kray, C., Blythe, P., James, P.: Immersive Video as a Rapid Prototyping and Evaluation Tool for Mobile and Ambient Applications. In: *Proceedings of Mobile HCI 2006*, Espoo, Finland, 12th-15th September (2006)
- [27] Weiser, M., Brown, J.: *Designing Calm Technology* (December 1995), <http://www.ubiq.com/hypertext/weiser/-calmtech/calmtech.htm>

# Visualisation of Personal Communication Patterns Using Mobile Phones

Bradley van Tonder and Janet Wesson

Department of Computer Science and Information Systems,  
Nelson Mandela Metropolitan University, P.O. Box 77000,  
Port Elizabeth, South Africa, 6031  
{Bradley.vanTonder, Janet.Wesson}@nmmu.ac.za

**Abstract.** Ambient displays are attractive, subtle visualisations of information. They are typically situated on the periphery of human perception, requiring minimal effort to be understood. The vast volume of communication facilitated by modern means of communication has led to research into methods of visualising this information to facilitate rapid understanding. These two research areas have, however, seldom been combined to include the use of ambient displays as visualisations of personal communication patterns. The research outlined in this paper addresses this issue by combining ambient displays and visualisation of personal communication patterns in a mobile context. This paper details the development of the *AmbiMate* system, analyses its usefulness and investigates the lessons which can be learned from its implementation in order to guide the future development of such systems.

**Keywords:** Ambient displays, visualisation, personal communication patterns, mobile devices.

## 1 Introduction

Mobile phones have developed rapidly from the primitive devices of previous decades to the advanced communication platforms they are today. Today, many people view mobile phones as their primary communication device [1]. The volume of communication taking place is often overwhelming, and users can often lose track of their own personal communication habits. One answer to this problem can be found in ambient displays [2].

Mark Weiser introduced the notion of ubiquitous computing [3], whereby computers and their associated technologies would disappear into the background and require less cognitive effort on our part. Weiser also described the concept of *Calm Technology* [4], where technologies would “*empower our periphery*”. Such technologies would be less intrusive, allowing us to focus on the tasks at hand. The user has the power to decide to focus on peripheral information when he or she wants to do so.

Ambient displays provide the perfect medium for communicating information in a manner which is not intrusive. According to Mankoff *et al.* [5], ambient displays are attractive displays of information, which are situated on the periphery of human

attention. Such displays typically make use of metaphorical visualisation techniques, to display information in a form which can easily be understood by their users. Ambient displays have previously been effectively used as a tool for visualising personal communication habits, as can be seen in the visualisation of personal email presented by Redstrom *et al.* [6]. These visualisations allow users to easily appreciate information relating to their personal communications habits, with little concentration or mental effort required.

A natural evolution of such systems is to consider the development of mobile ambient displays to visualise the personal communication data stored on mobile devices. Mobile devices provide a number of unique advantages in terms of privacy and personalisation. Indeed, with mobile phones typically situated on the periphery of their users' attention, they provide a medium well-suited to be utilised as an ambient display.

Recently, the work by Schmidt *et al.* [7], has investigated the use of ambient displays to visualise personal communication on mobile phones. This work was, however, limited to prototyping designs, and no actual systems were (as yet) implemented on mobile devices. In order to establish whether mobile phone-based ambient displays are useful for visualising personal communication patterns, it is necessary that such systems be implemented and evaluated.

This paper outlines the development and evaluation of the *AmbiMate* system, a mobile ambient display used to visualise personal communication patterns. The system was developed in order to investigate its potential usefulness. Practical issues encountered during the development of the system will also be discussed, and could provide valuable insight for the future development of such systems.

## 2 Related Work

This research integrates a number of different research areas. Research into visualisation, the use of ambient displays and studies of personal communication patterns are combined. This research seeks to establish whether the benefits of using ambient displays to visualise personal communication patterns can be extended to a mobile context.

### 2.1 Ambient Displays

The origins of ambient displays can be traced back to the birth of Ubiquitous Computing in the early 1990s. Mark Weiser and researchers at Xerox PARC foresaw a future where computers would be fully integrated into every area of our lives [3]. They suggested that in order for computers to integrate successfully into the environment, it would be necessary for them to disappear into the background, no longer requiring our full attention. One of the first devices to build on this idea was the *Dangling String* [8]. This device consisted of a plastic string hanging from the ceiling, which communicated the level of network traffic through its level of motion. It made use of the user's peripheral vision, as well as characteristic sounds, to communicate information to users without them having to devote attention to it. Further examples include the *Water Lamp* [2], which used ripples in water to denote a

variety of digital information, and the *Ambient Orb* [9], a translucent sphere which changed colour to denote stock prices.

Much of the early work into ambient displays focused on the development of physical devices and used a variety of inventive means to communicate information. The limitation of this approach was that the physical nature of these devices limited their usefulness and flexibility as a means of visualising information. Research into the use of electronic ambient displays is addressing this problem, and the full potential of ambient displays as a medium for information visualisation is becoming apparent [10].

A typical feature which distinguishes information visualisation in general, and information visualisation with regard to ambient displays, is the use of metaphors and other abstract representations of information. Ambient displays typically make use of visual metaphors rather than traditional visualisations (e.g. bar graphs or pie charts) to convey information. An excellent example of this is the *InfoCanvas* system [11]. This system visualises a number of different pieces of information through metaphors, all integrated into a single static ambient display. Information which is of personal interest to users is communicated using various elements of the display. For example, the *InfoCanvas* made use of a beach scene metaphor, with elements of the display such as shells on the beach, clouds in the sky and the height of a kite all representing different information of personal interest to the user.

Other systems, such as the *Kandinsky System* [12], put an even stronger focus on the aesthetic aspect of ambient displays. The *Kandinsky System* uses compositions of images to represent information. It takes textual information, relates this information to images, and then combines the images in a collage. A variety of techniques are used to ensure that a collage is created which is aesthetically appealing, but still communicates useful information. Other ambient displays were based on the work of artists, such as the bus schedule and weather visualisations based on the style used by the Dutch artist Piet Mondrian [6].

The *Hello.Wall* system [13] was one of the first systems to use mobile devices as ambient displays. The *Hello.Wall* system was used in conjunction with PDA-like devices called *ViewPorts*. While the *Hello.Wall* was used to publicly display information in an office environment using light patterns, it could also interface with the mobile *ViewPorts* to communicate information privately. The mobile devices could be used to decode private messages left on the *Hello.Wall*.

## 2.2 Personal Communication Patterns

Several studies have looked at the characteristics of personal communication from different perspectives. Four different approaches were identified as the most widely used in research into email visualisations [14]. Three of these are general enough to characterise personal communication research in general:

- *Temporal visualisations*: studying how communication and relationships with different contacts have changed over time.
- *Contact-based visualisations*: studying communication with various regular contacts and extracting patterns and trends.
- *Social-network visualisations*: understanding social groupings through their personal communication habits.

Email, instant messaging and Internet discussion groups have all been the subject of visualisation research falling into these three categories. Systems such as ContactMap [15] visualise email to allow users to keep track of the different contacts with whom they communicate. Other systems, such as the PeopleGarden system [16], which models online discussion group interactions, provide a more overall view of communication. In the PeopleGarden system, different users in the discussion group are visualised as flowers. The height of the flowers represents time spent logged in, with the each flower visualising the number of messages posted by the corresponding user.

Ambient displays have also been utilised as a medium for visualising personal communications data. Redstrom *et al.* [6] describe an electronic email visualisation that uses an ambient display to visualise email communication. In this ambient display, different rectangles represent different people, with the size of the rectangle reflecting how many emails that person has sent and received. The colour and position of the blocks remain the same, to allow identification of different users. The display was designed to be displayed publicly in an office environment.

Instant messaging has also been the subject of visualisation systems, such as the CrystalChat [17] system, which used strings of circles to represent messages sent to different contacts. Circles are used to represent messages sent on the instant messaging program, MSN Messenger, with colour representing the person sending the message and size visualising message length.

### 2.3 Mobile Visualisation of Personal Communication Patterns

The work of Schmidt *et al.* [7] provides some of the first research into visualising mobile communication patterns. They proposed a number of prototypes to visualise personal communication data that is typically stored on mobile devices. Information typically available in phone logs includes data relating to calls made and received, who the other party was, call duration, direction of the call and time of day. Their prototypes utilised ambient displays on mobile devices to visualise this information. Various metaphors were suggested, including an aquarium metaphor, a solar system metaphor and a flower metaphor. In the aquarium metaphor, different colour fish represent different contacts, with the size of the fish representing the total volume of communication with that contact (the larger the fish, the more contact). The direction in which the fish are swimming represents the dominant direction of communication with that contact, and the speed of the fish signifies the time elapsed since the last communication with the contact in question (the faster the fish, the shorter the time elapsed).

## 3 Design

In order to investigate the potential usefulness of mobile ambient displays in visualising communication patterns, it was necessary to implement such a system. The development of this system served two useful purposes. Firstly, it provided a tool to be used in evaluating the usefulness of the system. Secondly, the development process helped to investigate the practical implications of implementing such a system.

In order to select appropriate design metaphors to implement, a pilot study was conducted amongst potential users. Four designs were chosen from related work,

three of which were proposed as prototypes in the work of Schmidt *et al.* [7], and the fourth a variation on a design used successfully by Redstrom *et al.* [6] as an ambient display. This process resulted in the decision to implement two alternative designs, one of which visualised contacts as fish (Aquarium Design) and the other visualising contacts as flowers (Flower Design).

This section outlines the design of the *AmbiMate* system, a mobile phone-based ambient display, which utilises animated metaphors to visualise personal communication data stored on mobile phones.

### 3.1 Functional Requirements

The *AmbiMate* system not only visualises personal communication patterns through the use of animated metaphors, but also supports a range of customisation options. These include customising the display in terms of display mode (Aquarium or Flower design), contacts to be displayed, data type, time period and update interval. An option also exists to view display details, as well as system help.

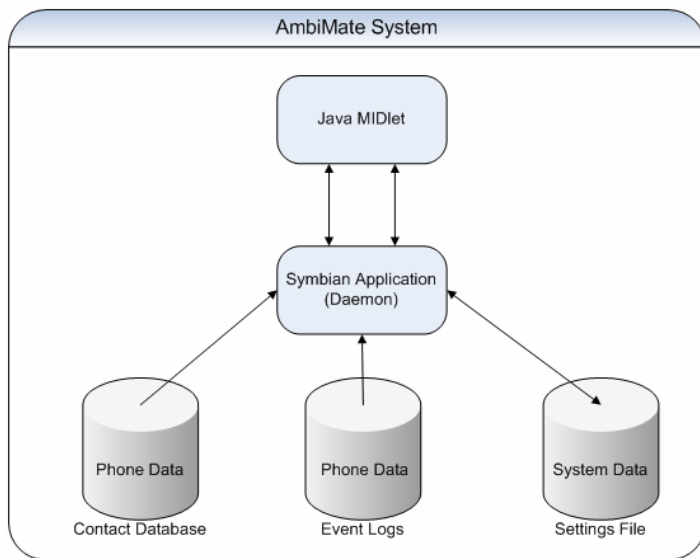
### 3.2 Implementation Tools

The architecture of the *AmbiMate* system was greatly influenced by the choice of implementation tools. In order to develop a 3<sup>rd</sup> party application with access to the communication data stored on the mobile device, it was necessary to target the so-called “smartphone” category of mobile phones. The Symbian OS is by far the market leader in this area, with approximately 67% of the smartphone market share in 2006 [18], making it the obvious selection as the target OS. This choice leaves two choices of implementation languages: Symbian C++ or Java Micro Edition (JME). Unfortunately, the Connected Limited Device Configuration (CLDC) of Java running on Symbian devices does not allow access to the necessary event log information, due to security considerations. Hence, a purely JME implementation of the system was not possible. However, Symbian C++ was considered inferior to JME for a number of reasons. These included a lack of garbage collection, inferior development tools, and greater complexity, particularly for the animation portion of the system. A purely Symbian C++ implementation would also result in a very limited set of devices on which the system is able to run. JME also provides the added advantage of a powerful, easy to use Scalable Vector Graphics (SVG) API called TinyLine2D [19].

It was therefore decided to implement a system which uses Symbian C++ in order to access the required data, with the animation and user interface implemented in JME. This provides the advantage of efficient access to the required data provided by Symbian C++, combined with the easy-to-use SVG animation power of JME.

### 3.3 Architecture

The choice of implementation tools resulted in a system consisting of two separate, but interconnected applications, as shown in Figure 1. The first application, the Symbian application, has access to data stored on the phone (contact and event information). This application then communicates with the Java MIDlet, which renders the animation based on the data it receives from the Symbian application. A settings file is used to



**Fig. 1.** System architecture of *AmbiMate* System

permanently store user preferences. When changes are made in the customisation portion of the Java application, these changes are sent to the Symbian application which updates the settings file. The Symbian application also accesses the settings file on start-up to determine which data to send to the Java application.

**3.4 Data Design**

The data needed by the *AmbiMate* system can be divided into two sections – phone data and system data. The phone data is stored on the mobile device itself. Information regarding phone calls and text messages (and any other form of communication that mobile phone users engage in) is automatically recorded in a log file by the Symbian OS when the event takes place. Information regarding contacts, such as is typically entered by users in the “Phonebook” of the mobile device, is also stored on the device.

The system data consists of user settings maintained in the settings file on the device to permanently store user preferences. Table 1 summarises this data.

**Table 1.** *AmbiMate* system data

Attribute	Description
mode	Currently selected display mode (e.g. Aquarium mode).
datePeriod	Date range currently being visualised.
eventType	The event type currently visualised (phone calls or text messages).
updateInterval	How often the display is updated to include events since last update.
numContacts	The number of contacts currently being visualised.
contactNum (for each contact)	Uniquely identifies a contact.

### 3.5 User Interface Design

Initial paper-based designs were implemented using a visual designer and run on an emulator to give a more accurate picture of the interface presented to the users. Figure 2 shows the main menu structure of the application. The significance of each menu option is summarised below:

- “Customise”, leads to a sub-menu which allows the user a variety of different customisations (Section 3.1).
- “Details” provides the user with a textual breakdown of the visualised information.
- “Start-up Settings” allows the user to customise the delay before the ambient display is activated, as it is designed to function as a screensaver-type application.
- “Help” leads to a screen giving a brief explanation of the mapping between the properties of the display and the data being visualised.

Figure 3 and Figure 4 show the two ambient display designs. Figure 3 shows the Aquarium Design, with different colour fish representing different contacts, the size of each fish size representing the volume of communication with the corresponding contact and the direction the fish are swimming indicating whether incoming or

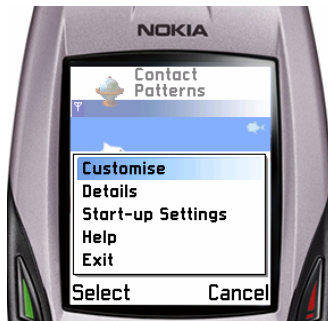


Fig. 2. Main menu structure



Fig. 3. Aquarium Design



Fig. 4. Flower Design (day-time)



outgoing communication is dominant. Figure 4 shows the Flower Design when day-time communication (6am to 6pm) is being visualised. Colour and size (height in this case) have similar meanings as in the Aquarium Design. The colour of the sky denotes whether day-time or night-time communication is being visualised. In the night-time version the sun is not visible and the sky is black.

## 4 Implementation

A Nokia 6600 was used for testing throughout the implementation phase. This phone model runs Symbian (S60) OS version 7.0 and also runs Java CLDC version 1.0 and MIDP 2.0. This phone represents the typical type of mobile device the system is aimed at, and hence was ideal for testing. Some aspects of the implementation, such as communication between the two applications, proved impossible to test on the emulators provided with the development SDK's, so device-based testing was necessary in order to test this functionality.

### 4.1 Functional Implementation

As can be seen from Figure 1, the architecture of the system relies on communication between a Symbian application and a Java MIDlet. In order to do this, an implementation built around the "MIDlet Native Services Framework", proposed by Gupta and de Jode [20] was used. A common problem with Java MIDlet development is that for security reasons, applications are unable to access native services running on the mobile device. The Native Services Framework gets around this by establishing a two-way local socket connection on a pre-defined port between a Java MIDlet and a Symbian daemon application. In this way, the Java application is able to access services it normally wouldn't be able to, such as the contact and event information needed by the *AmbiMate* system. This mechanism is used throughout the system. When the user selects customisation options in the Java application, these are sent to the Symbian application which writes them to the settings file. When the animation starts up, it first sends a message to the Symbian application, requesting that it send the information needed to render the display. When the user enters a customisation screen, the Symbian application retrieves the currently selected options and sends them to the Java application, so that the correct options are displayed. Protocols set up on either end of the communication channel ensure that traffic is kept to a minimum. The Symbian application is also responsible for all reading and writing to the settings file, which permanently stores user preferences.

The Symbian OS maintains a contact database and a log engine database, both of which contain phone data needed by the system. The Symbian daemon application provides access to this data, something which proved more challenging than originally anticipated. The original algorithm design included a step which filtered the event logs by contact, something which the Log Engine API reference showed to be possible. When this was implemented, however, it became apparent that contact details were not being recorded at the time the log events were entered, making it impossible to filter the events by contact. Instead, the filtering process had to be done manually. Different phone number formats also proved a problem, as the numbers in

the contact database were often recorded without the international dialing code prefix (e.g. +27 for South Africa). In the phone logs, however, the numbers were recorded with this prefix attached. This problem was overcome by comparing the last nine digits of the numbers in order to determine if an event corresponded to a known contact. Further complication was added because the Symbian OS contact model does not define a standard set of required attributes to be stored for each contact in the contact database. Indeed, a contact may not even have a phone number recorded at all, or may have several different numbers recorded. These include a mobile number, a home mobile number, a work mobile number, a fixed-line number, a home fixed-line number and a work fixed-line number, all of which have to be catered for, if they are present. The API reference was also not clear about which contact model fields correspond to these values.

It was also discovered that the log engine only stores log-event information for a maximum of one month. Since this is likely to represent a large enough volume of data to appreciate contact patterns and writing an application to create a custom log-file system was beyond the scope of the system, it was decided that this data was sufficient.

The log event information was grouped by contact, and split into incoming day-time communication, outgoing day-time communication, incoming night-time communication and outgoing night-time communication. This information was then sent to the Java application, along with the animation settings retrieved from the settings file, in order to facilitate the actual rendering of the animation.

## 4.2 Ambient Display

The two designs (Figures 3 and 4) that were implemented, namely the Aquarium Design and the Flower Design, provide different interpretations of the visualised data, and hence much of their functionality was implemented separately. Only the Aquarium Design visualises the directionality of the communication data, while only the Flower Design incorporates time of day (when the events took place) into the visualisation. Functionality that could be generalised was implemented in an abstract parent class, which the two animations extended.

The information received from the Symbian application was used to determine the size of the display elements relative to each other, and in the case of the Aquarium Design, their directionality. These calculations obviously had to take into account each element's size relative to the overall height (in the case of the fish) or width (in the case of the flowers) of the screen. Unfortunately, CLDC 1.0 devices, such as the Nokia 6600 on which testing took place, do not provide the standard Java floating point primitive types "float" and "double", making calculation of such ratios a problem. While this problem would not exist in later devices running CLDC 1.1 (in which floating point types were added), it was decided to try and overcome this problem. An API was found (the "MicroDouble" API), which to a degree overcame this problem by storing floating point numbers using a hexadecimal representation in a "long" primitive type. Exact calculations were still not possible, leading to round-off errors, but the format proved sufficiently accurate.

Once the display elements' size (and in the case of the Aquarium Design, directionality) had been determined, their position was calculated. This was done by

sorting the data to position the most frequent contacts (and hence the largest display elements) in the centre of the screen, and less frequent contacts on the edge of the screen.

A 2D drawing Scalable Vector Graphics (SVG) library that is not part of the standard Java Micro Edition class libraries, called “TinyLine 2D” [19], was used to create the animations. Using this 2D library, it was possible to only have a single standard fish and flower template, which could then be scaled to the necessary size for each contact. Translation and rotation transformation matrices could then be applied in order to create the animation effects. Effects such as the movement of a fish’s tail independently of the rest of its body could then be achieved more easily than would have been the case had the same effect been attempted using standard JME libraries.

In order to accommodate the operation of a mobile phone as an ambient display, it was decided to implement the system to operate similarly to a typical mobile phone screen-saver. Such an implementation would mean that the user would not have to consciously activate the display once installed, and that it would run unsupervised in the background – in essence fulfilling the requirements of an ambient display. The architecture of the system implied that a number of steps were required to achieve this goal:

- Starting the Symbian daemon application when the phone boots.
- Disabling the standard system screen-saver.
- Starting the animation automatically after the desired interval.

A number of approaches were tested to start the Symbian daemon application on boot-up. Ultimately, EZ-boot, a boot manager application provided by NewLC, was used to achieve the desired result. EZ-boot waits until the phone has completed the boot sequence, and then launches applications registered with it.

Once the daemon application is booted, it then continues running in the background, running a low-priority thread to check whether the time has elapsed after which the ambient display should be activated. Once the ambient display has been activated, it continues running a low-priority thread in the background to disable the standard system screen-saver which would otherwise obscure the ambient display.

JME provides a useful mechanism for automatically starting MIDlets, known as the Push Registry. Applications can register themselves on a particular port with the Push Registry when they are installed. When the application is not running, the system’s Application Management Software (AMS) listens for incoming connections on that port, and if it detects such a connection, starts the MIDlet registered at that port. This mechanism is used to auto-start the ambient display once the specified time interval has elapsed. At the appropriate time, the Symbian application opens a socket connection on a pre-defined port, which results in the Java MIDlet being activated.

### **4.3 Certification, Performance and Integration Issues**

MIDP 2.0 applications come in two main forms, namely trusted and untrusted applications. Untrusted applications are not signed with the digital certificate of a Certification Authority (CA), and can only access restricted API’s if the user specifically grants permission to do so. This would obviously be undesirable in an ambient display, as given the architecture of the system, the user would be constantly

prompted to authorise network access to allow communication between the Java MIDlet and the Symbian application. As a result, the MIDlet had to be signed.

In order to allow the *AmbiMate* system to integrate seamlessly into the operation of a mobile phone and continue running in the background, it is also necessary for the animation to pause when it loses focus, and resume when it regains focus. Failing to do so would result in phone performance being drastically reduced when the user attempted to perform another task, such as sending a text message. Fortunately, the standard MIDlet lifecycle provides for this, with methods to determine when the MIDlet gains and loses focus [21]. Implementing these methods allows the ambient display to remain dormant in the background while the user is busy, and then resume once the user has completed his/her task.

The issue of whether to leave the backlight on proved a particularly tricky one for this system. Typical system screensavers on devices such as the Nokia 6600 operate with the backlight off, in order to conserve battery power. However, leaving the backlight off with the ambient display active would make it difficult to appreciate the visualisation being performed. A balance needed to be found between conserving battery power, while still making the display visible. The best compromise was to leave the light on for short intervals each time the display was updated. By doing so, the user would be made aware that the display was updating, and the phone's battery would not be put under unnecessary strain.

The limited processing power of mobile devices also had to be taken into account when designing the animations. Attempting to create animations that were too detailed pushed the boundaries of the relatively limited processing power of mobile devices.

## 5 Evaluation

This section discusses the evaluation of the *AmbiMate* system, the primary goal of which was to evaluate the usefulness of the system.

### 5.1 Evaluating Ambient Displays

Ambient displays remain a relatively new research frontier in computing and standard techniques for evaluating them are still being developed [22]. Mobile phone-based displays are newer still. Many of the ambient display systems developed previously, have merely been exploratory in nature, with little or no evaluation being conducted.

Those evaluations of ambient display systems that have previously been conducted have used a variety of approaches. Some systems, such as the artistic bus schedule ambient display system described by Skog *et al.* [10], have been evaluated using a combination of field studies and user interviews. Others have been evaluated by testing users' ability to understand and/or recall the information being visualised [22].

Kaikkonen *et al.* [23], compared field studies and laboratory testing for evaluating the usability of mobile applications. They found field studies to be more time-consuming than laboratory testing, and also that field studies provided no significant benefits for evaluating the usability of mobile applications. Skog *et al.* [24] argue that only by conducting longitudinal evaluations of ambient displays can issues specific to their use in a particular environment be uncovered. Hence, a longitudinal field study allowing users to interact with the system on their own personal mobile phones in everyday situations would likely be the best means of evaluating the *AmbiMate* system.

Unfortunately, due to limitations in terms of the number of devices that *AmbiMate* is able to run on, it proved infeasible to conduct field studies. Instead, a more traditional evaluation involving user testing was conducted. This evaluation and its results are discussed in Section 5.2.

## 5.2 User Testing

**Methodology.** The *AmbiMate* system was evaluated by eight users. Participants were selected to represent a cross-section of the target user population of the system. As a result, only experienced mobile phone users were selected as participants. Participants included male and female, undergraduate and postgraduate students and staff at NMMU in order to involve a balanced cross-section of the user population.

Each participant in the evaluation was presented with a test plan, consisting of a task list to be performed. The task list comprised the following tasks:

- Activating the system;
- Viewing help;
- Viewing display details;
- Changing the colour associated with a contact;
- Changing the data type being visualised;
- Changing the time period being visualised; and
- Changing the display mode (from Aquarium Design to Flower Design).

A number of questions were included in the test plan after each task, in order to determine the users' understanding of the visualised information, and the changes in the display as a result of the customisations they were asked to perform. On completion of the test plan, participants were also required to complete a questionnaire – a customised version of the widely used Questionnaire for User Interface Satisfaction (QUIS). Users were asked to rate the system according to various criteria using a 5-point Likert scale.

A Nokia 6600 with the *AmbiMate* system installed was used for the user testing. Users were passively observed while carrying out the test plan and their comments were recorded.

**Results.** Table 2 shows the summarised results for each of the main categories of the user satisfaction questionnaire completed by participants in the evaluation.

**Table 2.** Quantitative questionnaire results summary (n=8)

Variable	Mean	Median	Std. Dev.
Overall Reactions	4.04	4.04	0.33
Interface Design	4.23	4.20	0.43
Terminology & System Info	4.56	4.63	0.37
Navigation & Functionality	4.64	4.79	0.31
Information Visualisation	4.13	4.13	0.48
Learning	4.25	4.33	0.64
System Usefulness	4.13	4.00	0.64

General comments received were strongly positively. Users commented that the display was visually appealing and easy to understand.

In the final question of the evaluation questionnaire, users were asked to rate the usefulness of the system on a scale of 1 to 5. The mean rating for this question was 4.13, with a median of 4. All but one user (who gave a rating of 3) gave a rating of 4 or 5. This provides a fairly clear endorsement of the usefulness of the system by the participants in the evaluation. Several users throughout the different evaluations also expressed a keen interest in having the system installed on their own private mobile phones.

Users were also asked to rate the usefulness of the visualised information. This question is slightly more complicated, because ambient displays are typically designed to “*support monitoring of non-critical information*” [5]. As a result, the information sources visualised by ambient displays typically include information that is of passing interest, rather than critical importance. Given this, the mean rating of 3.88 and median of 4.0 given by the participants for the usefulness of the visualised information can be regarded as highly positive. Clearly, while the answer to this question remains largely subjective and dependent on an individual user’s needs, the vast majority of users found the information to be useful.

## 6 Lessons Learned

A number of problems were encountered during the development of the *AmbiMate* system, from which valuable lessons can be learned. Some of these are summarised below:

- A truly device-independent mobile ambient display to visualise personal communication patterns is not possible, mainly because a purely JME implementation is not presently possible (Section 3.2).
- Any application which, like the *AmbiMate* system, relies on data from the contact database needs to take into account the possible lack of information in some fields (Section 4.1).
- While JME proved to be a superior choice for the development of the ambient display, it is by no means an easy task to integrate a Java MIDlet into the operation of a Symbian phone as a screensaver. Special measures had to be taken to achieve screensaver functionality (Section 4.2).
- Special consideration has to be given to battery life when developing mobile ambient displays (Section 4.3).
- Limitations in terms of the processing power of mobile devices also need to be considered when designing a mobile ambient display (Section 4.3).
- It is crucial that Java MIDlets be signed in order to avoid annoying error messages that would detract from the ambient nature of such systems (Section 4.3).

## 7 Conclusions

The development and evaluation of the *AmbiMate* system described in this paper provide valuable insight into the usefulness of mobile ambient displays. In particular,

the results of user testing show that users found the system and the information visualised to be highly useful (Section 5). However, considering that ambient displays are by their very nature designed to fit into the everyday life of the user, future work incorporating field studies is needed to confirm the results of this evaluation.

The development of the *AmbiMate* system also identified a number of problems and limitations in the successful deployment of a mobile ambient display system, from which lessons can be learned (Section 6). These include lessons regarding implementation tool selection, device independence, battery life, processing power and data access. Seamless integration into the functioning of the mobile device is no trivial task, and the problems encountered and lessons learned in the development of the *AmbiMate* system could provide valuable insight for anyone implementing such systems in the future.

## References

1. Smit, T., Hurst, R.: The Power of Mobile IP, ITWeb Market Monitor (2006)
2. Wisneski, C., Ishii, H., Dahley, A., Gorbet, M., Brave, S., Ullmer, B., Yarin, P.: Ambient Displays: Turning Architectural Space into an Interface between People and Digital Information. In: Proceedings of 1st International Workshop on Cooperative Buildings (1998)
3. Weiser, M.: The Computer for the 21st Century. Scientific American (1991)
4. Weiser, M., Brown, J.S.: Designing Calm Technology. Powergrid Journal (1995) [Accessed on 22 March 2006], <http://www.powergrid.com/1.01/calmtech.html>
5. Mankoff, J., Dey, A.K., Hsieh, G., Kientz, J., Lederer, S., Ames, M.: Heuristic Evaluation of Ambient Displays. In: Proceedings of CHI 2003 (2003)
6. Redstrom, J., Skog, T., Hallnas, L.: Informative Art: Using Amplified Artworks as Information Displays. In: Proceedings of Designed Augmented Reality Environments (2000)
7. Schmidt, A., Hakkila, J., Atterer, R., Rukzio, E., Holleis, P.: Using Mobile Phones as Ambient Information Displays. In: Proceedings of CHI 2006 (2006)
8. Weiser, M., Brown, J.S.: The Coming Age of Calm Technology (1996) [Accessed on 22 March 2006], <http://www.ubiq.com/hypertext/weiser/acmfuture2endnote.htm>
9. Ambient Orb. Ambient Devices Inc. (2006) [Accessed on 22 October 2006], <http://www.ambientdevices.com>
10. Skog, T., Ljungblad, S., Holmquist, L.: Between Aesthetics and Utility: Designing Ambient Information Visualizations. In: Proceedings of IEEE Symposium on Information Visualization (2003)
11. Miller, T., Stasko, J.: Personalized Peripheral Information Awareness through Information Art. In: Davies, N., Mynatt, E.D., Siio, I. (eds.) UbiComp 2004. LNCS, vol. 3205, pp. 18–35. Springer, Heidelberg (2004)
12. Fogarty, J., Forlizzi, J., Hudson, S.E.: Aesthetic Information Collages: Generating Decorative Displays that Contain Information. In: Proceedings of UIST 2001, ACM Symposium on User Interface Software and Technology (2001)
13. Streitz, N., Prante, J., Röcker, C., Van Alphen, D., Magerkurth, C., Stenzel, R., Plewe, D.: Ambient Displays and Mobile Devices for the Creation of Social Architectural Spaces. Kluwer Academic Publisher, Dordrecht (2003)

14. Viégas, F., Golder, S., Donath, J.: Visualising Email Content: Portraying Relationships from Conversational Histories. In: Proceedings of CHI 2006 (2006)
15. Nardi, B., Whittaker, S., Isaacs, E., Creech, M., Johnson, J., Hainsworth, J.: ContactMap: Integrating Communication and Information Through Visualizing Personal Social Networks. Communications of the ACM (2002)
16. Xiong, R., Donath, J.: PeopleGarden: Creating Data Portraits for Users. In: Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology (1999)
17. Carpendale, S., Tat, A.: CrystalChat: Visualising Personal Chat History. In: Proceedings of 39th Hawaii International Conference on System Sciences (2006)
18. Canals: Worldwide Smart Mobile Device Research (2006) [Accessed 30 November 2006], <http://www.canalys.com/pr/2006/r2006071.htm>
19. TinyLine: Programmer's Guide to TinyLine 2D API (2006) [Accessed on 11 October 2006], <http://www.tinyline.com/2d/guide/>
20. Gupta, A. and de Jode, M.: Extending the Reach of MIDlets: how MIDlets can access native services (2005) [Accessed on 19 October 2006], [http://developer.symbian.com/main/downloads/papers/MIDlet\\_Native\\_Services\\_Framework/MIDlet\\_Native\\_Services\\_Framework\\_v1.1.zip](http://developer.symbian.com/main/downloads/papers/MIDlet_Native_Services_Framework/MIDlet_Native_Services_Framework_v1.1.zip)
21. de Jode, M.: Programming the MIDlet Lifecycle on Symbian OS (2004) [Accessed on 19 October 2006], <http://www.symbian.com/developer/techlib/papers/midplifecycle/midplifecycle.pdf>
22. Plaue, C., Miller, T., Stasko, J.: Is a Picture Worth a Thousand Words? An Evaluation of Information Awareness Displays. In: Proceedings of 2004 conference on Graphics Interface (2004)
23. Kaikkonen, A., Kallio, T., Kekäläinen, A., Kankainen, A., Cankar, M.: Usability Testing of Mobile Applications: A Comparison between Laboratory and Field Testing. Journal of Usability Studies (2005)
24. Skog, T., Ljungblad, S., Holmquist, L.: Issues with Long-term Evaluation of Ambient Displays. IT University of Göteborg (2006)



# Integration of Distributed User Input to Extend Interaction Possibilities with Local Applications

Kay Kadner and Stephan Mueller

SAP AG, SAP Research CEC Dresden  
Institute of System Architecture, Faculty of Computer Science,  
Dresden University of Technology  
kay.kadner@sap.com, sm853234@inf.tu-dresden.de

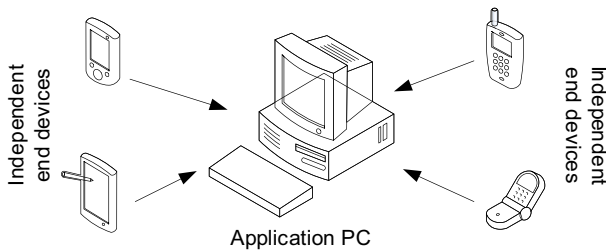
**Abstract.** Computing devices do not offer every modality for interaction that a user might want to choose for interacting with an application. Instead of buying new hardware for extending the interaction capabilities, it should be possible to leverage modalities of independent existing devices that are in the vicinity. Therefore, an architecture has to be developed that gathers events on distributed devices and transfers them to the local device for execution. This allows the user to choose devices even at runtime that are better suited for a particular input task. For a convenient use, the system should support input that can be both independent and dependent from the application. Application-dependent input commands imply that meta-information about the application is provided. Since the system should allow the extension of existing applications, the meta-information has to be provided in a way that is transparent for the application. The following paper describes a system that realises those features.

## 1 Introduction

Electronic devices are ubiquitous in the modern society. According to a market survey [1], the number of mobile phone contracts exceeded the number of residents in Germany at the end of 2006, whereas this is already the case in other countries. PDAs are getting increasingly popular (besides mobile phones), which allow the user to communicate over a variety of technologies like Bluetooth, WiFi, GSM, UMTS and so on. Users always carry PDAs and mobile phones with them, which makes those devices ubiquitous computing units, whose capabilities are barely utilised today.

The different interaction modalities of independent devices like pen input of PDAs can not easily be used to extend the interaction modalities of applications running on a PC, for instance. Many work has been done for extending a PC's desktop, e.g. by remote desktops like VNC [15]. Similar approaches allow implementing distributed services, which can be accessed from various devices resulting in various user interfaces. As a consequence, most related work has explicit impact on applications in order to extend them for remote access or simply replicates the original application's user interface. A system, which allows a lightweight extension of an application's user interface without the need of modifying the implementation or replicating the whole user interface while being open to arbitrary modalities is still missing.

Therefore, we developed a system, which enables the user to leverage the additional interaction modalities of independent devices by capturing distributed input and forwarding it to the application PC, which does not necessarily has to be a desktop PC or laptop. In fact, every device is suitable, as long as it is able to run the remote control application, which receives input and issues that in into the system. For the remainder of the paper we assume that the application PC is the entity, whose interaction capabilities are limited and thus should be enhanced by the use of additional devices. The independent devices are connected to the PC over a network. An overview of the architecture and its participants is shown in Fig. 1. Once the independent devices are connected, the user can use them to issue application-independent input (e.g. simple keyboard input) and, after determining the currently active application on the application PC, the remote devices can also be used to issue application-dependent input commands (e.g. "next slide" in MS PowerPoint).



**Fig. 1.** High-level view of the remote input system

The presented architecture does not provide a separate view on the application through the independent end devices. It is impossible to use the end devices without having access to the application PC, because they do not give feedback originating from the target application to the user. They only provide an extension of the traditional input capabilities according to their local modalities, whereas the extended input capabilities are used in addition to the traditional ones. This results often in additional buttons, but is not limited to a particular representation, which only depends on the devices' modalities.

The paper is structured as follows: an example use case and the requirements to such a system are described in Section 2. Concepts of the architecture are explained in Section 3. Section 4 contains a description of the implemented prototype before the paper closes with a brief overview about related work in Section 5 and a conclusion and outlook in Section 6.

## 2 Use Case and Requirements

This section gives a brief description of an example use case followed by seven requirements that must be fulfilled for developing a convenient and secure system for the described scenario. At the end of the paper, the requirements are used to assess the developed prototype.

## 2.1 Use Case

For graphical designers, it is often more easy to use a stylus for drawing instead of the mouse because of the different way of interacting with the device. Therefore, they usually have a tablet for stylus input, which is attached via cable (e.g. USB) to the PC or laptop. However, such a tablet is not available in every situation, e.g. if the graphical designer is travelling from his company to the customer. His PDA offers the same way of interaction, so he turns it on and configures it for its use as stylus input device on the laptop. An application is started, which captures the cursor movements within a certain area and forwards those to the application PC. Arriving at the customer site, the designer has to give a presentation, which he prepared as a set of slides. As he does not want to be tied to his laptop, he again uses his PDA and configures it for controlling the presentation application. The PDA provides access to commands that are most important for giving a presentation (previous, next, home) and leaves out other commands that are rather used for editing (copy, paste). In both cases, the PDA has no physical connection to the laptop but communicates over wireless networks.

Besides replicating already available interaction means like buttons or menus on the client device, it is also possible to create shortcuts to functionality that might be hidden or hard to access as well as a combination of multiple functionalities. In MS Word for instance, if you want to turn on the thumbnails view, you have to click the respective button from the View menu. By creating a shortcut button to that functionality on the PDA, you only need one click instead of two. More complex commands like selecting and formatting a paragraph to a certain font type, font size, and line spacing can also be defined.

## 2.2 Requirements

The requirements for the scenario are described in the following:

1. For ensuring a high flexibility with regard to extensibility of the application PC, the system must allow the use of independent and distributed devices for remotely issuing input. Due to their distributed nature, those devices must be connected to the PC over a network (R1), which is still a loose coupling.
2. The independent devices must allow for both application-dependent (R3) and application-independent input (R2). Since the devices are heterogeneous, they present their local user interface according to their user interface capabilities, which results in multimodal interfaces and therefore multimodal control of the PC's applications.
3. Furthermore, the system must provide means for securing the communication channels between the application PC and the independent devices (R4), because modifying or recording the communication channels by unauthorised parties must be prevented. This is especially critical for such a system as it is described here, because it realises the interaction between user and applications, which is regarded to be safe in traditional scenarios (PC and attached keyboard and mouse).
4. Since the system must allow the extension of existing applications (R7), it needs information about available application commands that the user wants to use.
5. Because it is not feasible that existing applications are modified and not all applications provide interfaces for accessing this information, the system must implement a means to provide the command description for application-dependent input

commands in another way (R5), which allows the independent devices to create output in their available modalities.

6. The independent devices must always be informed about the currently active application (the target application) on the PC, because this affects their locally offered application-dependent input commands (R6).

### 3 Architecture of the System

In the first part of this section, the architecture of the system will be presented. The architecture supports both application-independent and application-dependent input. The latter type is called *input commands*, since this input is usually at a semantically higher level than application-independent input. Application-independent input is realised by simply forwarding the keyboard and mouse events, which can be regarded as simple input commands and are therefore not explained further in this paper. The realisation of application-dependent input commands will be explained in the last part of this section after a detailed view on the server-side input receiver.

#### 3.1 Architecture

The system architecture is shown in Figure 2. The application PC can be controlled by traditional local input via keyboard and mouse. The independent end devices make also use of their local input methods, whereas the remote control application can distinguish between application-independent or application-dependent input. Application-independent input can always be used, whereas application-dependent input is especially tailored for the application that should be controlled. Depending on its concrete implementation, the remote control application can offer separate subsets of the overall interaction possibilities on the end device for supporting different kinds of tasks, for instance navigating requires cursor keys, page up/down, home/end, whereas for text input, a soft keyboard is be required.

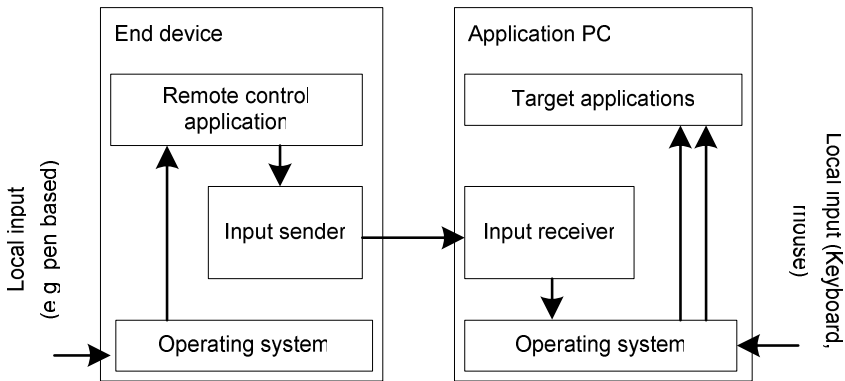


Fig. 2. Architecture of the system

Application-independent input is mapped on device-independent keyboard or mouse events before it is forwarded to the application PC. If application-dependent input commands are entered, their ID, as specified in the application description document (see 3.3), is forwarded to the application PC where the ID is mapped to concrete keyboard and mouse events. Mapping input commands to concrete events on the application PC is advantageous because a different implementation of the input receiver may be able to realise the command's function in a different way, e.g. by directly accessing some interface of the target application. In this case, the input sender implementation does not need to be changed. However, the application description also supports the other way, i.e. executing the events that belong to a command on the end device. This solely depends on the concrete implementation of the remote control application.

### 3.2 Details of the Input Receiver

The input receiver is responsible for receiving and processing the events according to application-dependent and application-independent input on the client devices. The actual reception is implemented in the Receiver component (see Fig. 3). It listens on a specific port for incoming connections and checks with the Security component, if the requesting client is allowed to connect. The security component can for instance simply display a warning to the user and asks for permission. The channel component encapsulates the communication details of sending to and receiving from the client device. Sending messages is omitted in Fig. 3 since it is only used for notifying the client about updates of the currently active application. If the channel component receives a message with input events, it dispatches them to the appropriate subsequent component. Application-independent events are forwarded to the Event service, which generates the according events in the operating system. Application-dependent input commands are first processed by the Command service, which maps the input commands to concrete key events and forwards them to the Event service for generation. As already mentioned, another implementation of the Command service may directly access the target application via an interface and execute the input command by calling an appropriate method. The context component provides information about the application PC, which is relevant for generating application-independent events on the client, like the display resolution or the used keyboard layout.

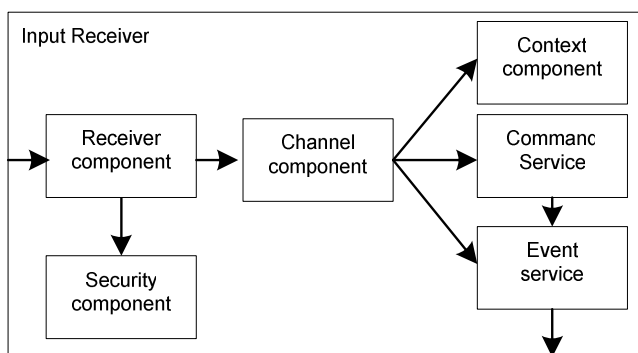


Fig. 3. Detailed architecture of the Input Receiver

Both the channel component and the event service are responsible for handling sudden disconnections of client devices. This needs special attention because otherwise it might happen that a "key press" event is properly executed but the according "key released" event does never occur. The Event service is notified by the channel component, which device has been disconnected. Now, the event service is able to perform compensation actions. The compensation is realised by artificially creating the proper compensation event (the "key released" in case of keyboard input).

### 3.3 Creation of Application-Dependent Input Commands

For creating application-dependent input commands, the remote control application needs information about the target application on the application PC. The remote control application then downloads a description file of this application from the application PC, which contains a list of application-dependent input commands. For ensuring a convenient user interface on the independent devices, the file consists of information how each command can be realised in different output modalities because an extensive textual explanation can be useful for a monitor but not for voice output.

Figure 4 shows an example of an application description file, which defines the command "next" that is used for forward navigation in a presentation. The remote control application chooses between a long or short name for rendering, depending on the current situation for rendering output (e.g. number of commands, screen size). If speech recognition is used, a voice recognition grammar can be created based on the provided voice commands. In the <events> section, the concrete key event types and codes are specified, which shall be created on the PC. The two key events shown in the example are the Eclipse SWT [3] definition of the right cursor key.

```
<application name="MS Powerpoint" version="2003">
  <command id="next">
    <shortname>Next</shortname>
    <longname>Next Slide</longname>
    <speechcommand>next</speechcommand>
    <events>
      <keyboard type="keypress" keycode="16777220"/>
      <keyboard type="keyrelease" keycode="16777220"/>
    </events>
  </command>
</application>
```

Fig. 4. Example of a configuration file for application dependent commands

## 4 Prototype

The prototype was implemented in Java for ensuring platform independence. The determination of the currently active application on the application PC is not possible in Java and would imply implementations on operating system level. Since this contradicts with the platform independence, we decided to leave the task of application determination up to the user. Therefore, the user has to define manually the currently

active application on the application PC by selecting it in the administration tool (R6, Figure 5). A combination of the Eclipse Standard Widget Toolkit [3] and the `Robot`-class of the Java Abstract Window Toolkit [9] was necessary to create events on the application PC because neither of them is capable of creating all possible events.

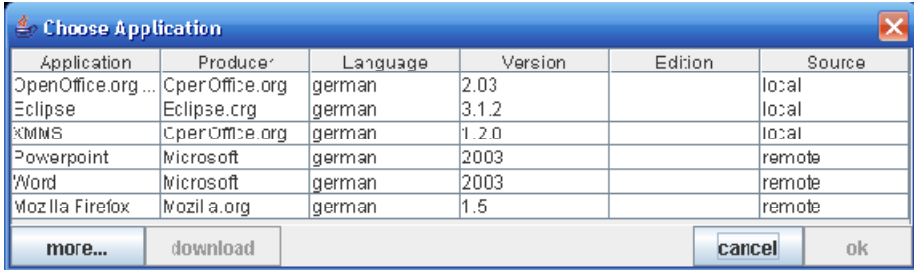


Fig. 5. Screenshot of the administration tool

We decided to implement the application on the end device with Java Micro Edition (JME, [8]) for supporting a wide range of client devices like mobile phones and PDAs. This application allows establishing a connection over plain sockets to the application PC (R1) and issuing of application-dependent and application-independent events (R2, R3, Figure 6). Because of both, the indirect injection of the input events from the PC into the target application via the operating system and the external description of applications (R7), a transparent extension of existing applications is possible (R5).

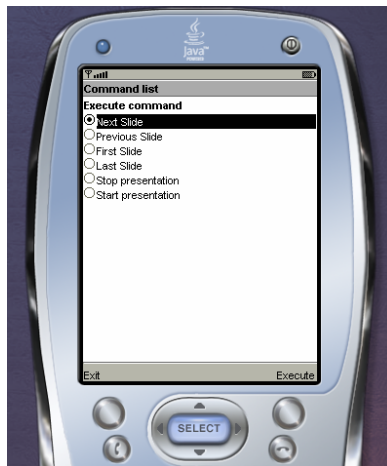


Fig. 6. Screenshot of the JME client

If the end device connects to the PC, the user, which is currently logged onto the application PC, will be asked whether to allow or deny the access by the end device. The connection would have to be encrypted for completely fulfilling the security requirement (R4). It was first intended to use an HTTPS connection, but since the SSL implementation Java ME caused several problems, which were not solvable within the project, we decided to leave secure communication out. Figure 5 shows the configuration application of the PC and Figure 6 contains a screenshot of the JME application on the right.

## 5 Related Work

The Input Adaptation Project (IAT) has the goal to replace standard input devices like mouse or keyboard with other input devices [10]. A complex theoretical model is the basis for mapping input events of other devices to concrete keyboard events like pressing a key. The process of adapting the input events is done in three subsequent phases. First, the input events are transformed by a device-specific InputAdapter to a uniform state space. In the second phase, the uniform state space is mapped to the states of the target device. Finally, in the third phase, the states of the target devices are mapped to one or multiple concrete events, which are then created by the target devices. The last mapping is realised by a device-specific OutputExplorer. Since the mappings are defined in a special description language and a change of that mapping description is not possible at runtime, the mapping cannot be changed according to the currently active application, which is a major goal of the presented system.

The project Pebbles [6] investigates scenarios in which PCs and PDAs can collaborate. Pebbles' architecture is divided in three components: the *service user* runs on the PDA and offers the user interface. The *service* runs on the PC and controls the application and the system, and the *PebblesPC* runs on the PC and acts as name service as well as message router between the service user and the service. This architecture offers the possibility to create adaptations of special applications with application-dependent services. Nevertheless, each application that should make use of the Pebbles architecture has to be implemented separately. Therefore, the flexibility of enhancing an arbitrary application is limited. Furthermore, PebblesPC only supports the collaboration of PDA and PC for including pen input. Other modalities were not considered.

With the help of the Personal Universal Controller (PUC, [13]), users are enabled to interact with arbitrary appliances in their environment. The PUC architecture consists of four elements: the appliance adapter, the communication protocol, the specification language and the user interface generator. The communication protocol supports peer-to-peer communication, thus allowing multiple user interface generators to be connected to multiple appliance adapters. The specification language describes the user interface in an abstract way and is used to transport state information of the appliance. The PUC system focussed on automatic generation of high-quality user interfaces for appliance control, which is different from providing additional input means to applications, because each appliance needs a special adapter with a special user interface specification. In contrast, controlling a previously unknown application is always possible, because application-independent input is always supported.



The ICrafter service framework allows the user to interact with services in the user's interactive workspace [14]. Services can be devices or applications that provide useful functions. The service framework is used by developers to deploy services and to create the services' user interfaces for various appliances. Appliances request user interfaces from the interface manager for a particular application, specified by an application description. The interface manager retrieves all necessary information (service and appliance description, context information) and generates a user interface, which is returned to the requesting appliance. As the PUC before, services for the ICrafter system must be explicitly implemented. Our solution focusses on the extension of existing applications without the need to reimplement them.

The aim of the OSI Virtual Terminal Service (VTS) is to provide access to virtual terminals within a distributed network [11]. The VTS employs a mapping of events to a uniform state space on the client device and vice versa on the terminal device, which is quite similar to the mapping function of the IAT project. Although the system distinguishes objects that are used for either output or input only, it is used for remote terminal access, which involves input and output of information through the same device. In contrast to the presented approach, the VTS aims at replicating the target applications at the application level. This implies an enormous implementation effort on the application PC, because not all applications are suitable for that. This prevents a seamless and easy extension of existing applications.

Microsoft developed the Remote Desktop Protocol [12] for accessing Windows machines (the server) from different hosts (clients). This is realised by forwarding the user interface (i.e. the desktop) to the client where the local input is gathered and forwarded to the server. By this, the user can access the server's applications through a different device. However, the user can still use mouse and keyboard only. The client does not act as an extension of the already existing interaction capabilities but instead replaces the currently existing device.

Furthermore, there are several voice navigation systems (e.g. Realize Voice [7], VRCommander [4], and e-speaking [2]) that enable the user to extend existing applications by voice commands. This includes the use of application-dependent input commands in order to control the applications in a more goal-oriented way. Using these systems limits the extension to voice interaction and the static use of command mappings since they are unable to dynamically adjust the mappings at runtime.

The goal of Salling Clicker (<http://www.salling.com>) is similar to that of our approach. However, the concrete architecture is not described on their website. The scripts for application extension are based on Javascript and do not support the employment of additional modalities like voice. Additionally, application-independent input is not possible.

## 6 Conclusion and Outlook

The presented system offers a transparent extension of existing applications with interaction capabilities of various end devices. This is achieved by gathering distributed input events and forwarding them to the applications on the PC. Due to the explicit support of heterogeneous end devices, multimodal application control based on a federation of end devices is possible. This covers the input side of the architecture

for federated devices, which is sketched in [5]. The transparency is achieved by using external application description documents for defining application-dependent input commands, which define the mapping of higher level input commands to concrete mouse or key events, which are executed on the application PC. The application description supports rendering in multiple modalities by providing several representations of a certain element, which can be used by the remote control application according to the current rendering situation.

The system can be further developed for supporting program APIs for command execution instead of mappings to keyboard events and indirect injection through the operating system. However, this requires a more comprehensive application description than the current one. It should also be evaluated, if and how this approach can be used to increase the robustness against erroneous user input especially if multiple end devices are used. Furthermore, the possibilities to ensure trust and integrity have not been fully employed in the current prototype and might be realised by the use of SSL communication or certificate authentication. The application-dependent input may be extended by supporting parameters that enhance the actual command.

## References

1. Axel Springer, A.G.: Telekommunikation 2006. Market report (2006)
2. e Speaking.com: Voice and Speech Recognition, <http://www.espeaking.com>
3. The Eclipse Foundation: The Standard Widget Toolkit, <http://www.eclipse.org>
4. Interactive Voice Technologies: VRCommander, <http://www.vrcommander.com>
5. Kadner, K.: A flexible architecture for multimodal applications using federated devices. In: Proceedings of Visual Languages and Human-Centric Computing, Brighton, UK, September 2006, pp. 236–237. IEEE Computer Society, Los Alamitos (2006)
6. Myers, B.A.: Using handhelds and PCs together. *Communications of the ACM* 44(11), 34–41 (2001)
7. Realize Software Corporation: Realize Voice, <http://www.realizesoftware.com/>
8. Sun Microsystems: Java Micro Edition, <http://java.sun.com/j2me>
9. Sun Microsystems: The Abstract Window Toolkit, <http://java.sun.com/j2se>
10. Wang, J., Mankoff, J.: Theoretical and architectural support for input device adaptation. In: CUU 2003: Proceedings of the 2003 conference on Universal usability, pp. 85–92. ACM Press, New York (2003)
11. Lowe, H.: OSI virtual terminal service. *Proceedings of the IEEE* 71(12), 1408–1413 (1983)
12. Microsoft Corp.: Understanding the Remote Desktop Protocol (RDP), <http://support.microsoft.com/kb/186607>
13. Nichols, J., Myers, B.A.: Controlling Home and Office Appliances with Smart Phones. *IEEE Pervasive* 5(3) (July–September 2006)
14. Ponnekanti, S.R., Lee, B., Fox, A., Hanrahan, P., Winograd, T.: ICrafter: A service framework for ubiquitous computing environments. In: Abowd, G.D., Brumitt, B., Shafer, S. (eds.) *UbiComp 2001*. LNCS, vol. 2201, p. 56. Springer, Heidelberg (2001)
15. Richardson, T., Stafford-Fraser, Q., Wood, K.R., Hopper, A.: Virtual network computing. *IEEE Internet Computing* 2(1), 33–38 (1998)

# Reverse Engineering Cross-Modal User Interfaces for Ubiquitous Environments

Renata Bandelloni, Fabio Paternò, and Carmen Santoro

ISTI-CNR, Via G.Moruzzi, 1  
56124, Pisa, Italy

{Renata.Bandelloni, Fabio.Paterno, Carmen.Santoro}@isti.cnr.it

**Abstract.** Ubiquitous environments make various types of interaction platforms available to users. There is an increasing need for automatic tools able to transform user interfaces for one platform into versions suitable for a different one. To this end, it is important to have solutions able to take user interfaces for a given platform and build the corresponding logical descriptions, which can then be manipulated to obtain versions adapted to different platforms. In this paper we present a solution to this issue that is able to reverse engineer even interfaces supporting different modalities (graphical and voice).

**Keywords:** Reverse Engineering, Cross-Modal User Interfaces, Model-based Approaches.

## 1 Introduction

In recent years, one of the main characteristics of Information and Communication Technology is the continuous proliferation of new interactive platforms available for the mass market. They vary not only in terms of screen size, but also in terms of the interaction modalities supported. Indeed, if we consider the Web, which is the most common interaction environment, we can notice that recently a number of W3C standards have been under development in order to also consider interaction modalities other than the simple graphical one.

One important consequent problem is how to obtain applications that can be accessed through such a variety of devices. It can be difficult and time-consuming to develop user interfaces for each potential platform from scratch. In order to address such issues in recent years there has been an increasing interest in model-based approaches able to allow designers to focus on the main logical aspects without having to deal with a plethora of low-level details. To this end, a number of device-independent markup languages have been proposed to represent the relevant models in device-independent languages (see for example XIML, UIML, UsiXML, TERESA XML). However, developing such model-based specifications still takes considerable effort. In order to reduce such effort there are two possible general approaches: informal-to-formal transformations or reverse engineering. In informal-to-formal approaches the basic idea is to take informal descriptions, such as graphical sketches or natural language descriptions of scenarios, and try to infer, at least partly, the corresponding logical abstractions. Reverse engineering techniques aim to obtain transformations able to

analyse implementations and derive the corresponding logical descriptions. Thus, they can be a useful step towards obtaining new versions of an implementation more suitable for different platforms.

Solutions based on syntactical transcoders (for example from HTML to WML) usually provide results with poor usability because they tend to fit the same design to platforms with substantial differences in terms of interaction resources. One possible solution to this problem is to develop reverse engineering techniques able to take the user interface of existing applications for any platform and then build the corresponding logical descriptions that can be manipulated in order to obtain user interfaces for different platforms that share the original communication goal, but are implemented taking into account the interaction resources available in the target platforms. This requires novel solutions for reverse engineering of user interfaces, given that previous work has focused only on reverse engineering of graphical desktop user interfaces.

In this paper we present ReverseAllUIs, a new method and the associated tool able to address such issues. We first provide some background information regarding the logical framework underlying this work and the various logical descriptions that are considered. We introduce the architecture of our tool, indicating its main components, their relations and describing its user interface. Then, we discuss how in our environment both vocal and graphical interfaces can be reverse engineered through a number of transformations by describing each transformation involved when considering cross-modal interfaces (interfaces of applications that can be accessed through either one modality or another one). Lastly, some conclusions along with indications for future work are provided.

## 2 Related Work

Early work in reverse engineering for user interfaces was motivated by the need to support maintenance activities aiming to re-engineer legacy systems for new versions using different user interface toolkits [9, 13], in some cases even supporting migration from character-oriented user interfaces to graphical user interfaces.

More recently, interest in user interface reverse engineering has received strong impetus from the advent of mobile technologies and the need to support multi-device applications. To this end, a good deal of work has been dedicated to user interfaces reverse engineering in order to identify corresponding meaningful abstractions [see for example 2, 3, 6, 7, 11]. Other studies have investigated how to derive the task model of an interactive application starting with the logs generated during user sessions [8]. However, this approach is limited to building descriptions of the actual past use of the interface, which is described by the logs, but it is not able to provide a general description of the tasks supported, which includes even those not considered in the logs. A different approach [5] proposes re-engineering Java graphical desktop applications to mobile devices with limited resources, without considering logical descriptions of the user interface. One of the main areas of interest has been how to recover semantic relations from Web pages. An approach based on visual cues is

presented in [15], in which semantic relations usually apply to neighbouring rectangle blocks and define larger logical rectangle blocks.

The next section discusses the various possible logical levels that can be considered for user interfaces in ubiquitous environments. Previous work in reverse engineering has addressed only one level at a time. For example, Vaquita and its successors [2, 3] have focused on creating a concrete user interface from Web pages for desktop systems. WebRevenge [11] has addressed the same types of applications in order to build only the corresponding task models.

In general, there is a lack of approaches able to address different platforms, especially involving different interaction modalities, and to build the corresponding logical descriptions at different abstraction levels: our work aims to overcome this limitation.

### 3 Background

In the research community in model-based design of user interfaces there is a consensus on what constitutes useful logical descriptions [4, 12, 14].

We provide a short summary for readers unfamiliar with them:

- The task and object level, which reflects the user view of the interactive system in terms of logical activities and objects that should be manipulated in order to accomplish them;
- The abstract user interface, which provides a modality independent description of the user interface;
- The concrete user interface, which provides a modality dependent, but implementation language independent, description of the user interface;
- The final implementation, in an implementation language for user interfaces.

Thus, for example we can consider the task “select an artwork”: this implies the need for a selection object at the abstract level, which indicates nothing regarding the modality in which the selection will be performed (it could be through a gesture or a vocal command or a graphical interaction). When we move to the concrete description then we have to assume a specific modality, for example the graphical modality, and indicate a specific modality-dependent interaction technique to support the interaction in question (for example, selection could be through a radio-button or a list or a drop-down menu), but nothing is indicated in terms of a specific implementation language. When we choose an implementation language we are ready to make the last transformation from the concrete description into the syntax of a specific user interface implementation language. The advantage of this type of approach is that it allows designers to focus on logical aspects and take into account the user view right from the earliest stages of the design process.

In the case of interfaces that can be accessed through different types of devices the approach has additional advantages. First of all, the task and the abstract level can be described through the same language for whatever platform we aim to address, which means through device-independent languages. Then, in our approach, TERESA XML

[1], we have a concrete interface language for each target platform. By platform we mean a set of interaction resources that share similar capabilities (for example the graphical desktop, the vocal one, the cellphone, the graphical and vocal desktop). Thus, a given platform identifies the type of interaction environment available for the user, and this clearly depends on the modalities supported by the platform itself. Actually, in our approach the concrete level is a refinement of the abstract interface depending on the associated platform. This means that all the concrete interface languages share the same structure and add concrete platform-dependent details on the possible attributes for implementing the logical interaction objects and the ways to compose them indicated in the abstract level. All languages in our approach, for any abstraction level, are defined in terms of XML in order to make them more easily manageable and allow their export/import in different tools.

Another advantage of this approach is that maintaining links among the elements in the various abstraction levels provides the possibility of linking semantic information (such as the activity that users intend to do) and implementation levels, which can be exploited in many ways. A further advantage is that designers of multi-device interfaces do not have to learn the many details of the many possible implementation languages because the environment allows them to have full control over the design through the logical descriptions and leave the implementation to an automatic transformation from the concrete level to the target implementation language. In addition, if a new implementation language needs to be addressed, the entire structure of the environment does not change, but only the transformation from the associated concrete level to the new language has to be added. This is not difficult because the concrete level is already a detailed description of how the interface should be structured.

The purpose of the logical user interface XML-based languages is to represent the semantics of the user interface elements, which is the type of desired effect they should achieve: they should be able to allow the user to accomplish a specific basic task or to communicate some information to the user. In particular, in TERESA XML there is a classification of the possible interactors (interface elements) depending on the type of basic task supported (for example single selection, navigator, activator, ...) and the ways to compose them. Indeed, the composition operators in TERESA XML are associated with the typical communication goals that designers want to achieve when they structure the interface by deciding how to put together the various elements: highlighting grouping of interface elements (*grouping*), one-to-many relations among such elements (*relation*), hierarchies in terms of importance (*hierarchy*), or specific ordering (*ordering*).

## 4 Architecture

The architecture of our tool is represented in Figure 1. It can handle multiple types of input and generate multiple types of output, which are represented by the arrows on the border of the rectangle associated with the ReversAllUIs tool.

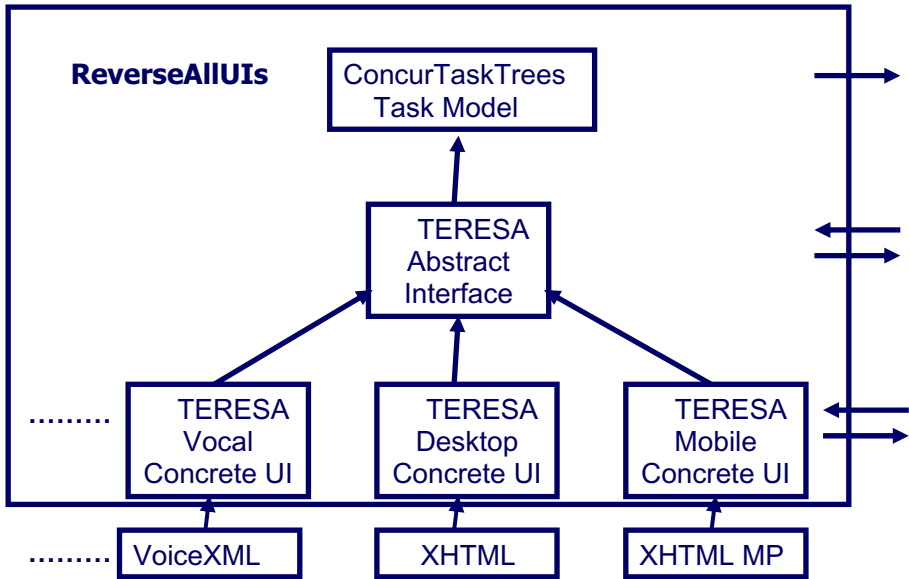


Fig. 1. The architecture of the tool

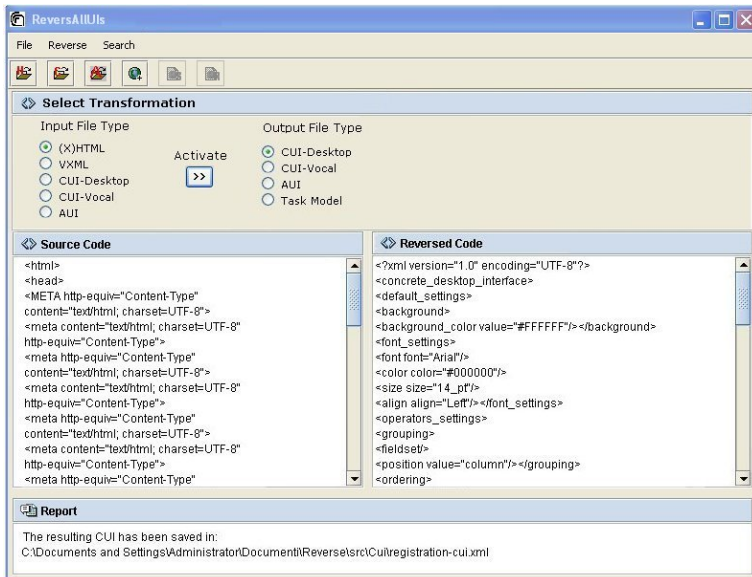


Fig. 2. The user interface of the tool

Our current tool addresses VoiceXML, XHTML and XHTML Mobile Profile (MP) as implementation languages, but we are planning to support additional languages (such as X+V and Java, including the version for digital TV). One main characteristic

is that the tool can receive as input not only user interface implementations, but also descriptions at intermediate abstraction levels, which can be reversed in order to obtain higher level descriptions. The highest level description is the task model, which, consequently, can only be an output for our tool.

Figure 2 shows the user interface of the tool. It allows the designer to select the type of input and output file. In the list of available input files there are implementation languages (such as XHTML and VoiceXML), concrete user interfaces (CUI) that depend on the platform (such as desktop and vocal) and the abstract specification (AUD), which is both implementation language and platform-independent.

Both the source file and the resulting reversed file are displayed. In the bottom some report messages are presented.

## 5 XHTML/CSS-to-Desktop or Mobile Concrete Descriptions Transformation

The reverse tool can reverse both single XHTML pages and whole Web sites. A Web site is reversed considering one page at a time and reversing it into a concrete presentation. Thus, the tool builds connections among the different presentations depending on the navigation structure of the Web site, and the presentations are arranged into a single concrete description representing the whole Web site.

When a single page is reversed into a presentation, its elements are reversed into different types of concrete interactors and combination of them. The reversing algorithm recursively analyses the DOM tree of the X/HTML page starting with the *body* element and going in depth. For each tag that can be directly mapped onto a concrete element, a specific function analyses the corresponding node and extracts information to generate the proper interactor or composition operator. In the event that a CSS file is associated to the analysed page, for each tag that could be affected by a style definition (such as background colour, text style, text font) the tool checks possible property definitions in the CSS file and retrieves such information to make a complete description of the corresponding concrete interactor.

Then, depending on the XHTML DOM node analysed by the recursive function, we have three basic cases:

- The XHTML element is mapped into a concrete interactor. A tag can correspond to multiple interactors (e.g. input or select tag): in this case the choice of the corresponding interactor depends on the associated type or attributes. This is a recursion endpoint. The appropriate interactor element is built and inserted into the XML-based logical description.
- The XHTML node corresponds to a composition operator, for example in the case of a div or a fieldset. The proper composition element is built and the function is called recursively on the XHTML node subtrees. The subtree analysis can return both interactor and interactor composition elements. Whichever they are, the resulting concrete nodes are appended to the composition element from which the recursive analysis started.
- The XHTML node has no direct mapping to any concrete element. If the XHTML node has no child, no action is taken and we have a recursion endpoint, otherwise recursion is applied to the element subtrees and each



child subtree is reversed and the resulting nodes are collected into a grouping composition.

Table 1 shows the main XHTML tags and the corresponding interactors/operators for the desktop concrete description.

**Table 1.** Reversing XHTML to CUI-Desktop

<b>X/HTML Element</b>	<b>CUI-Desktop Element</b>
<ol>	OrderedList(Ordering)
<ul>	Bullet(Grouping)
<table>	Fieldset, BgColor(Grouping) DescriptionTable
<tr>	Fieldset, BgColor(Grouping) TableRow
<td>	Fieldset, BgColor(Grouping) TableData
<select>	List_box Drop_down_list
<select multiple>	ListBox
<textarea>	Textfield
<form>	Form(Relation)
<input type=text>	Textfield
<input type=checkbox>	Checkbox
<input type=radio>	Radiobutton
<input type=reset>	ResetButton
<input type=submit>	SubmitButton
<input type=button >	Button ButtonAndScript
<div>	Fieldset, Bullet, BgColor(Grouping)
<fieldset>	Fieldset(Grouping)
<a>	TextLink ImageLink mailto
<h1>.. <h6&gt; &lt;b&gt;<br=""></h6&gt;> <strong> <em> <i> <tt> <code> <cite> <def> <kbd> <big> <small> <sub> <sup> <var>	Textual
<img>	Image

As we can see from Table 1, some of the XHTML tags can be reversed into more than one concrete element. The choice of the proper elements depends on the attributes of the XHTML tags.

In the case of the `<table>` tag, we considered that often it is used in order to define the layout of the page, even if it is generally considered not a good design choice. When reversing a XHTML table it is necessary to recognise the purpose for which it has been used. When it is a proper table showing data, it is reversed into the corresponding *table* concrete element, otherwise it is considered as a technique for grouping the contained elements. Some rules used to distinguish layout tables from data tables are:

- tables with attribute “border = 0” are probably layout tables,
- tables with attribute border set to a value greater than 0 are probably data tables,
- tables having tag `<body>` as a parent and no other sibling tags are layout tables,
- tables having the summary attribute are data tables,
- tables that define a caption element are data tables.

After the first generation step, the logical description is optimised by eliminating some unnecessary grouping operators (mainly groupings composed of one single element) that may result from the first phase. This can happen for example with tags such as `<div>` and `<fieldset>` that are automatically reversed into groupings but whose content includes only a single interactor, such as piece of text and images that can be joined into a single description interactor.

XHTML MP is a subset of XHTML more suitable for mobile devices. The concrete description for the mobile device platform is also a subset of that for the desktop system, it provides a smaller set of elements for implementing the higher level interactors and composition operators. Thus, when a XHTML MP implementation is found, then it is required to apply a transformation that works on a subset of input and output of the transformation previously described.

## 6 VoiceXML to Vocal Concrete Description Transformation

The basic elements of a voice application written in VoiceXML are *form(s)* and *menu(s)*. The *form* element has the same purpose as the XHTML form, that is, to collect information and pass them to a server for further processing. Thus, the VoiceXML form is reversed into a *Relation* operator like the XHTML form. Inside the form, we can find *Grouping* of interactors obtained from reversing the VoiceXML form elements for entering input. Mainly they are specified through *subdialog*, *record* and *field*.

*Subdialog* is a kind of smaller voice dialog contained in the main voice dialog, thus it is reversed into a grouping of the contained elements.

*Record* performs the registration of a vocal input from the user in an audio file format, which is reversed into a concrete *vocal\_input\_file* element.

*Field* is used to recognize user vocal input, not in an audio file format, but as text that can be eventually matched against a grammar specified in the VoiceXML file. The field can be reversed into a *vocal\_input\_text* element, in case it allows free or grammar-driven vocal input, or into a *vocal\_selection* in case it contains *<option>* children nodes specifying the only possible answers among which the user can choose. The field tag can specify a grammar to restrict the range of possible vocal input from the user. Such a grammar is also retrieved and specified in the corresponding concrete element.

In addition, the Relation composition obtained as output of reversing a form can also contain a Grouping of control elements derived from reversing the VoiceXML *<clear>* and *<submit>* tags.

The second basic element of VoiceXML presentations is the *menu*. The menu is used to allow the user to navigate through the same dialogue or into a new one. Thus the menu is reverse engineered into a concrete *Ordering* of navigator elements. More specifically, this navigator can be: *enumerate\_menu*, *dtmf\_menu* or *message\_menu*, depending on the type of VoiceXML menu.

**Table 2.** Mappings from VXML to the Vocal Concrete User Interfaces

VXML Tag	CUI-Vocal Element
<i>&lt;form&gt;</i>	ChangeContext(Relation)
<i>&lt;block&gt;</i>	Insert_sound, Insert_pause, Change_volume, Keywords(Grouping)
<i>&lt;subdialog&gt;</i>	Insert_sound , Insert_pause, Change_volume, Keywords(Grouping)
<i>&lt;record&gt;</i>	VocalInputFile
<i>&lt;field&gt;</i>	VocalInputText VocalSelection
<i>&lt;clear&gt;</i>	ResetCmd
<i>&lt;reset&gt;</i>	SubmitCmd
<i>&lt;menu&gt;</i>	EnumerateMenu DtmfMenu MessageMenu
<i>&lt;prompt&gt;</i>	FeedbackMessage SimpleText
<i>&lt;paragraph&gt;</i>	SimpleText
<i>#text</i>	SimpleText
<i>&lt;link&gt;</i>	VocalCommand
<i>&lt;prosody volume = "+X"&gt;</i>	IncreaseVolume(Hierarchy)
<i>&lt;prosody volume = "-X"&gt;</i>	DecreaseVolume(Hierarchy)
<i>&lt;audio&gt;</i>	Sound

A properly designed voice user interface includes feedback messages summarising the user activity. Each concrete interactor can define a feedback message associated to the interaction object. In order to identify the feedback messages and associate them to the proper interactor, we analyse all the messages contained in the vocal presentation. Thus, all those messages that contain a field value reading as vocal output are considered to be feedback messages of the field.

The elements described can be further composed by the Hierarchy operator in the event that an increase or decrease of the vocal volume is detected. Another composition of elements is identified when different VoiceXML interface elements are enclosed between a starting and ending sound, in this case a Grouping structure is associated with the interactors corresponding to the enclosed VoiceXML elements.

**Table 3.** Mappings of CUI-Desktop and CUI-Vocal elements to Abstract elements

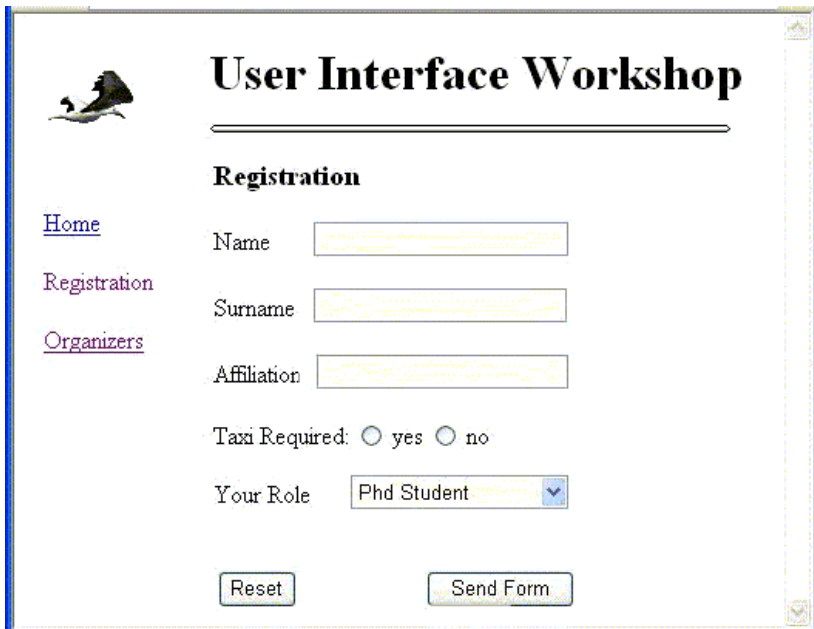
<b>CUI-Desktop</b>	<b>CUI-Vocal</b>	<b>Abstract Interface</b>
OrderedList	alphabeticalOrder Keywords	Ordering
BiggerFont	IncreaseVolume DecreaseVolume	Hierarchy
Form	ChangeContext	Relation
Fieldset Bullet BgColor Bullet	InsertSound InsertPause ChangeVolume Keywords	Grouping
RadioButton ListBox DropDownList	VocalSelection	SelectionSingle
CheckBox ListBox	VocalSelection	SelectionMultiple
Textfield	VocalInputText	TextEdit
Textfield	VocalInputText	NumericalEdit
NOT SUPPORTED	VocalInputFile	ObjectEdit
ImageMap	NOT SUPPORTED	PositionEdit
TextLink ImageLink Button	VocalCommand EnumerateMenu DtmfMenu MessageMenu	Navigator
ResetButton ButtonScript MailTo	ResetCmd SubmitCmd CmdAndScript	Activator
SimpleText TextFile	SimpleText TextFile AudioFile	Text
Image	Sound	Object
TextImage Table	VocalDescription	Description
NOT SUPPORTED	FeedbackMessage	Feedback

## 7 Concrete Descriptions to Abstract Description Transformation

The Abstract User Interface is a platform- and implementation language-independent description of the user interface, conversely to the Concrete User Interface, which is a language specific for each platform for which the user interface is designed. This means that reversing any platform-specific concrete description yields an abstract description always in the same language. Since in TERESA XML the concrete descriptions are a refinement of the abstract one, they add implementation details to the higher level interactors defined in the abstract descriptions. The process for reversing a concrete description into the corresponding abstract one is quite simple, since it simply consists in removing the lower level details from the interactor and composition operators specification, while the structure of the presentations and the connections among presentations remain unchanged.

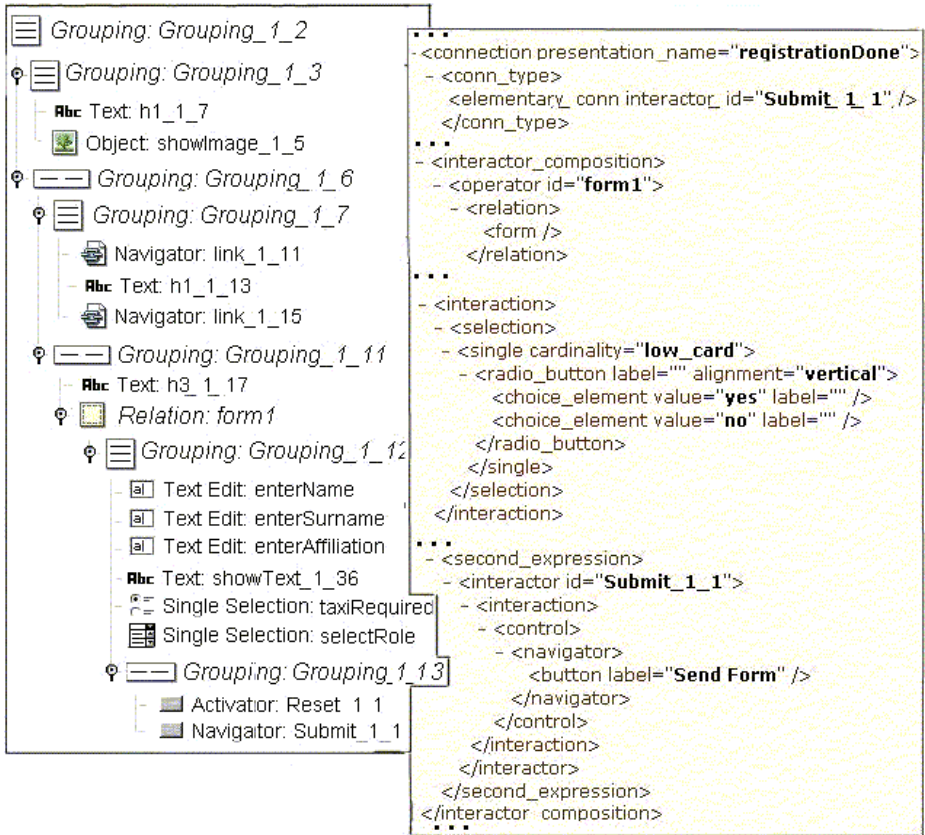
## 8 Example Applications

Figure 3 shows an example of a XHTML page for the desktop platform. It allows the user to navigate among different pages through a navigation menu on the left and shows a form that can be filled in and submitted for registering to a User Interface Workshop event. As you can note, when the registration page is visualised, the related link in the navigation menu on the left does not visualise “Registration” as a link.



The screenshot shows a web browser window displaying a registration page for a "User Interface Workshop". The page has a blue border. On the left side, there is a navigation menu with three links: "Home", "Registration", and "Organizers". The "Registration" link is highlighted in purple. The main content area features a logo of a person at a computer, the title "User Interface Workshop" in a large, bold, black font, and a horizontal line. Below the title, the word "Registration" is written in a bold, black font. The form consists of several fields: "Name" (text input), "Surname" (text input), "Affiliation" (text input), "Taxi Required:" with radio buttons for "yes" and "no", and "Your Role" (a dropdown menu currently showing "Phd Student"). At the bottom of the form, there are two buttons: "Reset" and "Send Form".

Fig. 3. Desktop XHTML example page



**Fig. 4.** An abstract description of the graphical example (left) and an excerpt of the corresponding XML concrete description (right)

Figure 4 shows the result of reversing the XHTML page into a Concrete User Interface. The representation of the Concrete User Interface has been obtained by loading the resulting CUI-Desktop specification in the TERESA tool, which shows in the tree-like format the higher level information (AUI level). The same figure also shows an excerpt of the XML specification of the concrete user interface obtained. Comparing the XHTML page shown in Figure 3 and the corresponding logical description shown in Figure 4 we can see that the reverse engineering of the page generates a main column grouping (Grouping\_1\_2) of two main groupings: Grouping\_1\_3 composing the interactors corresponding to the image and text at the top of the page and Grouping\_1\_6 containing two further compositions: Grouping\_1\_7 collects the interactors obtained from reversing the links of the navigation menu on the left of the page, while Grouping\_1\_11 composes the text introducing the form and the *Relation* that contains all the interactors corresponding to the form elements collected in Grouping\_1\_12. The form commands submit and reset have been reversed into the corresponding activators and collected into Grouping\_1\_13. In the XML specification shown on the right side of Figure 4 we can

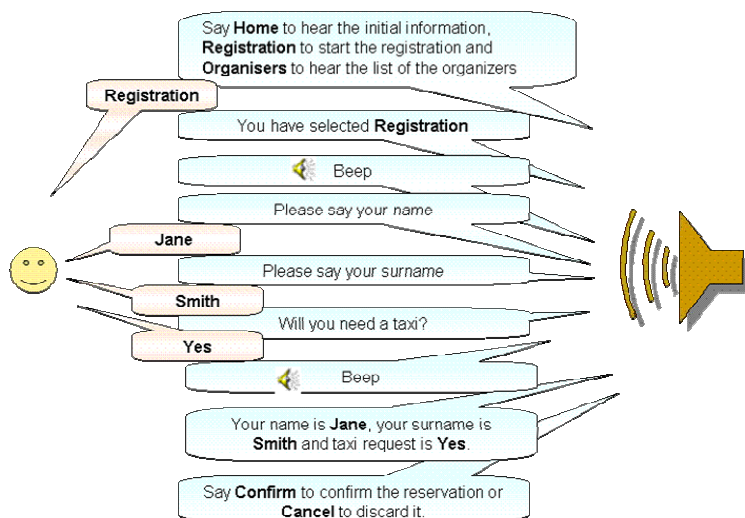


Fig. 5. Vocal VXML interface example

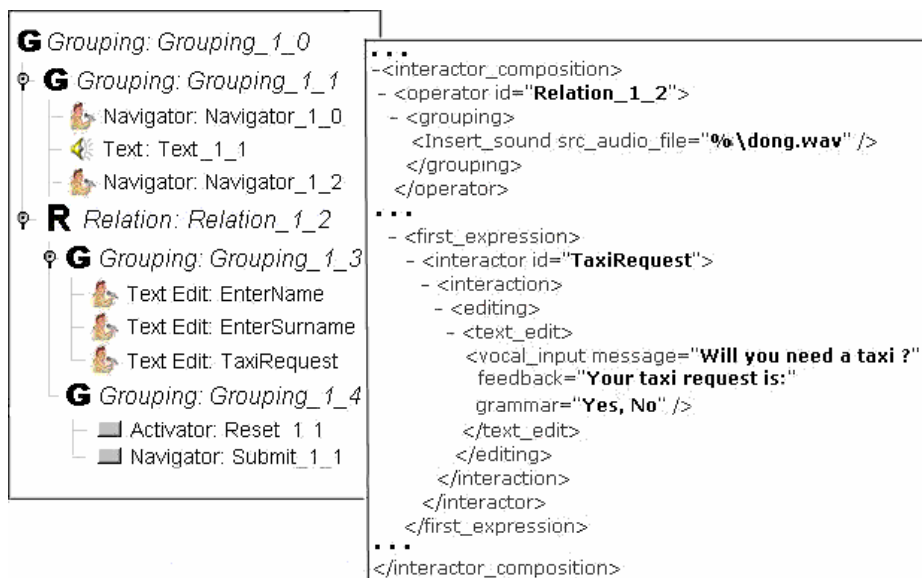


Fig. 6. An abstract description of the vocal example (left) and an excerpt of the corresponding XML concrete description (right)

see excerpts of the corresponding concrete description. In particular, the XML shows how the Relation composition operator is implemented by a form (*form1*) and it is connected to a *registrationDone* presentation through the *Submit1\_1* button, together with the concrete interface details of the SingleSelection element named *taxiRequired*.

Figure 5 shows the vocal VoiceXML version of the simple application considered. The vocal interface starts by asking the user which dialogue to start with. Then, it accesses the registration dialogue as requested and continues to prompt for information to fill in the vocal form and then submit it. Figure 6 provides a representation of the result obtained by reversing the VoiceXML application into a Concrete User Interface. The tree view shows the abstract elements, while the XML code excerpt shows the lower level concrete details. For example, the figure shows how the part of the dialogue delimited by the two “Beep” sounds has been reversed into the grouping `Grouping_1_3`. The concrete interface can implement the grouping operator in different ways (see Table 2, *subdialog* element), in this case we see that the “insert sound” option has been recognized. In Figure 6 we can also see an excerpt of the concrete specification concerning the part of the dialogue that prompts for the taxi option. The XML excerpt shows the message that the vocal interface uses both for prompting and for giving feedback to the user. Moreover, it also supports the recognition of the grammar associated to this particular vocal field.

## 9 Abstract Description to Task Model Transformation

The task models that we consider are specified in the ConcurTaskTrees (CTT) notation [12], which describes them in a hierarchical format with various temporal relations that can be indicated among tasks. In addition, a number of attributes can be specified for each task. A CTT task is characterised by its “category” and “type”.

The category indicates how the task performance is allocated and can take the following values:

- *Abstraction*: for higher level tasks with subtasks that do not have the same type of allocation. This category of task is associated with composition operator elements in the logical interface specification and therefore, it might be associated to the overall access to one presentation.
- *Interaction*: for tasks obtained by reversing interaction interactor elements.
- *Application*: for tasks obtained by reversing only-output interactor elements.

The root node of the task model is an abstraction task representing the whole application. As the whole application is generally composed of several presentations, the ReverseAllUIs tool starts building the task model associated to each presentation. Each presentation of the Abstract User Interface can contain both elements that are elementary interactor objects or composition operator elements.

The composition operators can contain both simple interactors and, in turn, multiple composition operators. Each composition operator in the logical user interface is reversed into an abstract task node, whose children are the tasks obtained by reversing the elements to which the abstract composition operator applies. The reversed children are connected through CTT temporal operators depending on the type of composition operator, as indicated in Table 4. For instance, if there are several objects in the same presentation and no constrain is put on the sequence about how the user is expected to interact with the different objects in the presentation, this behaviour will be translated by means of a concurrent CTT operator, which models the possibility of interacting in any order with the different objects.



**Table 4.** Reversing CUI composition operators into CTT temporal operators

<b>Abstract Composition Operator</b>	<b>CTT Temporal Operator</b>
Grouping	Concurrency
Ordering	SequentialEnabling
Hierarchy	SequentialEnabling or Concurrency
Relation	<p>Concurrency (among elements contained in the &lt;first_expression&gt; tag)</p> <p>Disabled by (elements contained in &lt;second_expression&gt; tag)</p>

Each elementary interactor is reversed into a CTT basic task, whose category is identified through the rules explained before. Further rules exist for reversing elementary interactors. For instance, if an interactor supports an activity that might be or not carried out by the user, then such interactor will be reversed onto an optional task. Also, elementary abstract interactors can be mapped onto elementary tasks by considering the type of activity supported by the task, which can be specified with CTT notation: for instance, a text\_edit AUI object will be mapped onto an interaction task having “*Edit*” as its type of activity. As a particularly interesting case of elementary interactor we consider the reverse engineering of navigators, which are objects allowing moving from one presentation to another one, and therefore, their reverse engineering involves both the presentation to which the navigator belongs and the presentations that it is possible to reach through it. The basic rule that has been identified for reverse engineering navigators is that elementary interaction tasks corresponding to navigators can disable the set of tasks associated with the current presentation and enable the next presentation. Once all single presentations have been reversed, the corresponding CTT subtrees must be composed to build up the whole application task model. The presentation subtrees are inserted, directly or grouped through a further abstraction node, as children of the root.

We describe how to reverse navigators by considering the example page considered in Figure 3. From the point of view of the abstract user interface such presentation can be seen as a presentation P1 connected to more than one presentation. Referring to Figure 7, such presentations are respectively reversed into the abstract tasks *Access Form Results*, *Access Home Page*, *Access Organisers Page*. Then, the latter presentations can be accessed by means of navigators which are reversed into corresponding interaction tasks, in our example they respectively correspond to *Select Send Form*, *Select Home*, *Select Organisers*. The fact that through navigators it is possible to reach the corresponding different presentations is modelled by connecting such tasks to the correspondingly related abstract tasks through a *SequentialEnabling* operator (represented by the >> symbol), and forming in turn three higher level abstract tasks, which in our case correspond to *Send Form*,

*Access Home* and *Access Organisers* tasks. Such tasks will be in turn connected each other through a *Choice* temporal operator (represented by the  $\square$  symbol, see Figure 7) to model the fact that the user can select only one of these paths. The abstract task obtained by such composition is in turn connected through a disabling operator with the subtree derived by reverse engineering the other elements belonging to the presentation. The disabling operator models the fact that when the user selects the navigation to a different page, it will disable the other elements in the presentation.

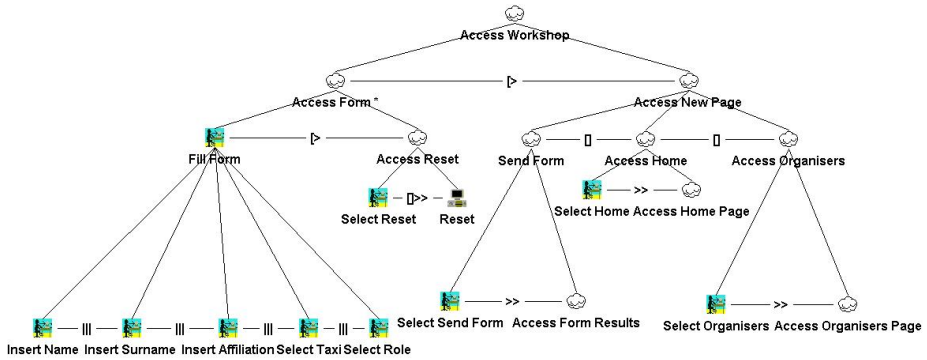


Fig. 7. Reverse engineering of multiple connections

As another type of navigator we consider the case when a presentation contains at least one link to a page external to the current application: in the task model an interactive task, called *Select External Link*, is added as subtask of the node grouping all the subtasks obtained by reverse engineering the whole application, which indicates that at this point the user leaves the application.

The recursive rules used in reversing the abstract logical description into the corresponding task model can generate task models with more nodes than what is strictly required. It may happen to find out *abstraction* tasks having only one child. In this case, the abstract task is removed and the child node is raised one level up. The CTT description language requires specifying the parent and sibling nodes for each task, hence, while removing a task from the tree and replacing it with its child, relationships among nodes must be updated.

## 10 Application of Reverse Engineering in Ubiquitous Environments

Reverse engineering can be used to support semantic redesign. In semantic redesign the basic idea is to transform the logical specification for a platform into a logical specification for a different one according to a number of design criteria.

Another useful application of reverse (and forward) engineering, combined with semantic redesign is the generation of migratory user interfaces. They are interfaces that can migrate among different devices while adapting to the characteristics of the

target platform and maintaining task continuity, so that the users have not to restart from scratch their activity when they change the device after a migration request.

We have developed a migration environment based on a proxy/migration server to which users have to subscribe for accessing Web applications through it. If the interaction platform used is different from the desktop, the server transforms the considered page by building the corresponding abstract description and using it as a starting point for creating the implementation adapted for the device accessing it. Also, in order to support task continuity, when a request of migration to another device is triggered, the environment detects the state of the application modified by the user input (elements selected, data entered, ...) and identifies the last element accessed in the source device. Then, a version of the interface for the target device is generated, the state detected in the source device version is associated with the target device version so that the selection performed and the data entered are not lost. Lastly, the user interface version for the target device is activated at the point supporting the last basic task performed in the initial device.

## 11 Conclusions and Future Work

In the paper we have presented the ReverseAllUIs environment supporting reverse engineering of user interfaces for different platforms and modalities (graphical and voice).

These features make the tool useful in ubiquitous environments, which are characterised by the presence of various types of interaction platforms.

The logical descriptions obtained in this way can be used for many purposes. One typical use is to exploit them in order to obtain user interfaces for different platforms by exploiting the semantic information reconstructed in order to obtain more meaningful results (through semantic redesign [10]) when deriving implementations for different target platforms. The task models obtained can also be used to support usability evaluation.

Future work will be dedicated to further increasing the number of interactive platforms and modalities supported by the reverse engineering tool. We also plan to develop a Web user interface of the reverse engineering tool so that users can access it remotely, indicate the URL of a web site and receive back the specification of the corresponding logical abstractions requested.

## References

1. Berti, S., Correani, F., Paternò, F., Santoro, C.: The TERESA XML Language for the Description of Interactive Systems at Multiple Abstraction Levels. In: Proceedings Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages, pp. 103–110 (May 2004)
2. Bouillon, L., Vanderdonckt, J.: Retargeting Web Pages to other Computing Platforms. In: Proceedings of IEEE 9th Working Conference on Reverse Engineering WCRE 2002, Richmond, 29 October-1 November 2002, pp. 339–348. IEEE Computer Society Press, Los Alamitos (2002)

3. Bouillon, L., Vanderdonckt, J., Chieu Chow, K.: Flexible Re-engineering of Web Sites. In: Proceedings of the International conference on Intelligent User Interfaces (IUI 2004), Madeira, pp. 132–139. ACM Press, New York (2004)
4. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers* 15(3), 289–308 (2003)
5. Canfora, G., Di Santo, G., Zimeo, E.: Toward Seamless Migration of Java AWT-Based Applications to Personal Wireless Devices. In: Proceedings WCRE 2004, pp. 1–9 (2004)
6. El-Ramly, M., Ingiski, P., Stroulia, E., Sorenson, P., Matichuk, B.: Modeling the System-User Dialog using Interaction Traces. In: Proc. of the eighth Working Conference on Reverse Engineering, Stuttgart, Germany, 2-5 October 2001, pp. 208–217. IEEE Computer Soc. Press, Los Alamitos (2001)
7. Gaeremynck, Y., Bergman, L.D., Lau, T.: MORE for less: model recovery from visual interfaces for multi-device application design. In: Proceedings of the international conference on Intelligent user interfaces, Miami, Florida, USA, January 2003, pp. 69–76. ACM Press, New York (2003)
8. Hudson, S., John, B., Knudsen, K., Byrne, M.: A Tool for Creating Predictive Performance Models from User Interface Demonstrations. In: Proceedings UIST 1999, pp. 93–102. ACM Press, New York (1999)
9. Moore, M.M.: Representation Issues for Reengineering Interactive Systems. *ACM Computing Surveys Special issue: position statements on strategic directions in computing research* 28(4), article # 199 (December 1996)
10. Mori, G., Paternò, F.: Automatic semantic platform-dependent redesign. In: Proceedings Smart Objects and Ambient Intelligence 2005, Grenoble, pp. 177–182 (October 2005)
11. Paganelli, L., Paternò, F.: Automatic Reconstruction of the Underlying Interaction Design of Web Applications. In: Proceedings Fourteenth International Conference on Software Engineering and Knowledge Engineering, pp. 439–445. ACM Press, Ischia (2002)
12. Paternò, F.: Model-based design and evaluation of interactive applications. Springer, Heidelberg (1999)
13. Stroulia, E., Kapoor, R.V.: Reverse Engineering Interaction Plans for Legacy Interface Migration. In: Proceedings of CADUI 2002, pp. 295–310 (2002)
14. Szekely, P.: Retrospective and Challenges for Model-Based Interface Development. In: 2nd International Workshop on Computer-Aided Design of User Interfaces. Namur University Press, Namur (1996)
15. Xiang, P., Shi, Y.: Recovering semantic relations from web pages based on visual cues. In: Proceedings of the 11th international conference on Intelligent user interfaces, Sydney, Australia, January 29-February 01, 2006, pp. 342–344 (2006)

# Intelligent Support for End-User Web Interface Customization

José A. Macías<sup>1</sup> and Fabio Paternò<sup>2</sup>

<sup>1</sup> Universidad Autónoma de Madrid. Ctra. De Colmenar, Km. 15,  
28049 Madrid, Spain

<sup>2</sup> ISTI-CNR. Via G. Moruzzi, 1,  
56124 Pisa, Italy

j.macias@uam.es, fabio.paterno@isti.cnr.it

**Abstract.** Nowadays, while the number of users of interactive software steadily increase, new applications and systems appear and provide further complexity. An example of such systems is represented by multi-device applications, where the user can interact with the system through different platforms. However, providing end-users with real capabilities to author user interfaces is still a problematic issue, which is beyond the ability of most end-users today. In this paper, we present an approach intended to enable users to modify Web interfaces easily, considering implicit user intents inferred from example interface modifications carried out by the user. We discuss the design issues involved in the implementation of such an intelligent approach, also reporting on some experimental results obtained from a user test.

**Keywords:** End-User Development, Intelligent User Interfaces, Model-Based Design of User Interfaces, Programming by Example.

## 1 Introduction

Very often, customizing software applications implies extra knowledge and effort that some users cannot simply afford. Providing users with real authoring facilities is not yet as widespread as one would expect. Most of the existing approaches pay poor attention to end-users, and the ease of customization of commercial applications is still barely visible.

In general terms, the explicit customization of interactive applications requires considerable skill in programming and technology. Some preliminary studies indicate that these limitations in user development activities are not due to lack of interest, but rather to the difficulties inherent in interactive development [14]. Some development tools already offer support for high-level functionality, but most of these tools are not aimed at non-programmers.

Our research is aimed at addressing such problems by providing end-users with easy and automatic mechanisms to customize Web applications. Our research experience is in Model-Based User Interfaces [12] design combined with End-User

Development [6] techniques to help users interact with computers through intelligent WYSIWYG authoring environments [9], [10]. To this end, one of our main concerns is end-user development environments oriented to nomadic Web applications [3], which are Web applications accessible through a variety of platforms, including wireless devices supporting mobile users.

In this work, our effort is aimed at allowing users to provide modification examples of nomadic interfaces in such a way that the system be able to learn and generalize customizations automatically. From this point of view, our system is based on Programming by Example (PBE) mechanisms. Programming by Example [4], [7] is one of the most relevant efforts in EUD for obtaining a real trade-off between ease of specification and expressiveness. In our approach the user provides the system with an example of what s/he wants to modify by means of a standard authoring tool and then the system analyses the modifications at the server side for a given user and platform.

In particular, in the paper we report on some design issues addressed in our environment, combined with a further analysis of the type of reasoning that our system is able to apply. We also report on empirical system verification through an experiment carried out with real users. The paper is structured as follows. Section 2 introduces related work and discusses it. Section 3 describes our approach in further detail. Next, Section 4 reports on design and architectural issues. Section 5 provides further detail and introduces rule firing, describing an experiment carried out with real users with the aim of verifying the proposed approach. Lastly, Section 6 draws some conclusions and provides indications for future work.

## 2 Related Work

Intelligent rule-based systems have been traditionally used in Programming by Example research mostly due to the execution speed and simplicity they supply. Other complex machine-learning algorithms usually suffer from high error rates and low generalization in real time interaction with users.

Some early tools developed by Myers's group, such as Peridot [11] applied a rule-based approach. Peridot is more oriented to supporting user interface design and uses about fifty hand-coded Interlisp-D rules to infer the graphical layout of the objects from the examples. This type of system has the disadvantage of being subjected to rule-based heuristics that generalize from a single example, which implies that only a limited form of behaviour can be generalized since the system can only base its guess on a single example. More complex behaviours are either not treatable or must be created manually by editing the code generated by the PBE system. These types of systems are mostly focused on static knowledge and can be considered domain-dependent. In contrast, our system proposes an approach to build dynamically knowledge, in which a complete rule structure is created in order to consider different kinds of conceptual knowledge that can be updated from time to time through an evolving approach.

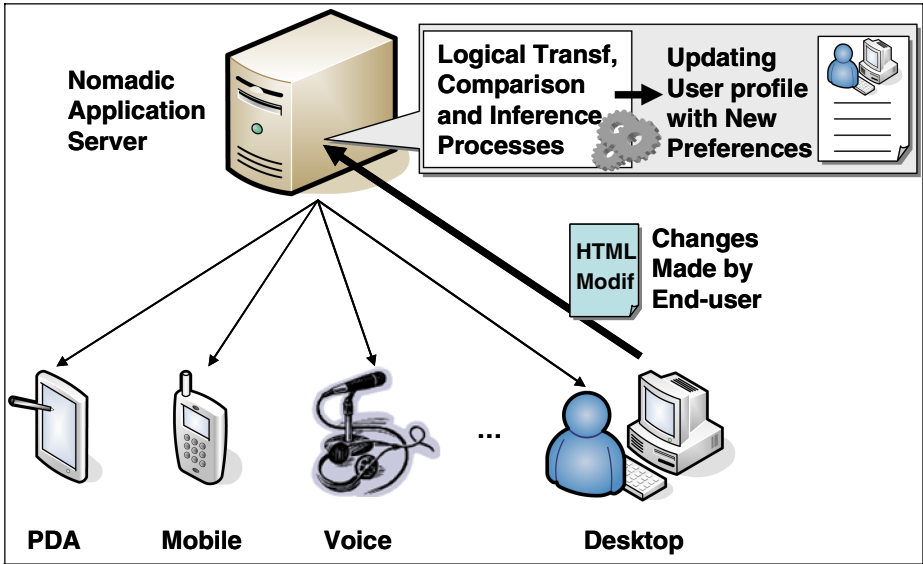
Recent systems such as AgentSheets [13] are examples of commercial EUD approaches for building intelligent interfaces. AgentSheets is a simulation environment that allows the user to create advanced simulation scenarios by defining intelligent agents and behaviour separately. AgentSheets combines PBE with graphical rewrite rules into an end-user programming paradigm. Like AgentSheets, our approach applies semantic rules for dealing with high-level behaviour. As pointed out by AgentSheets' authors, a first step toward creating more usable and reusable rewrite rules is to move from syntactic rewrite rules to semantic ones, including semantic meta-information. The lack of semantics not only makes reuse difficult, but also creates a significant problem for building new behaviours from scratch, reducing significantly the scalability of a PBE approach as well. Additionally, in our approach we consider different levels of knowledge and behaviour, dividing rules and facts into different conceptual levels that will help achieve an in-depth analysis automatically, inferring with accuracy the user's intents in order to obtain an evolutionary approach.

Another related work is DESK [8], which uses domain knowledge for characterizing changes from a dynamically generated interface, making minimal assumptions about the final user's skills on programming and specification languages. DESK uses the PEGASUS specification based on domain ontologies in order to specify explicit knowledge of both presentation and domain information separately [9]. DESK tracks and records information from user actions and builds a monitoring model specified in XML. This information is sent to the back-end application, which processes in turn the monitoring model and applies different heuristics by using domain knowledge. As a result of the inference process, the underlying models of PEGASUS (domain, presentation) are modified taking into account each change the user performs on the Web page. Our approach overcome the DESK's limitations by detecting user intents automatically, comparing original and modified interface logical specifications and with no need of having a specific authoring client application. In order to get maximal high-level domain independence, changes by users are obtained through processing directly a logical user interface description specified in TERESA XML [2]. By contrast, DESK is limited to deal with HTML code modifications, which are later processed to obtain meaningful information by means of fixed heuristics. We exploit the information provided by the logical interface description to obtain semantic information. The knowledge management is improved by defining different levels of knowledge that are applied to better characterize and obtain evolving knowledge for future inferences.

To summarise, the work presented in this paper provides a novel solution with respect to approaches such as DESK and AgentSheets because it applies reverse engineering tools able to build automatically descriptions at different abstract levels represented using TERESA XML, which is a domain-independent modelling language. Such semantic information is then exploited in our intelligent approach to supporting user customization.

### 3 Our End-User Approach

Our system supports a EUD framework intended to provide the user with an easy mechanism to freely customize Web interfaces.



**Fig. 1.** Our approach can be used to customize user interfaces for different platforms. Users make changes to express their preferences and then upload the modifications onto the server, which infers customizations from the changes accomplished.

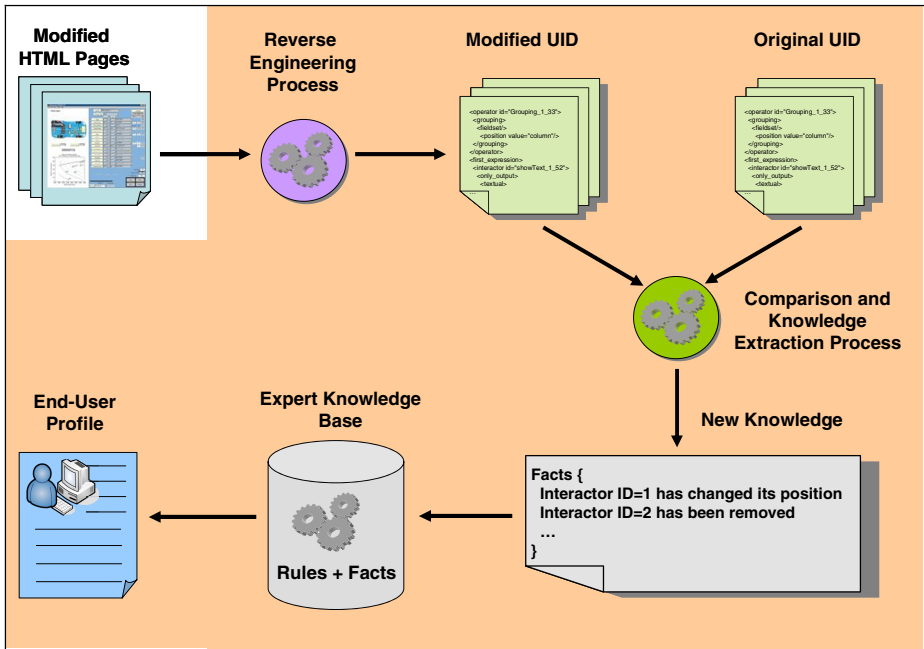
In particular, our approach supports the following steps (see Fig. 1):

1. The Web server of applications generates an interface adapted to the platform accessed by the user, so s/he can access the application using a desktop computer, laptop, mobile, PDA and vocal interface.
2. The end-user navigates through the information and, at some point, s/he decides to modify something by using a standard Web authoring tool (such as Macromedia Dreamweaver) that supports modifications by direct manipulation of the interface elements.
3. Once the user has finished the changes, s/he sends the modified page to the server, by using a specific Web application in which s/he first needs to login.
4. The server receives the Web page and then starts the inference process to identify the user's preferences.
  - a) First, the server transforms the modified page into a logical description stored into a XML file, using the reverse mechanism developed by our group [1]. The resulting file contains the user interface description of the page in terms of language-independent elements.
  - b) Then, the system compares the logical description corresponding to the modified page with the logical description of the previously generated one.
  - c) In the comparison process, the system also generates high-level information in order to find out meaningful information about the user's intents, and also to identify general user preferences.
  - d) At the end of the process, the system builds an End-User Profile taking into account all this high-level information inferred, as well as others previously generated.



5. The End-User Profile is then used to generate again the Web interface, taking into account preferences and personal customization. The system stores an End-User Profile for each user and platform, controlling which aspects of the generated interface could be significant for each one.

The most relevant information stored in the End-User Profile is the set of Interface Rules. Such rules are inferred from the logical descriptions' comparison and aim to reflect the knowledge acquired from the user's changes. The rules are used for driving the generation of the Web pages after the changes, customizing the Web presentation and navigation depending on the inferred preferences. The rules are based on knowledge acquisition algorithms and targeted at obtaining information regarding user intents in order to characterize some preferences for customization purposes. Such information can be modelled by means of both a knowledge base and a set of rules to be applied when new information about user modifications is identified.



**Fig. 2.** Comparing both modified and original user interface descriptions, the system automatically extracts information in order to feed the expert system, generates the information to reason about and updates the user profile

The intelligent approach is implemented by using an expert system, where the knowledge can be modelled conveniently and the inference takes place more efficiently. Particularly, it supports a framework able to deal with facts and rules, as well as the capability to populate the knowledge base with new information from time to time (i.e. evolutionary approach). In our approach, the facts represent the information coming from the user's modifications. This information is extracted by

comparing both modified and original logical interface descriptions (see Fig. 2). On the other hand, the rules are conditions used to get semantic information from the facts, that is, from the syntactical changes the user makes to the presentation and from other high-level information available in the expert knowledge base. The rules will reflect not only user changes but user information about the platform (Desktop, Mobile, and so on). By means of an evolutionary approach, continuous production and modification of facts helps the system refine the user's preferences and extract accurate information as interaction evolves.

In order to obtain greater precision and accuracy in the inference process, we use Jess [5], a Java framework which includes the Rete pattern matching algorithm for implementing rule-based (expert) systems. This algorithm was originally designed by Forgy at Carnegie Mellon University. It provides the basis for an efficient implementation of an expert system and is designed to sacrifice memory for increased speed.

### 3.1 Interface Knowledge Modelling and Construction

We base on TERESA XML language [2] for detecting changes on logical user interface descriptions. In this specification language, a user interface can be described at different abstraction levels. The concrete level is platform-dependent but implementation-language-independent, while the abstract level is also platform-independent. In both cases the user interface is composed of interactors and composition operators, indicating how to structure their composition. There are different one to many relationships between interactors at the abstract and the concrete level (e.g. a navigator can be a text link, an image link or a button), which indicate how an abstract interaction can be supported in a given platform at the concrete level. Modifications affecting the concrete level provide syntactical knowledge, while those that effect the abstract level provide semantic knowledge as the abstract level identifies the type of basic task associated with the interface element. We consider both kinds of modifications in order to construct a knowledge structure aimed to feed the expert system with suitable facts, activate expert rules and produce user customizations efficiently.

The conceptual levels in which the knowledge is structured is crucial. Thus, we need to consider the following steps in defining that knowledge:

- **Defining base knowledge** containing basic definition about user, platform and the previous knowledge on user modifications. This is the information that always remains in the expert system and is updated from session to session.
- **Defining syntactic knowledge** that contains facts and rules triggered by syntactical modifications to presentation elements such as concrete interactors and concrete composition operators. (e.g. when a concrete interactor changes the value of its attributes). Furthermore, this level of knowledge deals with the syntactic context associated with the concrete composition operators, detecting for instance when a concrete composition operator has changed its colour, alignment, justification and so on.
- **Defining semantic knowledge** for dealing with semantic information by taking into account the syntactic information already created. The semantic level uses the abstract platform-independent elements associated with interactors and composition

operators. For instance, it identifies when the number of interactors changes in one possible composition (i.e. ordering, hierarchy relation or grouping). The semantic level also constructs presentation context, that is, contextual information extracted from the surrounding elements where a change took place in the graphical interface. Presentation context allows the creation of expert rules based on contextual information that can be applied more than once.

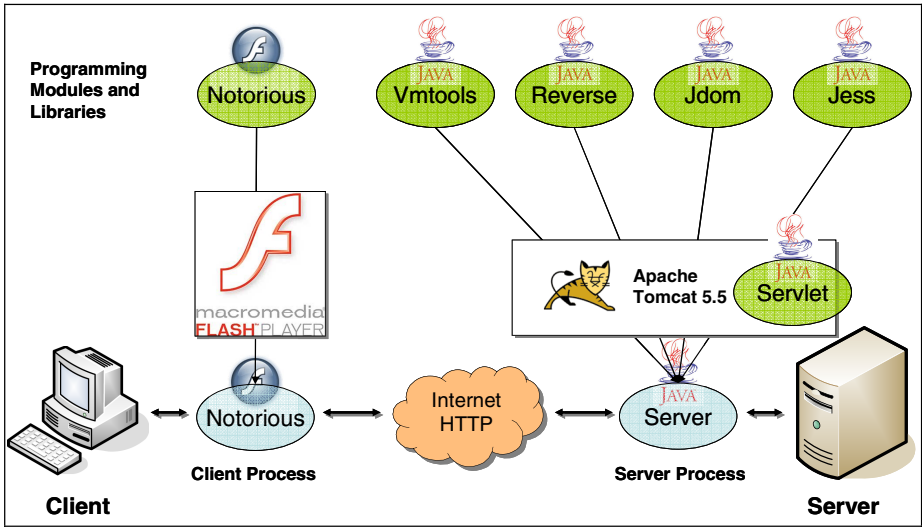
- **Defining expert rules** for dealing with further semantic aspects and characterizing user intents. The main goal of this level is to define both syntactic and semantic customization rules that will be deployed using the underlying knowledge available for the previous levels. Syntactic customization rules detect changes concerning concrete interactors and concrete composition operators, whereas semantic customization rules detect high-level changes affecting interactors and composition operators. For dealing with semantic customization rules, presentation context needs to be considered.

Knowledge construction is carried out progressively from the lowest levels to the highest ones. The knowledge constructed at lowest levels is basically composed of syntactic information automatically generated by the system. This information comes from the comparison of the specification of the concrete user interfaces before and after the user's changes and is related to the elements that the user implicitly manipulates when authoring a nomadic presentation. These elements are the concrete interactors and mostly indicate platform-dependent interaction techniques of different type (for instance, in a graphical desktop system concrete interactors can be Radio Button, List Box, Text Link, Button, Input Text and so on). All the changes concerning concrete interactors are added as syntactic knowledge in order to populate the expert system with detailed information about the type of concrete interactor, its implicit properties and so on. In concrete interface specifications, concrete interactors are composed through specific operators, in order to create relationships between different elements that will be presented for a platform and user. The concrete composition operators implement the abstract operators (grouping, hierarchy, ordering and relation) through constructs such as Fieldset, Unordered List, Ordered List, Table, Form, and so on.

On the other hand, the system extracts presentation context that is based on the abstract specification of the interface, which is platform-independent and hence useful in order to get high-level contextual information about the presentation. This allows defining more general rules that can be applied to similar presentation contexts more than once. The abstract information of both interactor and composition interactor is managed by the semantic level of the expert system. Actually, this information can be regarded as a knowledge add-on that is based on the syntactic information already added by the syntactic level. The semantic level is responsible for detecting when an interactor is moved from a composition operator to another, or when it is deleted or removed, generating knowledge that can even affect the task model of the application. The semantic level is also responsible for extracting presentation context, and then adding it to the system as semantic knowledge. However, the first and foremost goal of the semantic level is to populate the system with information that will be deployed by the expert level, in order to carry out generalization in applying advanced customization rules.

## 4 The Software Architecture

Our system was originally conceived as a client-server architecture, where two principal processes run on the client and the server side and communicate one another to carry through the approach here presented.



**Fig. 3.** The architecture of the system is mainly composed of a client and a server side, where two different processes run and communicate one another. The front-end sends to the server process the changes to be processed at the back-end of the application.

Fig. 3 depicts how the system is structured. At the client side, a Macromedia Web application called Notorious is executed. This application communicates with a server process, which is exported as a HTTP service by means of the Apache Tomcat Web Server 5.5. The process is a Java Servlet that is installed on the port 8080 of the server. The client application mainly consists of a user interface intended to identify the user when s/he connects to the server and uploads the modified Web pages. It also manages the feedback coming from the server process and visualizes the information reported (i.e. rules inferred and also the user interface descriptions). This application processes, by means of a XML connector, the user model from the server, and visualizes and stores such information properly. When the user decides to send Web pages using Notorious, this client application accesses the server. Then, the server takes up the request from the client application and in turn processes it, storing the Web page and generating the Concrete User Interface corresponding to the file that has been sent. Additionally, the Server routine compares both Concrete User Interface files (the original and modified one) and calls the expert system module to create new knowledge and obtain feedback to be sent to the client application. In doing so, the Server comprises the following modules:

- **Vmtools** is a library used to compare and obtain the differences from two XML files. The library comprises different classes and objects that can be used from a Java program. In our approach, this library was useful in order to compare the logical descriptions of the interface (the modified and the original one) and easily process the modifications by which the expert system is fed.
- **Reverse** is a Java library developed by our group which concerns the reverse-engineering routines for transforming HTML code into a logical user interface description called Concrete User Interface. Different methods are used to tidy and transform the code properly.
- **Jdom** is a Java library that comprises the routines used by Vmtools library. It is used to manipulate XML code and deal with XML-tree operations easily.
- **Jess** is the Java library used to deal with the expert system implementation. This library includes classes and objects to manipulate the inference engine called Rete algorithm, as well as the methods to activate, trace and deal with facts and rules in a nondeterministic way.

Additionally, the system uses the standard Java and Servlet libraries included in the standard edition of Java. Servlet routines are used to program different Web services in Apache Tomcat, so they can be regarded as a library as well.

## 5 Verification and Experimental Results

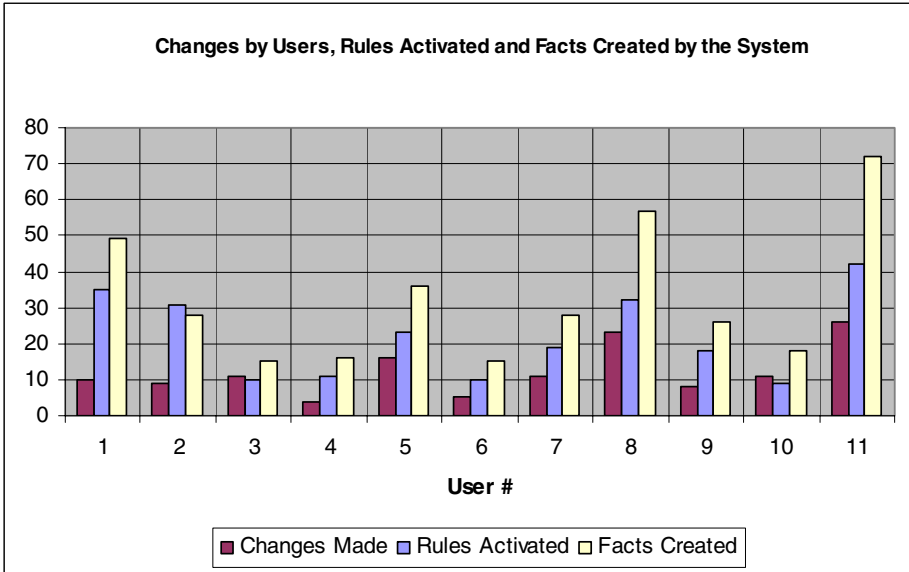
After the design of the system, one of our principal aims was to test the approach implemented. To this end, we carried out an experiment in order to check and obtain feedback on the methodology here proposed.

This experiment was mainly motivated by the need to measure the proposed rule-based approach. The test was aimed at detecting meaningful reactions of the system according to the user's modifications for a specific nomadic application. We recruited 11 participants from our institution, with heterogeneous scientific backgrounds, and asked them to freely customize a desktop Web museum application.

Based on different cases of use previously studied and analyzed, the expert system was programmed containing different kinds of expert rules, which can be divided into syntactic customization rules and semantic customization ones, as explained in Section 3.1. Furthermore, each rule has to be triggered at least three times to be considered a permanent customization, which the user can still turn on or off for future applications. In particular, a total of 14 syntactic customization rules and 10 semantic ones were created, with the intention of activating them according to the modifications performed by end-users. These included syntactic customization rules for detecting changes in text style preferences, interaction widgets and composition structures such as forms and fieldsets. On the other hand, semantic customization rules were also defined in order to deal with changes involving transformation, deletion and insertion of interactor groupings, as well as changes affecting composition operators that involve interactor repositioning. These reflect end-user preferences in navigation, ordering and hierarchical structure customization.

Additionally, the system was programmed to detect both user-dependent and user-independent customization. The user-dependent rules concern preferences associated

with a specific user and have been described previously. As for detecting user-independent preferences, the system checks whether the same rule is triggered by more than one user. User-independent tailoring helps define general changes to presentations for all users whenever the same rule is triggered by at least more than 5 users. At this point, the rule appears in every user profile and can be individually turned off whenever one user does not accept the changes.



**Fig. 4.** The system's response to user changes, where the number of changes made and rules activated are shown, along with the facts automatically created by the system throughout the experiment with 11 users

Fig. 4 shows the relation between the number of changes, the facts generated and the rules activated for each user during the experiment. At first sight, it seems clear that the more the changes made, the more facts and rules are activated. However, this relation is not always as linear as one might expect, since it mostly depends on the complexity of the changes performed. In the case of user #2, for instance, one can see that the number of changes is lower with respect to other users, but the number of facts and rules activated is instead higher. This is due to the fact that user #2 made a total of 9 changes, but all involved complex effects. These entail moving interactors, changing the navigational structure of the page, transforming composition operators and so on. This produced a high number of facts that had to be specified in terms of syntactic information and presentation context. In addition, the rules that had to deal with such changes were even more complex than simple syntactic ones, so that a chain of rules had to be activated to correctly detect the changes made by this user. In contrast, users #8 and #11 carried out a high number of changes (23 and 26, respectively) that generated a higher number of facts (57 and 72, respectively) created by the system, as well as a high rate of rule activations (32 and 42, respectively). In

these cases, most changes were syntactical, so the response of the system was quite proportional to the type and number of changes carried out by these users. In conclusion, it is possible to affirm that the response of the system is linear as long as the user's changes do not involve complex structural aspects. In any case, such complexity does not at all affect the system's performance and throughput.

In addition to semantic and syntactic rules, we also considered the number of times each type of change was made by the user. A customization is applied when a rule is triggered three or more times. Otherwise, the customization is considered pending for the time being. This mechanism helped us to classify pending and permanent customizations depending on their rule-activation frequency. From the total rule activations measured during the user sessions and depicted in Fig. 4, 80% corresponded to syntactic customization rules, whereas only 20% corresponded to semantic customization ones. Regarding syntactic customization rule activations, 64% can be considered pending, whereas only 36% were permanent. With respect to semantic customization rules, only 9% of activations were permanent, whereas 91% were considered pending.

### 5.1 Rule Activation

In the experiment, rules were activated by following different steps. Let us examine a piece of the output extracted from the expert system for one of the user tests, illustrating how rules are activated and detected by the system.

1)Change detection and contextualization	<pre> ==&gt; f-1 (MAIN::change (ID C1) (concrete_interactor Text Show_museum_info2) (change_type font_style_change bold) ==&gt; f-2 (MAIN::syntactic_context (ID SC1) (change C1) (from Presentation 2 FieldsetColumn 1) (above null) (below GraphicalLink 1 ) (user andrea) (platform Desktop)) ==&gt; Activation: MAIN::syntactic_change : f-2 ... </pre>
2)Syntactic customization rule activation	<pre> FIRE 19 MAIN:syntactic_change f-2 ==&gt; f-57 (syntactic_change_fact (syntactic_customization_rule6) (change C1) (syntactic_context SC1)) ==&gt; Activation: MAIN:: syntactic_customization_rule6 : f-57, f-54, f-51, f-44, ... </pre>
3)Pending and permanent rule activation	<pre> FIRE 20 MAIN: syntactic_customization_rule6 : f-57 <b>Pending Syntactic Customization (fired 1 times): Text style for Description Interactor will be bold</b> </pre>

```

FIRE 21 MAIN:
syntactic_customization_rule6 : f-54
Pending Syntactic Customization (fired 2
times): Text style for Description
Interactor will be bold
FIRE 22 MAIN:
syntactic_customization_rule6 : f-51
Permanent Syntactic Customization
(triggered more than twice): Text style
for Description Interactor will be bold

```

The output above has been divided into 3 different parts. The first part corresponds to the change detection process. This information is directly supplied by an algorithm that compares the logical descriptions of the interface (original and modified Concrete User Interface files) and extracts information about what has changed. Consequently, the first fact is added to the system (f-1), reflecting the change (font text style has changed to bold) as well as the concrete interactor affected (Text element called Show\_museum\_info2). In addition, the syntactic information about the change is also created as fact number 2 (f-2), reflecting the context of the change (in Presentation 2, in FieldsetColumn 1, where above there is nothing and below there is the GraphicalLink 1 element) and the platform and user who made the change (user Andrea on platform Desktop). This change activates an internal rule called syntactic change that deals with the previous information and tries to find a suitable match for the rule to be applied (either syntactic or semantic customization rule). For this case, the second part of the output shows that the system has detected a syntactic customization since the change made is likely to be considered syntactic (a text style has changed). Thus, a new fact has been created (f-57) that relates the change (C1), the syntactical context (SC1) and the syntactic customization rule to be activated (customization\_rule6). The syntactic customization rule number 6 deals with text style changes, and will be activated for the current fact (f-57) as well as for others which correspond to the same change and syntactic context (f-54, f-51, f-44, ...). The third part of the output depicts the activation of customization rule number 6 for each change (fact) previously specified in the expert system. In this way, fact f-57 triggers a pending syntactic customization rule for a description interactor (the Text concrete interactor). This pending rule is triggered again for a different fact (f-54). Then, at the third matching (fact f-51), the pending customization rule was turned into a permanent one. This means that the description interactor, in the context observed (in this case the first occurrence at the beginning of a page), will appear in bold style. Consequently, this preference will be included in the user profile and can be turned off later on by the user.

The detection of semantic rules implies a similar sequence of facts creation and rules activations. In contrast, semantic rules require identification of the presentation context. The following output shows an example extracted from the user test.



```

FIRE 5 MAIN:syntactic_change f-14
==> f-27 (syntactic_change_fact (presentation_context
PC1) (change C7) (syntactic_context SC7))
==> Activation: MAIN::semantic_change : f-27 ...
FIRE 22 MAIN:semantic_change : f-27
==> f-36 (Presentation_Context (ID PC1) (Change_type
Insertion) (From Grouping FieldSet Grouping FieldSet)
(Above Navigator GraphicalLink Navigator GraphicalLink)
(Below null))
==> Activation: MAIN::semantic_customization_rule1: f-36
FIRE 23 MAIN: semantic_customization_rule1 : f-36
Pending Semantic Customization (fired 1 times):
Navigational Preferences have been changed by user
(inserted Navigator)

```

The piece of output above shows how initially the system mapped (f-14) a syntactic change (C7) to the context (SC7). Later on (FIRE 22), the system realised that such change regards the insertion of a navigational element, which is an interactor, and has semantic implications for the system. To this end, a new element appears (presentation\_context PC1), identifying that a presentation context is needed in order to correctly identify this change. This causes the creation of a new fact (f-27) that involves semantic changes. Next, an internal rule (semantic\_change) is called in order to extract the presentation context for such change. The presentation context is created in the form of a new fact (f-36), which reflects the context of the change in terms of abstract elements (Grouping, Navigator, and so on). Lastly, semantic rule number 1 is activated by means of the creation of the previous fact (FIRE 23), and thereby activates a pending rule once the presentation context has been successfully matched. It is worth noting that this customization reflects the fact that the user decided to change the navigational structure by adding a new navigator (a link).

## 5.2 Comparative Example

Fig. 5 shows one of the pages of the marble museum used in the user test (window at the top), as well as three pages (at the bottom) corresponding to three different modifications made by three different users. Although there are some similarities between some of the changes, the modifications differ from one another significantly. The dotted text box near each window describes the most important changes effected by each of the three users. Let us see in detail how the system reacts to each change for each modified page in Fig. 5.

In the first modified page (#1), the main change is the addition of a grouping consisting of a new navigational set inserted on the top of the page. This action stems from the fact that the user copied and pasted a fieldset, containing the navigational links of the home page of the museum, into each page with the idea of navigating everywhere from every page without the need to go back to the home page. In this case, the system activates different pending semantic customization rules, since the change mainly affects a grouping composition operator and thus can be considered a semantic change rather than a syntactic one. The system's reaction to such change

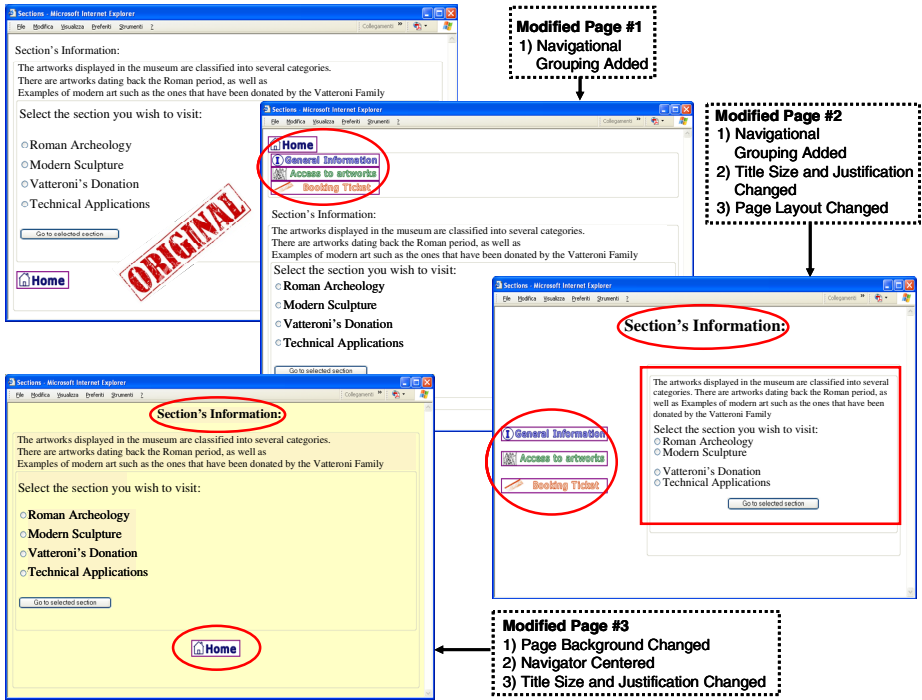


Fig. 5. Screenshots of 3 different pages modified by users during the test. The original page is at the top, whereas the other three windows depict the diversity of modifications made by users.

appears automatically specified by the system as rule firing numbers 9, 13 and 17. These can be summarized as follows:

```

FIRE 9 MAIN:semantic_customization_rule6 : f-22
Pending Semantic Customization (fired 1 times): New
Interactors have been added to an existing Grouping by
user (Grouping Add-on) ...
FIRE 13 MAIN:semantic_customization_rule6 : f-24
Pending Semantic Customization (fired 2 times): New
Interactors have been added to an existing Grouping by
user (Grouping Add-on) ...
FIRE 17 MAIN:semantic_customization_rule6 : f-26
Permanent Semantic Customization (triggered more than
twice): New Interactors have been added to an existing
Grouping by user (Grouping Add-on)
    
```

The above output describes how the system detected a semantic customization rule related to a grouping change (customization\_rule6). This process is carried out after analysing the change in the grouping composition operators and obtaining the presentation context involved in each change. Lastly, the system converted the pending rule into a permanent one. This is due to the fact that the user decided to

make the same change three times to more than one Web page, as shown in firing numbers 9, 13 and 17.

In the second modified page (#2), the user made different changes, some involving semantic changes and others only syntactic ones. The semantic changes were related, once again, to the movement of elements as well as changes in grouping composition operators. In this case, one can see how the user decided to copy and paste the navigational set from the home page into the modified one, and then made changes to the page layout as well. Three different semantic customization rules were activated. These rules were applied to changes associated with modification, movement and distribution of interactor groupings. Additionally for this user and presentation, some syntactic changes were detected, meaning that the user also decided to change the text size and justification for the description element. The following rules were eventually activated:

```
FIRE 16 MAIN::semantic_customization_rule4 : f-43
Pending Semantic Customization (fired 1 times): Grouping
movement into another by user (Grouping Movement) ...
FIRE 18 MAIN::semantic_customization_rule5 : f-44
Pending Semantic Customization (fired 1 times): Grouping
layout has been set to horizontal by user (Grouping
Distribution) ...
FIRE 22 MAIN::semantic_customization_rule6 : f-46
Pending Semantic Customization (fired 1 times): New
Interactors have been added to an existing Grouping by
user (Grouping Add-on)
```

In this case, three different semantic customization rules were activated (4, 5 and 6). These rules deal with detecting changes in, and movement and distribution of, groupings. Like in the first modified page, the system firstly detected the change, obtained the syntactic and presentation context and then detected a matching in the presentation context that triggered this pending rule multiple times. This time, no pending rule was turned into a permanent one since the user only decided to make the change more than twice on different contexts, hence the system did not consider it to be the same change.

Additionally for this user, some syntactic changes were also performed, leading to the following output from the system:

```
FIRE 31 MAIN:syntactic_customization_rule6 : f-36
Pending Syntactic Customization (fired 1 times): Text
Font justification for Description Interactor will be
centred
FIRE 33 MAIN: syntactic_customization_rule1 : f-35
Pending Syntactic Customization (fired 1 times): Text
Size for Description Interactor will be 14
```

The output above reflects that the user also decided to change the text size (to 14 points) and justification for the description interactor (Text) on the top of the page. In this case, two syntactic pending customization rules were activated (6 and 1) that deal

with text justification and size, respectively. As before, no permanent execution was considered for such changes either.

The last page (#3) modified by the user contained mostly syntactic changes: only one navigator that the user centred, the description element at the page top, which the user also centred and enlarged in size, and a change to the page background colour. For these, the reaction of the system was to activate syntactic customization rules as follows:

```
FIRE 10 MAIN:syntactic_customization_rule5 : f-27
Pending Syntactic Customization (fired 1 times): Page
Background will be #FCF4CD ...
FIRE 11 MAIN:syntactic_customization_rule5 : f-26
Pending Syntactic Customization (fired 2 times): Page
Background will be #FCF4CD ...
FIRE 12 MAIN:syntactic_customization_rule5 : f-22
Permanent Syntactic Customization (triggered more than
twice): Page Background will be #FCF4CD ...
FIRE 13 MAIN:syntactic_customization_rule1 : f-25
Pending Syntactic Customization (fired 1 times): Back
Graphical-Link Navigator alignment will be centred ...
FIRE 14 MAIN:syntactic_customization_rule1 : f-24
Pending Syntactic Customization (fired 1 times): Text
size for Description Interactor will be 18 ...
FIRE 16 MAIN:syntactic_customization_rule4 : f-23
Pending Syntactic Customization (fired 1 times): Text
font justification for Description Interactor will be
centred
```

As the previous cases, the system firstly processed the changes and then triggered the syntactic customization rules for this case (5, 11, 1 and 4). The first syntactic customization rule concerned the change in the background, as the user decided to set another colour. As one can see, this pending rule became permanent since the user carried out this same change to more than two pages. This means this customization was stored in the user profile. Additionally, the user decided to centre the back navigational link at the bottom, which triggered the syntactic customization rule 11. Some other temporary customization activations were carried out as well: these affected text style and justification and concerned the description interactor at the page top. These last changes were not considered permanent, since the user decided to perform them less than three times.

## 6 Conclusions and Future Work

Customization of software artefacts is commonly considered as an activity that requires specialized knowledge that most end-users do not have. This is mainly due to the fact that authoring environments require manipulating programming languages and abstract specifications. Although much progress has been made by commercially

available development tools, most of them lack not functionality, but rather ease-of-use [15].

Our approach overcomes such limitations and provides easy and efficient mechanisms based on Programming by Example techniques, where the user provides the system with example changes and the system generates customizations that will be applied automatically in future interaction. More concretely, the user carries out changes to applications generated by a server for a specific platform using any commercial authoring tool, and then s/he sends the modified pages to the server. Lastly, the system processes all the pages and tries to infer meaningful customizations to be applied in the future. Instead of forcing end-users to learn programming languages and complex specifications, our system carries out Web customization automatically by extracting meaningful information from the user's changes that will be stored in a profile and used to support future sessions.

We report on a detailed example of activations extracted from a user test, which has been introduced and further commented. Although only permanent activations were taken into account for a specific user and platform, more general information can be extracted. Collective knowledge can be deployed to detect general preferences by simply matching coincidences from more than one user. In the previous examples some changes can be understood to be general semantic customizations when the same rule is activated consistently by different users. For instance, as depicted in Fig. 5, modifications #2 and #3 reflect that both users made changes affecting the description element located at the page top, specifically changes concerning font size and justification. Independent of the user and platform, this information can be used to activate more general rules that can be triggered when the same modifications occur for more than one user. Moreover, general rules could be defined, for example "If activation X is converted from pending into permanent for at least N users, then this rule can be included in every user profile as a general preference". This information is easy to obtain by our approach, since the expert system can be regarded as a database where queries can be executed in order to mine the desired information from the knowledge stored. Additionally, other high-level rules can be defined to detect problems concerning page design. We are carefully studying and analysing such issues in order to further improve our system.

**Acknowledgments.** The work reported in this paper has been supported by the European Training Network ADVISES, project EU HPRN-CT-2002-00288, and by the Spanish Ministry of Science and Technology (MCyT), projects TIN2005-06885 and TSI2005-08225-C07-06.

## References

1. Bandelloni, R., Mori, G., Paternò, F.: Reverse Engineering Cross-Modal User Interfaces for Ubiquitous Environments. In: Proceedings of Engineering Interactive Systems, Salamanca (March 2007)
2. Berti, S., Correani, F., Paternò, F., Santoro, C.: The TERESA XML Language for the Description of Interactive Systems at Multiple Abstraction Levels. In: Proceedings Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages, pp. 103–110 (May 2004)

3. Berti, S., Paternò, F., Santoro, C.: Natural Development of Nomadic Interfaces Based on Conceptual Descriptions. In: Lieberman, H., Paternò, F., Wulf, V. (eds.) *End-User Development*. Human Computer Interaction Series, pp. 143–160. Springer, Heidelberg (2006)
4. Cypher, A.: *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge (1993)
5. Jess. The Rule Engine for the Java<sup>TM</sup> Platform, <http://herzberg.ca.sandia.gov/jess/>
6. Lieberman, H., Paternò, F., Wulf, V. (eds.): *End-User Development*. Human Computer Interaction Series. Springer, Heidelberg (2006)
7. Lieberman, H. (ed.): *Your Wish is my Command. Programming By Example*. Morgan Kaufmann Publishers, Academic Press, USA (2001)
8. Macías, J.A., Puerta, A., Castells, P.: Model-Based User Interface Reengineering. In: Lorés, J., Navarro, R. (eds.) *HCI Related Papers of Interacción 2004*, pp. 155–162. Springer, Heidelberg (2006)
9. Macías, J.A., Castells, P.: Finding Interaction Patterns in Dynamic Web Page Authoring. In: Bastide, R., Palanque, P., Roth, J. (eds.) *DSV-IS 2004 and EHCI 2004*. LNCS, vol. 3425, pp. 164–178. Springer, Heidelberg (2005)
10. Macías, J.A., Castells, P.: An EUD Approach for Making MBUI Practical. In: *Proceedings of the First International Workshop on Making model-based user interface design practical CADUI*, Funchal, Madeira, Portugal, January 13 (2004)
11. Myers, B.A.: *Creating User Interfaces by Demonstration*. Academic Press, San Diego (1998)
12. Paternò, F.: *Model-Based Design and Evaluation of Interactive Applications*. Springer, Heidelberg (1999)
13. Repenning, A., Ioannidou, A.: What Makes End-User Development tick? 13 Design Guidelines. In: Lieberman, H., Paternò, F., Wulf, V. (eds.) *End-User Development*. Human Computer Interaction Series, pp. 51–85. Springer, Heidelberg (2006)
14. Rode, J., Rosson, M.B., Pérez, M.A.: End-User Development of Web Applications. In: Lieberman, H., Paternò, F., Wulf, V. (eds.) *End-User Development*. Human Computer Interaction Series. Springer, Heidelberg (2006)
15. Rode, J., Rosson, M.B.: Programming at Runtime: Requiriments & Paradigms for nonprogrammer Web Application Development. In: *IEEE 2003 Symposium on Human-Centric computing Languages and Environments*, New York, pp. 23–30 (2003)

# Improving Modularity of Interactive Software with the MDPC Architecture

Stéphane Conversy<sup>1,2</sup>, Eric Barboni<sup>2</sup>, David Navarre<sup>2</sup>, and Philippe Palanque<sup>2</sup>

<sup>1</sup> ENAC – Ecole Nationale de l’Aviation Civile  
7, avenue Edouard Belin, 31055 Toulouse, France  
stephane.conversy@enac.fr

<sup>2</sup> LIIHS – IRIT, Université Paul Sabatier  
118 route de Narbonne, 31062 Toulouse Cedex 4, France  
{barboni, conversy, navarre, palanque}@irit.fr  
<http://liihs.irit.fr/{barboni,navarre,palanque}>

**Abstract.** The “Model - Display view - Picking view - Controller” model is a refinement of the MVC architecture. It introduces the “Picking View” component, which offloads the need from the controller to analytically compute the picked element. We describe how using the MPDC architecture leads to benefits in terms of modularity and descriptive ability when implementing interactive components. We report on the use of the MDPC architecture in a real application: we effectively measured gains in controller code, which is simpler and more focused.

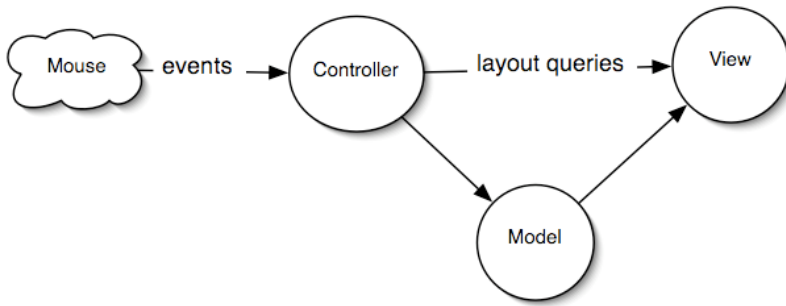
**Keywords:** MVC, interactive software, modularity, Model Driven Architecture.

## 1 Introduction

Modularity is an aspect of software engineering that helps improve quality and safety of software: once designed, implemented, and verified, modular components can be reused in multiple software so that such software can rely on their soundness. The advent of rich interaction on the web, and the advent of WIMP interaction in airplane cockpits [1][2] raise interest in interactive software architecture. The need to use, develop, and extend toolkits for interaction makes programmers eager to study this area. Similarly, a number of widgets have been formally described, so as to comply with important properties of interactive systems [14]. As a toolkit programmer point of view, reusing these components would ensure that his particular implementation complies with the same properties.

*Separation of concerns* is a design principle that can help to achieve modularity: the idea is to break a problem into separate sub-problems and design software components that would handle each sub-problem. The Model-View-Controller (MVC) architecture is a well-known attempt to improve modularity of software [5] through separation of concerns (cf Fig. 1). In MVC, the Model encapsulates the data to be interacted with, the View implements the graphical representation and is updated when the Model changes, and the Controller translates actions from the user to operations on the Model. MVC has been successfully applied to high-level

interactive components, though in this form it resembles more to the PAC architecture than its original description [6]. For example, frameworks to help develop interactive application, such as Microsoft MFC, organize the data structure in a document, and views on the document that are updated when the document changes. When applied to very low-level interactive components though, such as scrollbars, programmers encounter difficulties to clearly modularize the components so that the original goal of reusing components is reached: the View and the Controller components of the widget are so tightly coupled that it seems useless and a waste of time to separate them, as they cannot be reused for other interactive widgets<sup>1</sup>.



**Fig. 1.** MVC: The controller queries the view to know which part of the view has been clicked in order to react accordingly

We argue in this paper that by externalizing the *picking concern* from the Controller, we can actually modularize a set of interactive widgets so that the Controller can be reused across different classes of Views of the same model. We first present the causes of the problem mentioned above. We then introduce the Model – Display view – Picking view – Controller (MDPC) architecture, and show with examples how to use it. We then report our experience at refactoring a real application with the MDPC model.

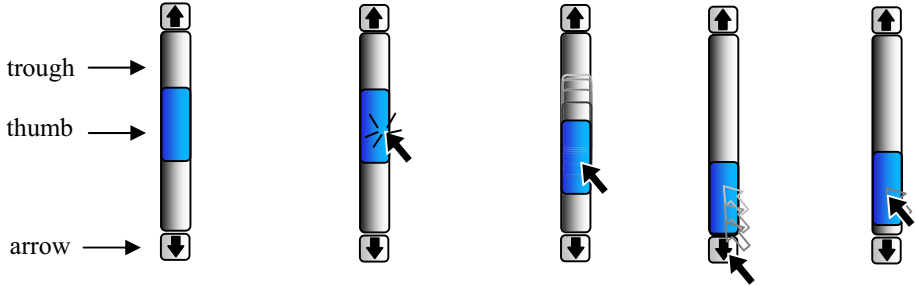
## 2 The Need to Externalize Picking

At its lowest level, today's interactions usually involve a rasterized image (i.e. a digital/sampled/pixel-based image) and a pointer that the user controls to point at a given pixel. *Rendering* is the process of transforming a logical description or the *conceptual* model of an interactive component to a graphical representation or a *perceptual* model. *Picking* can be considered as the inverse process of rendering: *Picking* is the process of determining/querying the graphical primitives that

<sup>1</sup> As stated by the designers of JAVA Swing: “We quickly discovered that this split didn't work well in practical terms because the view and controller parts of a component required a tight coupling (for example, it was very difficult to write a generic controller that didn't know specifics about the view). So we collapsed these two entities into a single UI (user-interface) object [...]”. <http://java.sun.com/products/jfc/tsc/articles/architecture/#roots>



colored/filled a given pixel, and in turn the corresponding conceptual entity. Usually, interactive systems use the pixel underlying the cursor, in order to react when the user clicks on an interactive component. Picking is also used during passive movements, for example to determine when the cursor enters an interactive component so as to highlight it.



**Fig. 2.** A scrollbar and its parts

For the remaining of this section, we take the scrollbar as an example (Fig. 2). A scrollbar is an instrument that enables a user to specify the location of a range by direct manipulation. For example, the user can use a scrollbar to modify the position of the view of a document too large to be displayed at once. Conceptually, a scrollbar is composed of four parts: a *thumb* to control the position of a range of values, a *trough* in which the user can drag the thumb, i.e. the position of the thumb is constrained inside the trough, and two *arrows* for decrementing/incrementing the position of the thumb by a fixed amount.

```

if( (event.y > y_up_widget) and (event.y <
    y_bottom_widget) { // test if it is in the widget
    if (event.y < y_up_widget+harrow) {
        // scroll down by one line
        ...
    } else if (event.y < ythumb) {
        // scroll down by one viewing area
    } else //...and so on

```

**Fig. 3.** An example of code using analytic picking

In the original form of MVC, the Controller usually handles picking by receiving low-level events such as mouse clicks or mouse moves. For example, if the user clicks in the image of a scrollbar for a text editor document, the Controller computes which part of the view has been clicked on, and calls a particular method of the Model with a computed parameter: if the part is one of the arrows, the Controller sets the Model's value by decreasing or increasing it by an amount equivalent to that of one line. If the part is the space between the thumb and the arrows, the amount is equivalent to that of one viewing area. In order to determine the part that has been clicked on, the Controller must know the layout of the widget parts, *i.e.* the location of

parts that are displayed on the screen [15]. For example, with a vertical scrollbar, if the upper ordinate of the widget is  $y_{widget}$ , the height of an arrow is  $h_{arrow}$ , and the upper ordinate of the thumb is  $y_{thumb}$ , a Controller can determine which part has been clicked on by using the code in Fig. 3.

The code is embedded into the method that reacts to the click on the view. This prevents modularization of the controller: it is specially designed for one particular view, even if some of the values can be parameterized, such as the location of the whole widget. In particular, the relative layout of the different parts of the widget is often hard-coded, and is not a parameter of the widget.

In fact, most interactive widgets are structured around parts that embody a spatial mode of interaction i.e. a same event in two different parts lead to two different behaviors of the widget. For example, clicking in an arrow triggers a different action than the one corresponding to clicking in the thumb. In a part, the action triggered by an event is the same regardless of the parameters of the event. Only the parameters of the action may depend on the dimensions of the event. What is important then to implement part-dependant code, is not the low level parameters of events such as the x and y coordinates, but the part on which the event took place. Thus, the Controller behavior must be dependant on parts below the cursor, and not the cursor's x and y position, so that the code that describes it would resemble to code in Fig. 4.

```

if( isin(event, scrollbar)) { // test if it is in the widget
  if (isin(event,uparrow)) {
    // scroll down by one line
    ...
  } else if (isin(event,thumb)) {
    // scroll down by one viewing area
    ...
  } else { //...and so on
    ...
  }
}

```

**Fig. 4.** An example of controller code independent of the exact position of parts

In this case, the "isin" function is a call to an external picking function. As such it is a mean to factor out the picking process from the Controller, and enables its reuse with other Views. However, implementing the controller with multiple *if/then/else* prevents extension and combination, as adding a part requires adding code to handle it. Instead, we propose to completely externalize the picking process, and make the Controller behavior dependant on *Leave/Enter* events, instead of *Move* events.

Usually, programmers describe graphics by the mean of graphical shapes: instead of filling pixels by themselves, they use a higher level of description, for example a circle at a given position with a given radius. A graphical library in turn fills the pixels according to the description. A *Leave* event is triggered when the shape under the cursor changes between two consecutive *Move* events. A *Leave* event is immediately followed by an *Enter* event, as leaving a shape means that the cursor enters another shape (we consider the background as a shape with infinite size, which lies under every other shape). *Leave* and *Enter* events are synthesized events: they are computed from *Move* events, and a description of the layout and contours of the

shapes in used. Thus, *Leave/Enter* events generation requires a data structure that keeps track of the layout of the shapes and their contours. This kind of data structure is called a scene-graph. Usually, a scene-graph is used as an intermediate stage in the rendering process described above: the programmer describes the rendering of the conceptual model in terms of shapes, their geometrical and styling transformation, that are stored in a scene-graph. Since a scene-graph knows about the layout and contours of shapes, it is able to determine the shape that is under the cursor. Thus a scene-graph can handle input and implement a picking service, as well as synthesize *Leave/Enter* events.

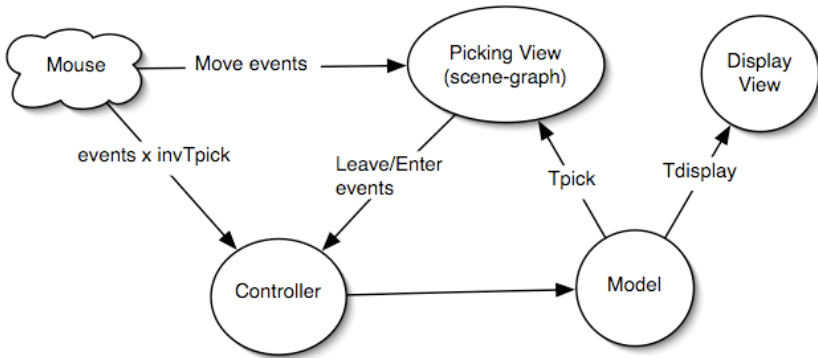


Fig. 5. The Model – Picking View – Display view Controller (MDPC) architecture

## 2.1 Display View and Picking View

We propose to split the View component into two components: the Display View, which is exactly the ancient MVC View, and the Picking View, which is an invisible rendering of the model that is specialized to facilitate interaction description. By splitting them, we deepen the separation of concerns aspect of the MVC model: while the display view handles the representation that has to be perceived by the user, the Picking View helps the determination of the part of the widget that is under the cursor. This separation also solves two problems of the design of interactive widgets, related to the differences between the structure of the graphics for interaction and the structure of the graphics for display: those due to graphic design concerns, and those due to transient, invisible interactive structure.

When developing widgets, a programmer can use graphical primitives that do not fit with interaction needs. For example a scrollbar can be seen as a thumb moving into a trough (Fig. 2). This can be described as two shapes, the thumb shape lying on top of the trough shape. If this structure were mapped to a scene graph, the Enter and Leave event would contain the identifier of the shapes, regardless of the position of the cursor relative to the thumb. Thus, the Controller would receive the same Move event, be it above the thumb, or below the thumb, and would not be able to discriminate the zone in which the cursor has actually entered (above or below the thumb), though this information is mandatory to implement control. This fact usually leads the programmer to implement analytic code, i.e. code that uses the x and y position of the thumb to eventually determine the zone. However, if the design of the

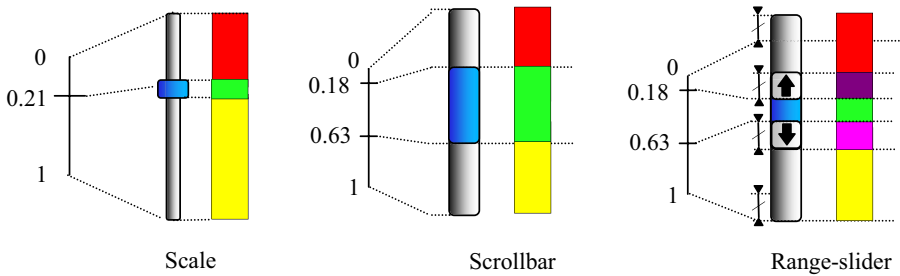
view is done with three parts, the "decrease part", the thumb, and the "increase part", the only information required to implement the interaction is the part identifier dimension of the Enter/Leave events. It is therefore necessary to decouple the display part of a widget from its picking part. Furthermore, interactive projects now involve graphic designers, whose creativity may be refrained by coding requirements. The separation between display and picking frees the graphic designer from the obligation to respect a graphical structure that does not map with the desired display: would the display view serves for both display and picking, the designer is required to use a three parts graphic, although two parts would have been enough. On the other hand, a designer can use as many graphical primitives she needs (like soft shadows, filters etc.), and in any configuration. In particular, she could have used sub-shapes like text of other images useful for the user to understand the display, but that are of no interest for interaction. As unnecessary graphical structures increases the complexity of formal checking of the controller code, reducing their number is important.

We saw above that the picking structure can be different from the display structure. But it can also change for the sake of interaction, while the display structure remains the same. In the scrollbar example, when one clicks on the thumb to move it along the trough, there are invisible zones in which spatial mode of interaction enters in action (Fig. 2, right). When the thumb "hits" the top or the bottom of the trough, the thumb does not move even if the user goes on with his movement, as the thumb is constrained in the borders of the widgets. However, when the user reverses his movement, there is a position from which the thumb starts moving again. This position is invisible, but can be computed as soon as the user clicks in the thumb: in the case of the vertical scrollbar, it is equal to the position of the widget plus the difference between the click and the top of the thumb. When the cursor is in this zone, the thumb moves as the cursor moves. When the cursor leaves this zone, and enters one of the two other zones, the thumb position is not updated anymore (and is set to 0 or 1). Usually, the interaction is described by using a "special mode" of the controller: as soon as the user clicks on the thumb, the controller "captures" the cursor so that moving it on top of unrelated display areas will not trigger associated actions. This behavior is traditionally implemented with analytic determination of distance from important points, such as the one described above. Instead, we propose to implement it using the same mechanism outlined above, namely with zones that are entered and left, with the difference that this time they are invisible and transient, as they are enabled only in certain states of the widgets. Hence, for one model, there can be one displayed view, whatever the interaction handled by the widget, and two different invisible, transient views to implement control, which leads to the split between Display Views and Picking Views.

### 3 Example 1: The Scrollbar in Depth

In this section we show how to use the MDPC model to describe the scrollbar. The model of the scrollbar is a range whose two boundaries lie in the range from 0 to 1 (Fig. 6). To specify values in an arbitrary range of values, not only 0 to 1, we can use a linear (i.e.  $ax+b$ ) transform function when notifying observers. The Scrollbar widget enables a user to specify position of the range, i.e. she can slide the range so that both

boundaries are changed at the same time. The extent of the range (i.e. the difference between the boundaries) is specified by either the application, or is tied to another widget such as a text widget. The range-slider is a scrollbar widget, augmented with instruments that enable the user to specify the values of the boundaries. Hence, the Scrollbar and the Range-slider share the same model.



**Fig. 6.** From left to right, the Model, the Display View, and the Picking View of the Scale, the Scrollbar, and the Range-slider. The model of the Scrollbar and the Range-slider is the same.

The display view is a drawing composed of several graphical shapes. In its simplest usable form, the drawing may resemble to Fig. 2: one background shape for the trough, and one shape for the thumb, lying over the background shape. The size of the trough is arbitrarily chosen. The size of the thumb can be computed from the values of the model and the size of the trough, using a simple linear function. However, the thumb has a minimum size to allow the user to pick it regardless of the extent it is supposed to reflect. As explained above, the structure of the display view cannot be used to implement the control, as it is necessary to differentiate between the part of the trough that is above the thumb from the part that is below. Hence, the picking view is composed of three shapes, one for the thumb, and two for the visible parts of the trough. When the user manipulates the thumb, the position of the thumb shape is changed in the display view and in the picking view, while the size of the two sub-shapes of the trough are changed in the picking view. The controller of the scrollbar can then be described with events that contain the identifier of the shapes, as there is no need to analytically compute which part has been clicked on.

### 3.1 Invariance to Geometrical Transform and Relative Layout Transform

The horizontal scrollbar is a  $90^\circ$  rotated vertical scrollbar. As such, it can be implemented by adding a  $90^\circ$  rotation in the display view and the picking view components. The interaction corresponding to a click in the arrows, and in the two parts of the trough, is exactly the same. However, in traditional MVC, the controller code of the vertical scrollbar has to be updated to handle the new positions of the parts. The controller as we defined it, does not need to be changed for a vertical scrollbar: it is invariant with respect to geometric transforms. This result is true for one type of interaction with the scrollbar, namely clicks in part that triggers action. With the  $90^\circ$  rotation example, the vertical movement corresponding to the manipulation of the trough is not compatible with the orientation of the scrollbar. To

overcome this problem, we use the inverse transform that enables the generation of the view, by transforming the events so that their coordinates are relative to the view, and not absolute (or relative to the screen). Using the inverse transforms, the controller remains the same.

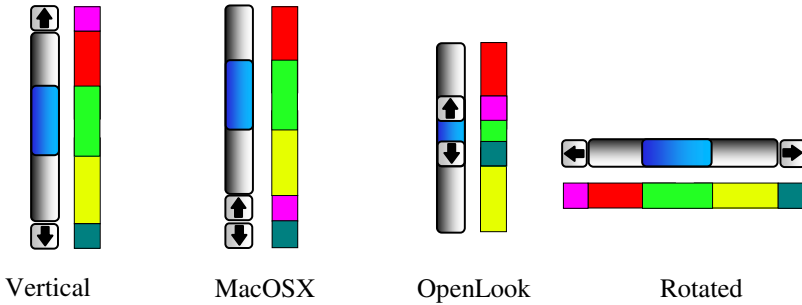
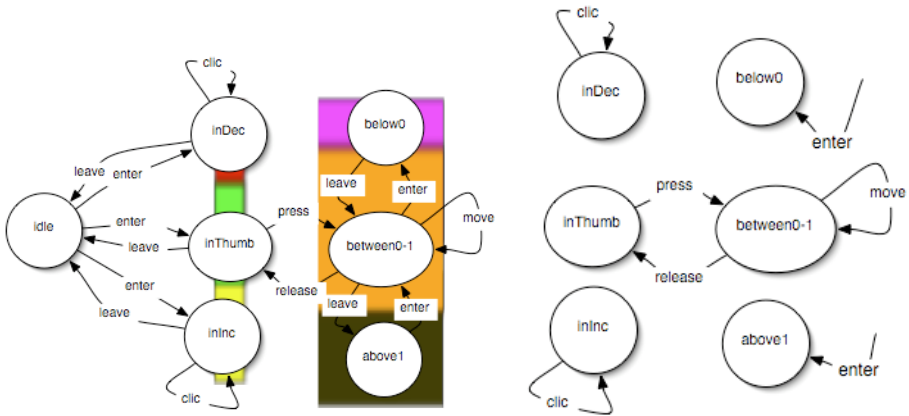


Fig. 7. The Display View, and the Picking View of varieties of Scrollbar

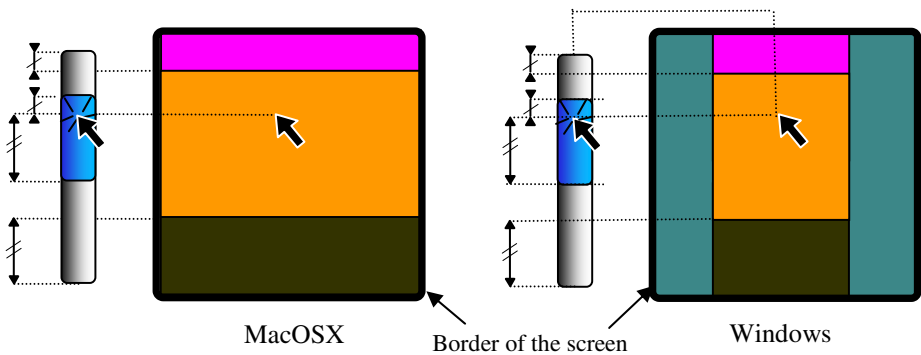
Moreover, the controller is invariant with respect to the relative layout of parts of the scrollbar. As shown on Fig. 7, the arrows can be move at one extremity of the trough (to mimic a variety of MacOSX scrollbar), or even at the ends of the thumb (to mimic OpenLook scrollbar). The same MDPC controller as the vertical scrollbar can control these kinds of scrollbar, whereas with MVC each variant requires a different controller.

### 3.2 Multiple Picking Views for Transient Behavior

When sliding the trough though, the user can go outside the widget and still hold control of the scrollbar. This has been handled in traditional architecture with a special mode of interaction, namely by “capturing” the cursor so that any other underlying system such as MVC is bypassed. With our model, moving outside the widget will trigger a Leave event, and eventually stops the controller. This behavior is due to the fact that dragging the thumb is actually a completely different interaction than clicking in scrollbar parts. In fact, the picking model is different from the one described above. The sliding interaction is dependent on three zones: one in which moving the cursor moves the thumb (or more precisely, set the boundaries of the scrollbar model, which is reflected by the view as a displacement of the thumb), and two in which movement has no effect on the model (and hence on the view of the thumb) because the thumb hit one of the edges of the trough. This can be implemented as another picking view (see Fig. 9, left). When clicking on the thumb, the “waiting-for-click” picking view of Fig. 8 is replaced by the “sliding” picking view. When the cursor moves inside the central part, the thumb follows its position. When the cursor enters the upper part, the value of the Model is set to 0, and does not move until it reenters the central area again. As long as the user holds the button pressed, the controller receives Leave, Enter and Move events and reacts accordingly. When the user releases the cursor, the “waiting-for-click” picking view comes back.



**Fig. 8.** To the left, the state-machine describing the behavior of the scrollbar. To the right, a simplified version with the transitions with associated actions only.

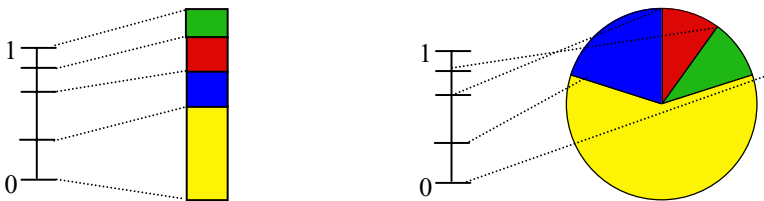


**Fig. 9.** When clicking on the thumb, a new Picking View is used. The thick rounded rectangle reflects the border of the screen.

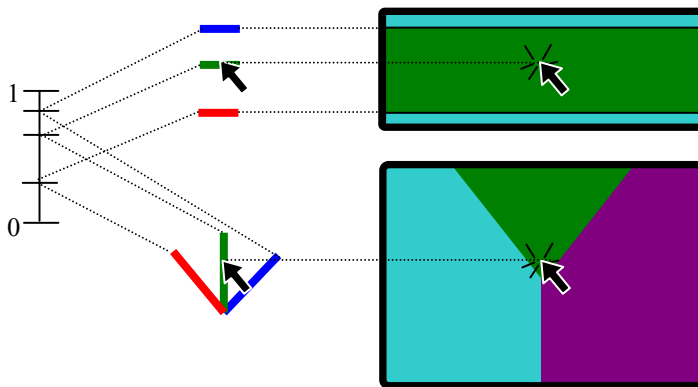
To assess the universality of this model, we can describe a variation of this interaction. While sliding the thumb, the user can move the cursor at a particular distance from the scrollbar. With a MacOSX scrollbar, this distance is infinite, and can be described with rectangular zones that extend horizontally up to the border of the screen. With a Windows scrollbar, the distance is finite, and when crossed, the thumb goes back to the position it has at the beginning of the interaction (i.e. when the user clicks on the thumb), enabling the user to cancel the interaction. This can be described by shrinking the three zones of the picking view (SEQ, right), so that the background appears at their sides: when the cursor enters the background zone, the Controller resets the thumb position back to its previous value.

### 4 Example 2: The Bar Chart and the Pie Chart

A partition model can be considered as a list of floats that range from 0 to 1. Each pair of floats specifies an interval. It can be represented with a bar chart, in which each part's height is proportional to its interval. It can also be represented with a pie chart, in which the extent angle of each part is proportional to its interval. Charts are often used as visualization only. However, a user can specify the values by clicking and dragging the borders between each part. Fig. 11 shows a picking view that enables this interaction. Thick borders reflect the interactive parts. They might be invisible in the display view, but are necessary to ease interaction. When clicking on a border, the second picking view enters in action, and precludes the user to move a value below or above neighbor values. It seems difficult to use the same controller for both Bar and Pie picking views since they are so different. However, they are topologically equivalent. We can use the inverse of the transform that generates the view: the Bar view involves a transformation from Cartesian coordinates, while the Chart view involves a transformation from polar coordinates.



**Fig. 10.** The Model of the partition and two Display Views: a Pie Chart, and a Bar Chart



**Fig. 11.** Above: the “wait-for-click” Picking View and “sliding” Picking View of the Bar Chart. Below: the “wait-for-click” Picking View and sliding Picking View of a Pie Chart. Both “sliding” picking view prevent the user to move a value below or above neighbor values.



## 5 Example 3: The Hierarchical Menu

When clicking on an entry of a hierarchical menu that has sub-entries, a pull-down menu shows up. On MacOSX, the controller allows the user to “fly over” entries of the first menu and reach entries of the submenu that are displayed at the bottom and left of the current location of the cursor. If the cursor goes downward vertically, it enters another entry, and the sub-menu hides itself. If the user does not initiate the interaction after a few milliseconds, the “fly over” mode is stopped. As shown in Fig. 12, it can be implemented with a transient triangular-like shape in the Picking View. Apart from the fact that the MDPC model eases the comprehension of the behavior, it leads again to less code in the controller, as no analytical computation is necessary to implement control. Moreover, it simplifies the architecture of the code, since no special mode of interaction in which the cursor is captured is necessary. It also shows that the picking structure can be very different from the display structure: it needs a transient state in which a shape helps implement interaction, but that is hidden to the developer. Finally, the set of necessary shapes for picking are less important than the set necessary for display (for each entry in the hierarchical menu: a sub-shape for the background, the text, the triangle icon). When using the same scene-graph for both display and picking, special code that prevents action for Leave/Enter events involving sub-shapes is needed. Separating the scene-graphs removes this obligation, and leads to smaller, more focused, code.

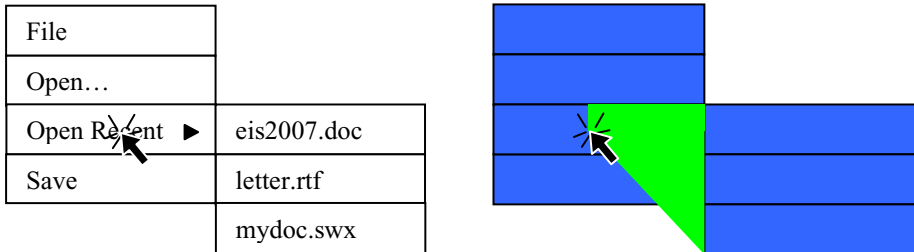


Fig. 12. The Display View and Picking View of a deployed hierarchical menu

## 6 Return of Experience with a Real Application

We updated the architecture of a real application that uses the ARINC 661 set of widgets [2]. The purpose of ARINC 661 specification [1] is to define interfaces to a Cockpit Display System used in interactive cockpits. MPIA is an airborne application that uses the ARINC 661 specification, and that aims at handling several flight parameters. It is made up of 3 pages (called WXR, GCAS and AIRCOND) between which crewmember are allowed to navigate using 3 buttons (as shown on Fig. 13). Interaction with MPIA relies on button-like widgets exclusively. Though we did not use the button as an example in previous sections, our observation that controller code is polluted by picking code holds true: with the previous architecture, picking is done by traversing the tree of widgets and by checking for each widget whether it is picked. We want to show with this example that externalizing the picking process leads to more simpler, more focused code.

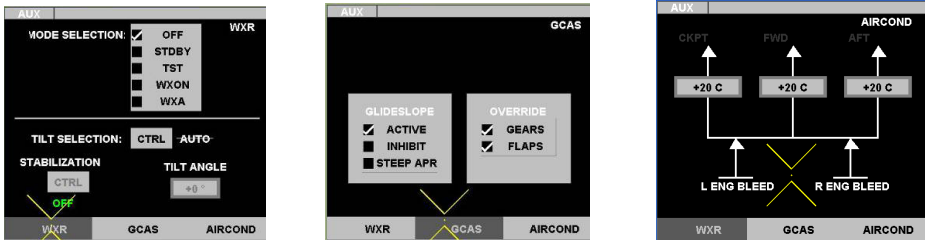


Fig. 13. The three pages of MPIA

Though we described control with state machines so far, for this application we used the ICO formalism [17], which is based on Petri-Nets. The next section describes how rendering is done using declarative specifications, and how the renderer implements picking services for the handling of low-level user input events.

## 6.1 Rendering

In the previous version, rendering was implemented with Java code, using the Java2D API. Instead, we now rely on an SVG description. SVG is an xml-based vector graphics format: it describes graphical primitives in terms of analytical shapes and transformations. As such, SVG is a scene-graph. To render SVG, we use the Batik renderer. Transforms from models to graphics are done with XSLT. XSLT is an xml-based format that describes how to transform an xml description (the source) to another xml description (the target). An XSLT description is called a “stylesheet”. XSLT is traditionally used in batch mode to transform a set of xml files, but XSLT can also be used in memory so that performances are compatible with interaction. We used the Apache Xalan library to handle XSLT transforms.

In our case, the source is a DOM description of the components the application: the “ARINC tree”. It is built at startup time, together with the instantiation of the ICOs components. Before running the application, the system compiles two stylesheets to two XSLT transformers: one for the display view, and one for the picking view (Fig. 14). This compilation can be triggered at any time, to update a stylesheet while designing and implementing it. While running the application, each time the state of an ARINC tree variable changes, the transformer transforms the ARINC tree to two DOM SVG trees, which in turn are passed to the SVG renderer (Fig. 16). The display view is then displayed in a window, while the picking view is rendered in an offscreen window.

Each time the cursor moves on the display view, the picking manager “picks” the topmost graphical item *of the picking view* at the position of the cursor, as if the cursor was moving over the picking view instead of the display view. Then, the picking manager sends an event to the Petri nets with the cursor position and the ID of the graphical item under the cursor as parameters. The Petri nets specification then uses the ID to retrieve the instance of the models over which the cursor is.

```

arinc xml description:
<arinc>
  <button x="10" y="10" width="200" height="50" text="submit"
enable="1"/>
</arinc>

xslt stylesheet:
<xsl:stylesheet>
  <xsl:template name="button">
    <rect x="{@x}" y="{-(@y+@height)}" width="{@width}"
height="{@height}" rx="150" fill="url(#gradientPanelBackground)"/>
    <text x="{@x}" y="{-@y}">submit</text>
  </xsl:stylesheet>

generated svg:
<rect x="100" y="-60" width="200" height="50" rx="150"
fill="url(#gradientPanelBackground)"/>
<text x="100" y="-50">submit</text>

```

**Fig. 14.** Examples of an ARINC tree, an XSLT transformer, and the resulting SVG Picking

## 6.2 Advantages of the Architecture

Our goal with this application is to show that it is possible to externalize picking from the controller. The resulting refactoring first shows that the architecture is implementable, and that it enabled us to reduce the complexity of the controller code by a significant amount (about 25% less), without removing functionality. While applying it to the entire modeling of the MPIA application and the user interface server compliant with ARINC 661 specification this produced a significant reduction of model size as shown in Fig. 15. This difference is more salient with widgets in charge of the assembly of widgets as the ones shown Fig. 15. For other terminal widgets (like command buttons, text boxes), the reduction of models size is still present but more limited.

Widget	Model size without MDPC		Model size with MDPC	
	Places	Transitions	Places	Transitions
RadioBox	49	29	28	21
TabbedPanelGroup	62	22	44	16
TabbedPanel	72	49	22	7
Panel	65	46	16	5

**Fig. 15.** Measure of volume of each widget in terms of model size

This architecture clearly distinguishes the conceptual model from the perceptual model, and gathers all the graphics and transforms description into one external entity. It has clear advantages over the previous architecture. First, it increases readability of the graphical code. Second, changing the look of an application is as simple as changing the XSLT file. Fig. 17 shows two renderings that can be alternatively presented without making any change in the models describing the

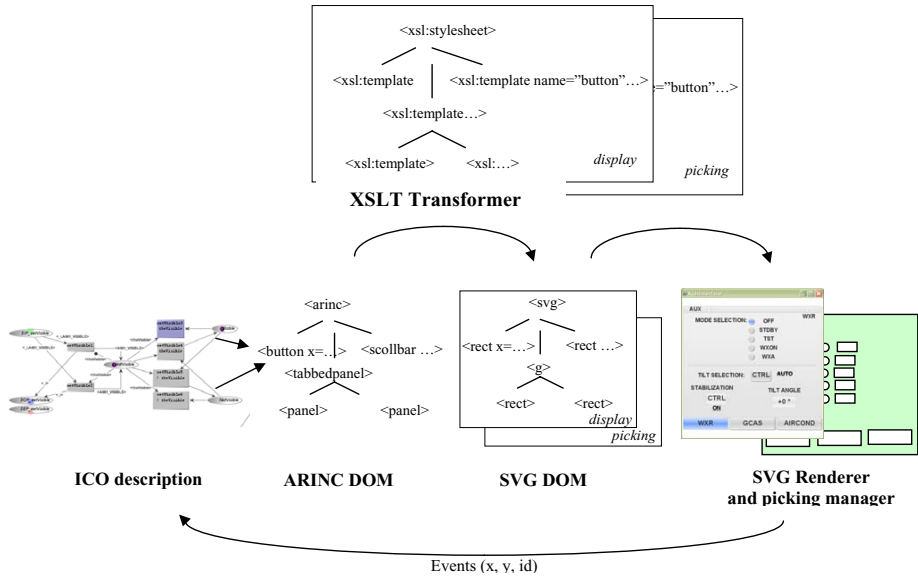


Fig. 16. The run-time architecture

widgets. Most commercial drawing and painting software can produce SVG graphics compatible with our system, allowing graphic artists to be involved earlier in the design process of interactive applications [5]. Finally, rendering is considered as a transformation process that uses functional programming without side-effect, which increases robustness [9]. It is also interesting to note that the advantages of the architecture are demonstrated both at the level of programming code and at the level of model description.

### 6.3 Drawbacks of the New Architecture

The performance of dedicated Java2D code is much better than the one exploiting SVG, XSLT, and Batik. The low performance of the new architecture comes from the fact that the transformation process involves the entire conceptual model each time it

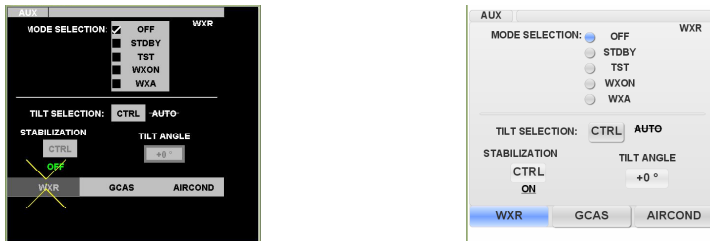


Fig. 17. The same User Application window with two different stylesheets

is triggered, leading to a whole new SVG DOM, even if a single variable of the ARINC DOM has changed. This problem is related to the current status of transforming tools, which are unable to do incremental transformations. An incremental transformer is able to only update the changing parts of the target tree, which increases performances (up to 500 times) [16][22]. Another solution is to use “active transformations”, i.e. transformation systems and specifications designed to implement incremental transformations [3].

## 7 Related Work

Fabrik is a direct manipulation - based user interface builder that enables a designer to specify transforms between widget with a visual flow language [10]. Events flow in the same flow graph that describes the geometrical transforms, so that they are automatically transformed to a position relative to the graphically transformed widget.

In [7], Dragicevic and Fekete introduce the MVzmC architecture. Like MDPC, the widget is divided into zones that embody a spatial mode of interaction. The “view controller” plays the role of our transform mechanism, and works similarly to Fabrik transforms. However, the Vzm component is still in charge of determining which zone has been hit, hence it is not invariant to changes of relative layout of parts. Similarly, in the Event-driven MVC [20], the code that handles picking is buried into the view, and hence precludes any simple change of layout. MDPC clearly factors out this task from the Controller and the View, which leads to more reusable code. Finally, both the MVzmC and Event-driven MVC use a single view, and cannot be used to implement transient picking structure.

Using a declarative description of an interface is not new (see [21] for a review). However, in much of these systems, the description is only a way to get the interface outside the code of the application: a run-time environment displays widgets by interpreting the description. Furthermore, the description is only about the layout of predefined WIMP widgets at the finer level of details. Such systems are primarily targeted at toolkit users (i.e. interactive application designers) that do not need to implement new or slightly different interaction techniques. In our case, the architecture describes all models, from the level of the application down to the inner mechanics of a widget. For example, we can describe the control and the rendering of a range slider using the same architecture that we use to describe the application, while it’s not possible with other systems.

The idea of transforming a conceptual model to a perceptual model comes from the Indigo project [4], a novel client-server architecture for highly interactive systems. While X11 splits rendering and interaction between the server and the client, Indigo makes the server in charge of the rendering *and* the interaction. To reflect changes of the logic of the client into the rendering, Indigo uses a transformation process similar to the one described in this paper. Indigo widgets are part of the server, and the rendering is not done using a transformation process. In our architecture, we apply the transformation model to the inner mechanics of the widgets. Transforms are also used in [13], but it is done once at the instantiation of widgets from a layout description, while transforms are used continuously in our architecture.

## 8 Discussion

This work attempts to tackle the question often asked when disserting about the MVC model: what is a controller exactly? As inventors of MVC apply separation of concerns to interaction code, we can apply separation of concerns down to the Controller itself: in MVC, the controller handles *picking*, the backward *translation* of dimension of events to arguments for operations on the model, and the *management* of the interactive state of interactive components (as opposed to graphical state). In MDPC, the combination of the scene-graph (the picking view) and Leave/Enter events synthesis handles picking. The picking code is hence offloaded from the Controller code, which makes the controller simpler. In order to pass computed values from events to arguments for operations on the model, the old Controller has to transform dimensions of the events in the widget referential: hence, it is dependent on the View, as it must queried its parameters (such as the orientation) to compute the inverse transform. This backward translation from the dimensions of the events to arguments for operation on the model can be handled by the inverse transform mechanism in the MDPC model. We have shown how to do it functionally with rotated scrollbar and pie charts. If this translation is more complex, it can be dealt with with a similar mechanism to MVzMC's one, i.e. a View Controller. Hence, in the MDPC architecture, what we call a controller is the piece of code that manages the interactive state of a component, i.e. the state-machine or the Petri Net that describes it. The interactive state is different from the graphical state. The graphic state is just a direct translation from the model to a graphical representation. For example, if a scrollbar is disabled because the interface does not allow the user to interact with it, there should be a Boolean in the model that should reflect it, and that would be used to draw a disabled toolbar (for example in gray). The management of interactive state is the core functionality of the Controller: it defines the behavior, or the inter-actions between the user and the model, i.e. the intertwined sequences of actions from the user, and actions from the system that change the set of future actions at user's disposal. With such a definition, Controllers presented in this paper seem simple. However, when dealing with multiple inputs, the description is complex, and may require Petri Nets with dozens of places and transitions. With the MPIA application, the code associated to transition is limited to change of values in the model, without any other computations. Hence the Controller is the Petri net, and almost nothing else, except the rules that change values in the model. In other words, it seems to us that it is impossible to remove other aspects of the Controller, as we reduced it to its crux.

Another goal of this project was to foster the use of an MDA approach to the design of interactive application. We designed the models of our widgets in order to make them as independent as possible from controllers and views, which led to the merge of the scrollbar model and the range slider model into a single range model. The choice of setting the bounds of the values inside the models to a range of  $[0,1]$  makes the model even more reusable, since the addition of a linear function makes it general enough to describe previous use of scrollbars. Our approach is an attempt to maximize the late binding aspect of our components: MDPC makes use of late binding of range bounds, of positions, and of relative position of parts.

## 9 Conclusion

In this paper, we have presented a new architecture for interactive systems implementation. We split the View component of MVC in a Picking View and Display View components. The picking task, traditionally handled by controllers of interactive widgets, is offloaded to a picking manager, which turns Move events to Leave/Enter events by using the picking view. Widgets following this architecture gain invariance from relative layout of components and invariance from geometrical transforms. The Controller code shrinks and is more focused to its functional core. The architecture can also be used to implement invisible, transient interactive structure. One of the goal of this project is to have a complete MDA driven widget set. The MDPC architecture is a first step towards this objective, as it enables the definition of interactive systems based on a MDA approach. The controller is specified using a formalism such as Petri Nets, the display and picking view are specified with a transformation model based on declarative specifications. In order to fully accomplish our goal, we need better and more efficient transform tools. In particular, we plan to design incremental, and bidirectional transform engine, in order to ease the definition of transforms. Another result is more conceptual: thinking of control as leaving/entering/moving over/clicking on possibly invisible parts helps design and describe it, as shown in the hierarchical menu example.

**Acknowledgments.** This work is partly funded by DPAC (Direction des Programmes de l'Aviation Civile) étude "validation cockpit interactif" and by EU via the Network of Excellence ResIST ([www.resist-noe.org](http://www.resist-noe.org)).

## References

1. ARINC 661 specification: Cockpit Display System Interfaces To User Systems, Prepared by Airlines Electronic Engineering Committee, Published by Aeronautical Radio (2002)
2. Barboni, E., Conversy, S., Navarre, D., Palanque, P.: Model-Based Engineering of Widgets, User Applications and Servers Compliant with ARINC 661 Specification. In: Doherty, G., Blandford, A. (eds.) DSVIS 2006. LNCS, vol. 4323, pp. 25–38. Springer, Heidelberg (2007)
3. Beaudoux, O.: XML Active Transformation (eXAcT): Transforming Documents within Interactive Systems. In: Proc. of the 2005 ACM Symposium on Document Engineering (DocEng 2005), pp. 146–148. ACM Press, New York (2005)
4. Blanch, R., Beaudouin-Lafon, M., Conversy, S., Jestin, Y., Baudel, T., Zhao, Y.P.: INDIGO: une architecture pour la conception d'applications graphiques interactives distribuées. In: Proceedings of IHM 2005, Toulouse, France, pp. 139–146 (September 2005)
5. Chatty, S., Sire, S., Vinot, J., Lecoanet, P., Lemort, A., Mertz, C.: Revisiting visual interface programming: creating GUI tools for designers and programmers. In: Proceedings of UIST 2004, pp. 267–276. ACM Press, New York (2004)
6. Coutaz, J.: PAC, an Object Oriented Model for Dialog Design. In: Proc. of Interact 1987, pp. 431–436. North Holland, Amsterdam (1987)
7. Dragicevic, P., Fekete, J.-D.: Étude d'une boîte à outils multi-dispositifs. In: Proc. of the 11th French speaking conf. on Human-Computer Interaction (IHM 1999), pp. 33–36 (1999)
8. Extensible Markup Language (XML) 1.0 (Third edn.) W3C Recommendation, <http://www.w3.org/TR/REC-xml/>
9. Hudak, P.: Conception, evolution, and application of functional programming languages. ACM Comput. Surv. 21(3), 359–411 (1989)

10. Ingalls, D., Wallace, S., Chow, Y., Ludolph, F., Doyle, K.: Fabrik: a visual programming environment. In: Proc. of OOPSLA, San Diego, California, United States, September 25 - 30, 1988, pp. 176–190. ACM Press, New York (1988)
11. Jacob, R.J.: A Visual Language for Non-WIMP User Interfaces. In: Proc. of Symposium on Visual Languages, VL, p. 231. IEEE Computer Society, Washington (1996)
12. Krasner, G.E., Pope, S.T.: A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *J. Object Oriented Program.* 1(3), 26–49 (1988)
13. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L.: UsiXML: A Language Supporting Multi-Path Development of User Interfaces. In: Bastide, R., Palanque, P., Roth, J. (eds.) DSV-IS 2004 and EHCI 2004. LNCS, vol. 3425, pp. 200–220. Springer, Heidelberg (2005)
14. David, N., Philippe, P., Rémi, B., Ousmane, S.: Structuring interactive systems specifications for executability and prototypability. In: Palanque, P., Paternó, F. (eds.) DSV-IS 2000. LNCS, vol. 1946, p. 97. Springer, Heidelberg (2001)
15. Olsen, D.R.: *Developing User Interfaces*. Morgan Kaufmann, San Francisco (1998)
16. Onizuka, M., Chan, F.Y., Michigami, R., Honishi, T.: Incremental maintenance for materialized XPath/XSLT views. In: Proc. of WWW 2005, pp. 671–681. ACM Press, New York (2005)
17. Palanque, P., Bastide, R.: Petri nets with objects for specification, design and validation of user-driven interfaces. In: Proc. of IFIP Interact 1990, Cambridge, UK, August 27-31 (1990)
18. Samet, H.: *Applications of Spatial Data Structures: Computer Graphics, Image Processing, GIS*. Addison-Wesley, Reading (1990)
19. Scalable Vector Graphics (SVG) 1.1 Specification,  
<http://www.w3.org/TR/SVG11/>
20. Shan, Y.: An event-driven model-view-controller framework for Smalltalk. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 1989, pp. 347–352. ACM Press, New York (1989)
21. Souchon, N., Vanderdonckt, J.: A Review of XML-Compliant User Interface Description Languages. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) DSV-IS 2003. LNCS, vol. 2844, pp. 377–391. Springer, Heidelberg (2003)
22. Villard, L., Layaida, N.: An incremental XSLT transformation processor for XML document manipulation. In: Proc. of WWW 2002, pp. 474–485. ACM Press, New York (2002)
23. XSL Transformations (XSLT) Version 1.0 W3C Recommendation,  
<http://www.w3.org/TR/xslt>

## Questions

### **Laurence Nigay:**

*Question: Can you combine picking views? I don't have an example in mind but it may be necessary in case of combined behaviour of widgets.*

Answer: I did not think of this issue. Such combination may be done at a high level of abstraction, not at the level of elementary widgets.

### **Yves Vandriessche:**

*Question: Do you use graphical acceleration hardware for the picking view. Getting the colours of a pixel on the hardware is pretty slow.*

Answer: The picking view does not have to be a bitmap, you can use a quadtree for example or other models.



# Toward Quality-Centered Design of Groupware Architectures

James Wu and T.C. Nicholas Graham

School of Computing, Queen's University, Kingston, Canada K7L 3N6  
JamesWu@lincsat.com, graham@cs.queensu.ca

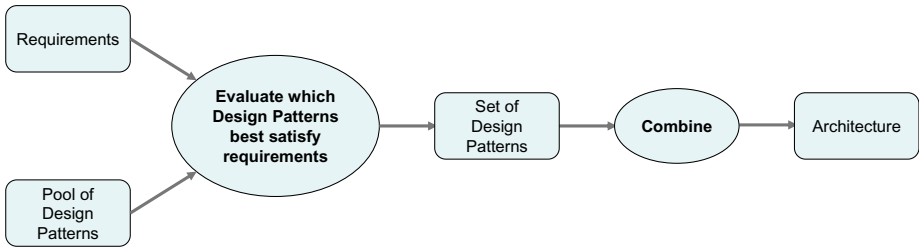
**Abstract.** Challenges in designing effective groupware include technical issues associated with concurrent and distributed work and social issues associated with supporting group activities. To address some of these problems, we have developed a *quality-centered* architectural design framework that links requirements analysis to architectural design decisions for groupware systems. The framework supports reasoned architectural design choices that are used to tailor software architecture to the unique quality and functional requirements of the software being developed. The framework has been applied to the development of the Software Design Board, a tool for collaborative software engineering.

## 1 Introduction

Groupware tools help people work and play together, providing integrated mechanisms for communication, collaboration and coordination [7]. Common examples of groupware include Lotus Notes' document repositories, the MSN Messenger instant messaging tool, the WebArrow/Conference online meeting tool, and the World of Warcraft massively multiplayer online game.

Groupware applications are difficult to construct, involving the difficult technological problems of supporting real-time interaction over a distributed system. A wide range of quality attributes affect the user's collaboration experience. Tools with poor *availability* may be unreliable and lead to inconvenience or loss of work. High *security* is required to ensure that the user's privacy is respected. Synchronous groupware requires high *performance* to support fluid interaction with other participants.

When translated into architectural choices, these requirements often conflict. For example, a requirement for high security might imply that all shared data should be stored at a single site, reducing the risk of unwanted data access. On the other hand, a requirement for high availability might imply that shared data should be replicated at multiple, redundant sites. Since there is no single groupware architecture that provides all of these qualities, architects of groupware systems must therefore carefully analyze their requirements to determine how to resolve these conflicts. *Architectural tradeoff analysis* involves the methodical comparison of architectural choices in order to determine what architecture best fits a system's requirements. Such analysis allows designers to reason about the properties of a system's implementation before it is developed, and as such is one of the fundamentals motivating architectural design.



**Fig. 1.** How an architect applies quality-centered architectural design

To perform such analysis, designers require a set of alternative architectures from which their system may be composed, and a reasoning framework allowing them to assess the properties of each architectural choice. Such architectural “tool boxes” have not been widely developed.

In this paper, we present a quality-centered design framework for the groupware application domain. The framework consists of a set of architectural design patterns that can be combined to create groupware architectures, and a set of analytical models for quality attributes of interest to groupware. Architects can select those design patterns whose qualities best match the requirements of their groupware system, and combine them into an architecture.

The groupware domain provides a rich field of study for architectural tradeoff analysis, as there are numerous solutions to each architectural problem with no clear means of choosing between them. To illustrate its utility, we have applied our framework to the design and implementation of the Software Design Board [13], a tool supporting collaborative design of software systems.

## 2 Quality-Centered Architectural Design

We aim to improve users’ experience with groupware applications through a novel quality-centered architectural design framework. The framework assists programmers in identifying candidate architectural styles for their groupware application, and in methodically determining which architecture best meets their requirements. Our contributions with the framework are:

- a set of *analytical models* that help relate software quality attributes to user experience,
- a set of *design patterns* that capture solutions to common problems in architecting groupware systems,
- a *quality impact matrix* that helps link the design patterns to desired system qualities.

Figure 1 shows how quality-centered architectural design links requirements analysis and architectural design, following the approach of Bass *et al.* [1]. Requirements are expressed in terms of key quality attributes such as performance, security, usability and availability. To help architects reason about design tradeoffs, our framework provides a *pool of architectural design patterns*, each of which embodies an architectural decision. In groupware, decisions might include

- whether to centralize or replicate shared data
- whether to use an optimistic or pessimistic concurrency control scheme
- how to reestablish service following the failure of a central communication hub
- how to distribute information required for awareness functions (such as telepointers).

The pool of design patterns includes different architectural solutions for these problems, representing different points in the space of tradeoffs. This provides architects with choices of how to best meet their application's requirements. The specification of a design pattern therefore includes analysis of its qualities, detailing the conditions where the pattern may improve (or worsen) the various quality attributes. For example, a pattern using an optimistic concurrency control scheme may improve feedback time while worsening the fidelity of different participants' views; a pattern involving data replication may improve the application's robustness to failure, while increasing its vulnerability to privacy violations.

The architect evaluates which design patterns best satisfy the application's requirements, and chooses a set of design patterns to be used in the architecture. These patterns must be combined to create an architecture for the system. This combination step may be straight-forward, but may involve further design work to enable the design patterns to work together. If combination of a set of patterns is not practical, new patterns may have to be chosen from the available pool.

In the following section, we examine a representative set of quality attributes, and develop analytical models which we will then use in section 4 to analyze our pool of groupware design patterns.

### 3 Qualities and Analytical Models

As seen in figure 1, architects select design patterns from a candidate pool based on their architectural qualities. Analytical models support this selection process, allowing the architect to evaluate design patterns with respect to a particular quality attribute. For example, *availability* is used to measure the frequency at which the system fails (and is unavailable for use); *security* measures how easily private data can be accessed by malicious third parties; *usability* measures how easily users can apply the system to performing their tasks; *functionality* measures how well the system matches the users' tasks; and *performance* measures how quickly the system responds to users' actions.

Analytical models serve as the basis for analyzing the qualities of design patterns. They provide a vocabulary for discussing quality attributes; for example, "performance" is computed from elements such as "local processing time", "network time" and "remote processing time", while "usability" of a groupware application comprises elements such as "fidelity", "consistency" and "awareness". Ultimately, analytical models allow us to determine the properties of architectural design patterns, supporting the choice of which design patterns best meet the requirements of a given application.

As representative examples, we now present analytical models for the availability, usability and performance quality attributes. These analytical models are developed specifically for the groupware domain. In section 4, we will show how these models allow us to precisely discuss the properties of design patterns.

In describing analytical models, we follow (but simplify) the approach of Bass *et al.* [1]. We specify an analytical model for each of a set of quality attributes as applied to the domain of collaborative applications. Analytical models are defined in terms of a set of *measures*, observable phenomena that influence the attribute of interest. For each analytical model, we then discuss what stimuli influence the measures, and give examples.

### 3.1 Analytical Model: Availability

Availability measures robustness of a groupware system in terms of what percentage of the time that the system is available for use. Poor availability leads to a negative user experience, as failures may lead to lost work or frustrating interruptions in collaborative sessions.

Analytical Model: Availability Domain: Collaborative applications Measures: Mean Time to Failure, Mean Time to Repair Details:

$$\text{availability} = \frac{\text{Mean Time to Failure}}{\text{Mean Time to Failure} + \text{Mean Time to Repair}}$$

Where: *Mean Time to Failure* is the average length of time between component failures, and *Mean Time to Repair* is the average length of time required to restore the functionality of a failed component.

Discussion: In this context, Mean Time to Failure is influenced by both network and software component reliability. Any architectural feature that can improve the reliability of these components will increase the Mean Time to Failure experienced by individual collaborators. Architectural features that allow a component to remain functional in the presence of faults will increase the Mean Time to Failure. Similarly, features that influence the ability to reconfigure or repair the system when failures have occurred will affect Mean Time to Repair.

Examples:

- 1 Localizing the effects of any component failure can reduce Mean Time to Failure. For example, if a failure in a document sharing system can be localized, reducing the number of users who are unable to interact with the document, then the overall availability of the document to the group is increased.
- 2 Mean Time to Repair can be reduced by using redundant copies of core components to re-establish functionality in the event of a failure. This eliminates the processing associated with recovering the failed component, allowing functionality to simply be resumed by the back-up component.

### 3.2 Analytical Model: Usability

Using synchronous groupware should come as close as possible to the experience of collaborating in the same location. Usability measures aspects of how closely the groupware system achieves this goal.

Analytical Model: Usability

Domain: Collaborative applications

Measures: Fidelity, Consistency, Awareness

Details: *Fidelity* measures the degree to which a participant's view of shared artifacts represents their actual state. *Consistency* measures the degree to which different collaboration channels are synchronized. *Awareness* measures to what degree a participant can perceive the actions and attention of other participants.

Discussion: A primary source of reduced *Fidelity* is the time that it takes for one participant's actions to be transmitted to other participants over a network. When participants are working asynchronously, their views of the system may become considerably out of date. Some algorithms for presenting participants consistent views of a shared state involve rollbacks of committed actions; in this case, *Fidelity* is compromised because the participant has been shown a view that is incorrect.

Groupware applications often allow people to collaborate using a variety of channels, such as voice, video, view of a shared artifact, and telepointers. *Consistency* measures how well these channels are synchronized. Poor consistency can lead to confusion, for example, a presenter talking over a slide that has not yet appeared on an audience member's display.

Groupware participants need to understand the activities and intentions of their collaborators. Such *awareness* may be improved via simple mechanisms such as telepointers, or advanced mechanisms such as gaze awareness.

Examples:

1. The use of an optimistic concurrency control algorithm allows a participant's actions to be reflected immediately in their view of a system. However, if this action conflicts with that of another participant, it may be rolled back. If conflicts are rare, the use of this optimistic concurrency control improves *Fidelity* by reducing feedback time; if conflicts are frequent, *Fidelity* is compromised due to high numbers of roll-backs.
2. Timestamping and buffering can be used to synchronize the data from different collaboration channels. This approach can improve *Consistency*, but at the cost of reducing *Fidelity* through increased latency.

### 3.3 Analytical Model: Performance

Performance affects the fluidity and naturalness of collaboration. If users find the tool to be unresponsive to their own actions or slow to report the actions of others, their experience of working together in a group will be negatively impacted.

Analytical Model: Performance Domain: Collaborative Applications Measures: Feedback Time, Feedthrough Time Details:

$$\begin{aligned} \text{Feedback Time} &= \text{Local Processing Time}_{FB} + \text{Network Time}_{FB} \\ &\quad + \text{Remote Processing Time}_{FB} \end{aligned} \quad \text{Feedthrough Time} = \text{Local}$$

$$\begin{aligned} \text{Processing Time}_{FT} &+ \text{Network Time}_{FT} \\ &\quad + \text{Remote Processing Time}_{FT} \end{aligned}$$

Where: *Local Processing Time* is the time taken to process events at the initiating user's local machine; *Network Time* is the time taken to transmit events across the network to remote machines, and *Remote Processing Time* is the time taken to process events received from the network at a remote machine.

Discussion: *Feedback Time* represents the time from a user performing an action to seeing the result of that action. *Feedthrough Time* represents the time from a user performing an action to *other users'* seeing the result of that action. These measures are influenced by two factors – the performance of the network connecting collaborators (i.e., with respect to bandwidth and/or latency) and the amount of time required to process events before results can be displayed to users.

Examples:

1. Feedback Time can be reduced by eliminating the need for an event to be sent over the network before updating the display of its initiating user. That is, if the user's display can be updated without any network interchange, both the Network and Remote Processing times are removed from the above equation; i.e., (Feedback Time = Local Processing Time<sub>FB</sub>.)
2. For Feedthrough Time, network traffic cannot be avoided; reducing the bandwidth required by events being sent across the network maximizes available bandwidth, thereby reducing the Network portion of the equation and therefore the overall time required.

The *performance* analytical model demonstrates how analytical models are developed for a particular application domain. The primary performance issues for groupware have to do with how quickly users see the results of their own actions (Feedback Time) and how quickly they see the results of others' actions (Feedthrough Time.) There are many other ways that performance of distributed systems can be measured (e.g., turnaround time, throughput, CPU load), but for groupware, Feedback and Feedthrough Time are the most important. By being able to concentrate on the measures that are most important for a particular domain, we can greatly reduce the complexity of analytical models.

## 4 Design Patterns

Following the approach of figure 1, developers of groupware applications first identify quality requirements, expressed in terms of the quality attributes discussed in section 3. The developer then selects from a pool of design patterns that best meet these requirements. The selected patterns are subsequently combined into a concrete architecture.

In order to architect groupware applications, we have identified a set of 21 design patterns supporting a range of groupware applications, involving real-time and asynchronous collaboration between co-located and remote collaborators. In addition to a description of how it is used, each design pattern is accompanied by an analysis summary. This summary explains the pattern's properties with respect to quality attributes, and is expressed relative to the relevant analytical models.

The design patterns shown in this paper are not intended to be comprehensive, but comprise a representative sample of the strategies that could be used to support synchronous groupware. As summarized in figure 2, the design patterns include support for:

- Both co-located and distributed interaction styles, including transitions between them;
- Both asynchronous and real-time interaction styles, including transitions between them;
- The creation of both syntactically correct and free-form artifacts, and the ability to seamlessly move between interactions with one style of artifact to the other;

	Performance	Availability	Usability
<i>Document Replication:</i> User's clients interact with a local copy of a shared document. Copies are synchronized to maintain an application-specific form of semantic consistency.	✓		
<i>Centralized Document Processing:</i> Users interact with a single remote copy of the shared document. Individual clients maintain local views of this remote data.	✓		
<i>Star Topology:</i> All client updates to a shared document are sent to a central hub for broadcast to the rest of the group.	✓	✓	
<i>Mesh Topology:</i> Every client broadcasts local updates directly to every other client.	✓	✓	
<i>Localized Conflict Detection:</i> Update events are broadcast to each client; the client is responsible for resolving conflicts as they occur, e.g., via operation transform.	✓		✓
<i>Central Serialization with Migration:</i> Update events are serialized before being broadcast to all clients, ensuring consistency concurrent updates.	✓		✓
<i>Update Timeout:</i> A ping/echo tactic allowing a client to determine whether an update has been received and processed by a server.		✓	
<i>Dynamic Hub Migration:</i> When multiple clients communicate via a hub located on one of the client nodes, the hub may migrate in case of failure of the hosting node.		✓	
<i>Voting for Reconfiguration:</i> A set of clients votes to decide whether to initiate reconfiguration of the topology.		✓	
<i>Wait, Retry, Resync:</i> A client that has detected a timed-out update briefly operates in "shadow" mode before attempting to update shared state again.		✓	
<i>Event Broadcasting:</i> Events affecting internal documents are forwarded to a central hub for broadcast to all interested clients.			✓
<i>Event Broadcasting with Centralized Coordination:</i> Events affecting external documents are forwarded to a central serializer before being broadcast to all interested clients.			✓
<i>Distributed Directory Services:</i> Each client maintains a directory of every other available client.			✓
<i>Interface Awareness Cues:</i> Each client implements interface features, such as telepointers, that support group awareness.			✓
<i>On-Line Recognition:</i> Free-hand sketches are sent to the recognizer as they are input, rather than batch-processed.			✓
<i>Batch Recognition:</i> Free-hand sketches are sent to the recognizer in batches.			✓
<i>Plug-in Recognizers:</i> Syntax recognizers have generic interfaces.			✓

**Fig. 2.** Extract from quality impact matrix: Summary of design patterns supporting the development of groupware architectures. Checkmarks indicate influence on quality attributes, either positive or negative.

- Both free-form and moderated interaction styles, including transitions between them;
- Interaction through a variety of devices, and movement between them.

To help designers navigate large numbers of design patterns, a *quality impact matrix* is provided (figure 2). This matrix shows the primary quality attributes that each pattern influences (either positively or negatively). Architects interested in improving a particular quality attribute can use the matrix to locate candidate design patterns for use in their architecture.

We now briefly describe two of the 21 design patterns that comprise the candidate pool compiled for the design of groupware tools. For both design patterns, we provide a brief description and an architectural diagram. The diagrams are based on the Workspace Architectural Model [12] (figure 3). The two selected design patterns show architectural alternatives that have equivalent functionality but markedly different influences on quality attributes. An architect would opt for one or the other based on the non-functional requirements specified for his/her particular project. Both design patterns address concurrency control.

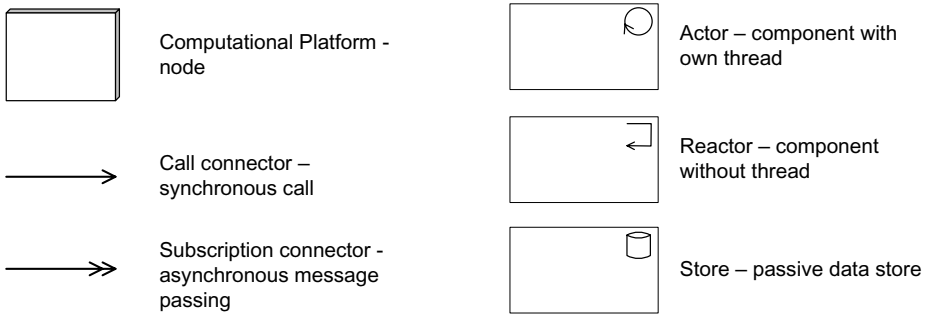


Fig. 3. The Workspace Architecture Notation used in figures 4, 5 and 7

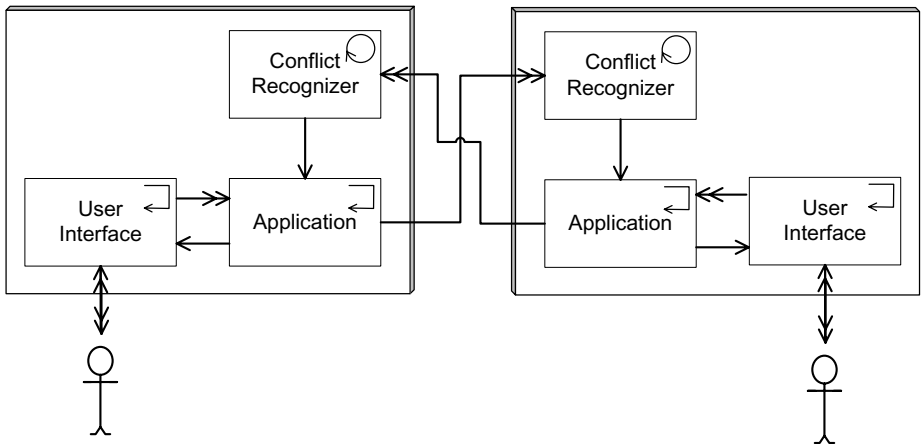


Fig. 4. Localized conflict detection design pattern



### 4.1 Localized Conflict Detection

The goal of *localized conflict detection* (figure 4) is to provide all participants in a groupware session with consistent views of shared state.

In this pattern, update events are broadcast to each client. Clients are responsible for detecting and appropriately resolving conflicts between different users updates. An appropriate implementation for this pattern could be operational transform [6].

Analysis Summary: This pattern influences both *availability* and *usability*. Under availability (section 3.1), Mean Time to Repair benefits from the localization of the conflict recognizer. If one participant’s node fails, the other participants can continue without problem, as they do not rely on the failed node’s state or the state of its conflict recognition. Therefore, partial repair is quick.

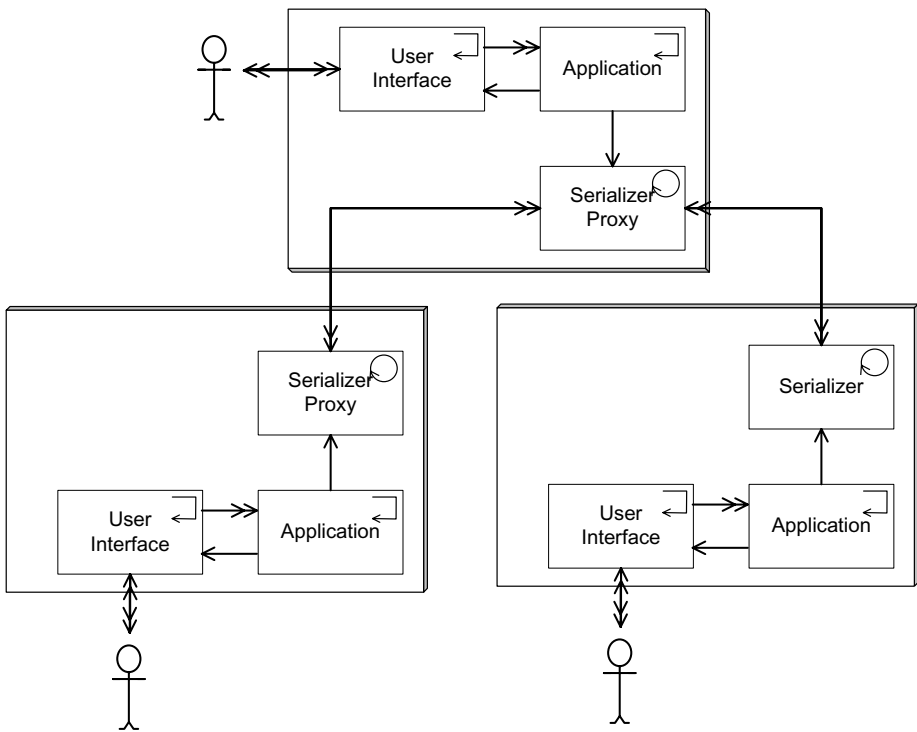


Fig. 5. Centralized Serialization with Migrating Serializer design pattern

Under usability (section 3.2), Fidelity may be improved or worsened by the adoption of this design pattern. Since the conflict recognizer is local, the results of participants’ own actions may be shown immediately, without any need to send messages over the network. In the case of conflicts, however, the view may have to be rolled back. If conflicts between participants’ actions are rare, Fidelity will be good; if conflicts are frequent, rollbacks will be frequent, having a negative effect on Fidelity.

## 4.2 Centralized Serialization with Migrating Serializer

As with the last pattern, the goal of *centralized serialization with migrating serializer* is to provide all participants in a groupware session with consistent views of shared state.

Update events are serialized before being broadcast to all clients. Since events are processed by each client in the same order, all users share a consistent view of the application's shared state. The component responsible for this serialization may migrate between client locations in response to patterns of update traffic. The architecture of this pattern is shown in figure 5.

**Analysis Summary:** This design pattern has an *availability* risk (section 3.1), particularly compared to Localized Conflict Detection. If the node hosting the serializer fails, then the system will be left in a bad state. A recovery algorithm would be required in order to choose a new node for the serializer.

Under *performance* (section 3.3), this pattern can increase Feedback Time relative to Local Conflict Detection because of increased Network times. However, this effect can be mitigated by migrating the serializer, reducing the average network delays experienced by all clients. Similarly, Feedthrough Time may be increased by this pattern due to contention at the centralized serialization component, or because migration of that component has increased the average Transmission Time between all clients. This pattern is particularly applicable to applications where only one user time performs input actions at a time, as the serializer will migrate to that user's computer.

Under *usability* (section 3.2), this approach has both negative and positive effects on Fidelity. Users on nodes with proxy serializers do not see the effects of their own actions until the action has been routed through a serializer on a different node, negatively impacting Fidelity. Conversely, the approach leads to no conflicts or rollbacks, positively affecting Fidelity.

## 4.3 Tradeoffs

The examples of the localized conflict detection and the centralized serialization with migrating serializer design patterns help illustrate the tradeoffs that developers must make when designing the architectures of groupware systems.

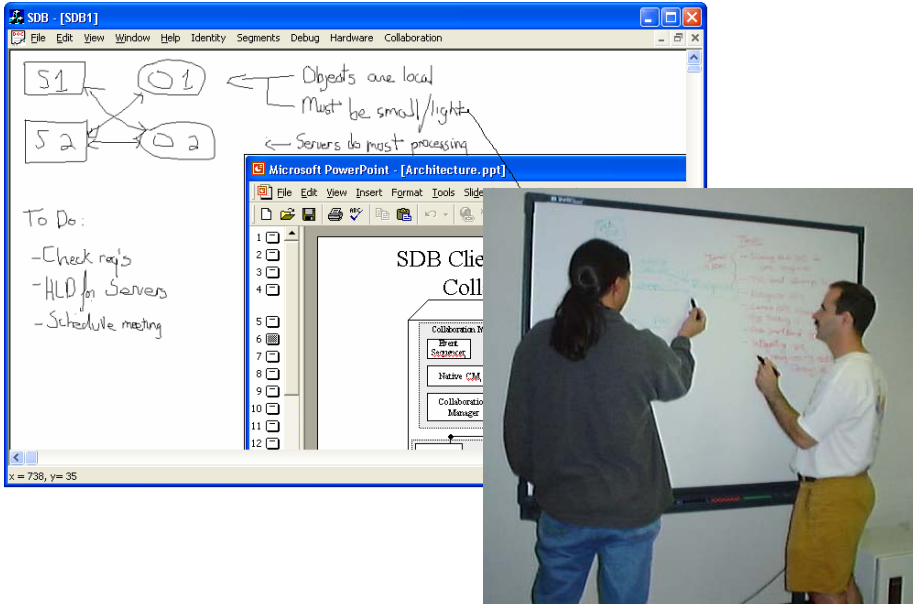
Localized conflict detection has excellent availability, and so is the better choice if good handling of partial failure is desired. Localized conflict detection provides good fidelity if conflicts are rare, but may be a poor choice if conflicts are frequent, leading to frequent undoing of users' actions. Centralized serialization is a good choice if conflicts are more frequent. However, centralized serialization may give poor feedback time; if  $NetworkT ime_{FB}$  is high (over a wide area network), this may be a poor choice. If most interaction is in the form of turntaking, then the migrating serializer will mitigate this problem.

In summary, therefore, localized conflict detection is a good choice when availability is important and conflicts are rare. Centralized serialization is superior if availability is less of a concern and if feedback time is unimportant (or clients are connected by a low-latency network.)

These examples illustrate the detailed analysis of architectural tradeoffs that is possible when design patterns are based on analytical models such as those of section 3.

## 5 Application: The Software Design Board

To gain experience with our quality-centered architectural design framework, we applied it to the development of *Software Design Board*, a tool supporting collaborative software design [13]. In section 5.1, we will show how our quality-centered design framework was used to develop the Software Design Board and discuss its success.



**Fig. 6.** The Software Design Board [13] permits free-hand drawing, automatic recognition of those drawings as structured diagrams, and supports collaborative use via electronic whiteboard or PC clients

The Software Design Board is a whiteboard-based, prototype tool intended to support collaboration in the early stages of software design. The tool supports a variety of styles of work helping in software design, and facilitates transitions between them. This is achieved by integrating informal media and flexible collaboration mechanisms, as well as supporting the migration between different software tools, devices and collaborative contexts. These facilities are intended to support fluid transitions between some of the different styles of work in which we have observed software designers to engage [14].

As can be seen in figure 6, the core of the Software Design Board is its support for free-hand drawing and sketching, appropriate for brainstorming activities. Any number of people can participate in a brainstorming session from different locations, using either an electronic whiteboard or a traditional PC. Each participant sees the drawings of other participants in real-time. Telepointers allow participants to see where other participants are pointing. Gesture-based zooming and panning allows easy management of large drawing areas. Documents created with traditional programs such as Word or PowerPoint can also be embedded in the drawing area.

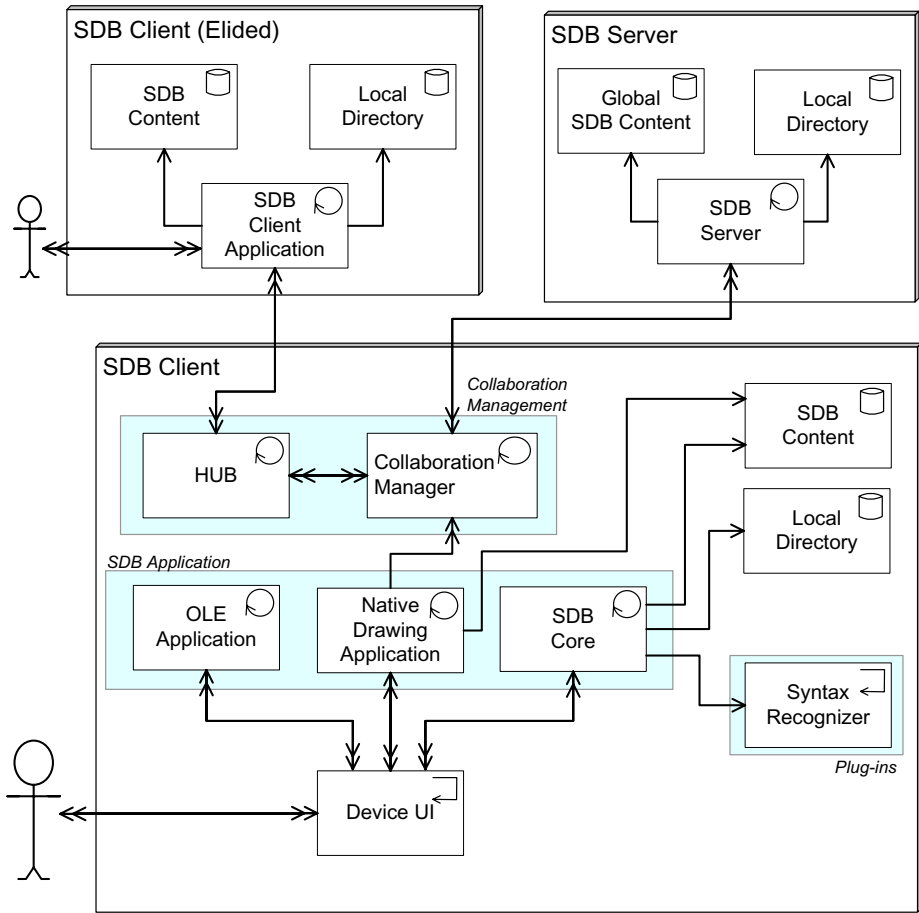


Fig. 7. Architecture of the Software Design Board

Free-hand drawings can be automatically converted to structure-drawings via a diagram recognition function, helping with the transition from rough sketches to formal documentation.

A participant can disconnect from the collaborative session (e.g., while traveling with a laptop), continue work, and merge his/her changes back when next reconnecting. If all participants disconnect, the state of the session is saved, allowing the next person to pick up where the session left off, using any device from any location.

The Software Design Board motivates quality requirements typical of groupware applications. It is important for partial failure to be handled effectively; if a participant's computer or network connection fails, the other participants should be able to continue uninterrupted. Security may be a significant issue, as design discussions may include sensitive data that should not be intercepted by malicious parties. Performance is important, as significant latency may inhibit the natural flow of discussion. And perhaps most importantly, the tool must enable natural collaboration, ensuring that participants easily understand the actions of other participants.

## 5.1 Architecture of the Software Design Board

In this section, we show which of the design patterns outlined in section 4 were selected and combined into the architecture of the Software Design Board.

The high-level architecture of the Software Design Board is described in figure 7. Each client application (*SDB Client Application*) maintains a local copy of all data (*SDB-Content*), as well as a directory of contact information (*Local Directory*) of people with ongoing collaborations. Each client application also interacts with a central server (*SDB-Server*), which maintains a global copy of all data. Additionally, the server maintains a global directory containing contact information for all clients in the system.

The *SDB Client Application* is expanded into four subsystems – *Collaboration Management*, *SDB Application*, *Plug-ins* and *Device UI*. The Collaboration Management Subsystem is responsible for managing shared data. The SDB Application Subsystem is responsible for the applications themselves, i.e., the native drawing application, control of external OLE applications and general functionality of the SDB itself (e.g. gesture interpretation.) The Plug-ins Subsystem maintains plug-in components, such as the free-hand drawing syntax recognizer. Finally, the Device UI Subsystem encapsulates the device-dependent user interfaces.

This architecture represents the composition of several of the design patterns summarized in figure 2:

- *Document Replication*: Each node maintains a local copy of its data (*SDB-Content*). The client applications (*SDB Client Application*) broadcast update events to each other in order to synchronize the distributed copies. This pattern was chosen over Centralized Document Processing for performance reasons.
- *Star Topology*: This is used to broadcast changes in the free-hand drawings to all session participants. The *Native Collaboration Manager* sends/receives events to/from a *Hub* component, which broadcasts those events to interested application components (other *Native Collaboration Managers*) in other nodes. Although this pattern has worse availability than the Mesh Topology, it was chosen to reduce the required number of network connections. The availability issue was addressed by the use of *Dynamic Hub Migration*, as described below.
- *Dynamic Hub Migration*: Within the star topology, a *Hub* is present on every node, facilitating migration of the broadcasting functionality between nodes. This pattern is effective when combined with the *Star Topology* pattern.
- *Distributed Directories*: Each node maintains a local directory of relevant peers (*Local Directory*). This directory is initially obtained from the server (*Global Directory*). Subsequently, clients directly broadcast relevant directory updates to each other in order to maintain current distributed directories without constantly checking the server for updates. A distributed directory has superior performance and availability to a centralized directory service.
- *Online Recognition*: The SDB performs structural recognition of hand-drawn diagrams. The application component (*SDB Core*) invokes the structure recognizer (*Syntax Recognizer*) before updating the local data

(*SDB Content*). This is performed for every update event received from the user interface.

- Online recognition was superior to Batch Recognition since it supports realtime feedback to the user.
- *Interface Awareness Cues*: A variety of interface awareness cues are implemented as part of the *SDB Core*, including telepointers and zooming/scrolling functionality.

The central question in evaluating our experience with quality-centered architectural design is whether the requirements of the Software Design Board were met. The approach helped us to methodically assess which of a set of design patterns best addressed the application's requirements. The quality impact matrix helped in identifying the design patterns of interest. The analysis frameworks effectively provided a vocabulary for discussing the tradeoffs between patterns, allowing the choices summarized above. Once the application was built, its performance, usability and availability requirements were met as far as possible within a prototype tool.

The framework is a work in progress, and should be extended both to provide additional design patterns and additional quality attributes. Two new quality attributes of particular interest are security and development time. Security heavily influences how well an application respects the user's privacy, a question of enormous importance to groupware users. Estimates of development time place a significant reality check on architectural design, as the desired architecture may simply not be realizable within the available time or budget.

## 6 Analysis and Related Work

The work described in this paper builds extensively on earlier work in taxonomies of quality attributes [4, 8] and catalogues of the relationship between software architecture and quality attributes [1, 3]. These lines of research have attempted to identify architectural styles that achieve particular quality attributes. Additionally, there have been other systematic attempts to document the relationship between software architecture and quality attributes, including the Non-Functional Requirement Framework [5] and Attribute Driven Design [2].

Our experience with developing the Software Design Board leads us to a number of conclusions about Quality-Centered Design of software architectures.

First, we emphasize the importance of QCAD frameworks being domainspecific. If the domain is too broad, the framework developer will have an unreasonable number of design patterns to specify and analyze. Similarly, the complexity of the analytical models will grow, as a wide range of quality concerns need to be taken into account. It is practical to apply this approach if the domain is sufficiently narrow to keep the development of the framework tractable. Others have had success with domain-specific frameworks, most notably in the area of human-computer interaction [9] and IT systems [11].

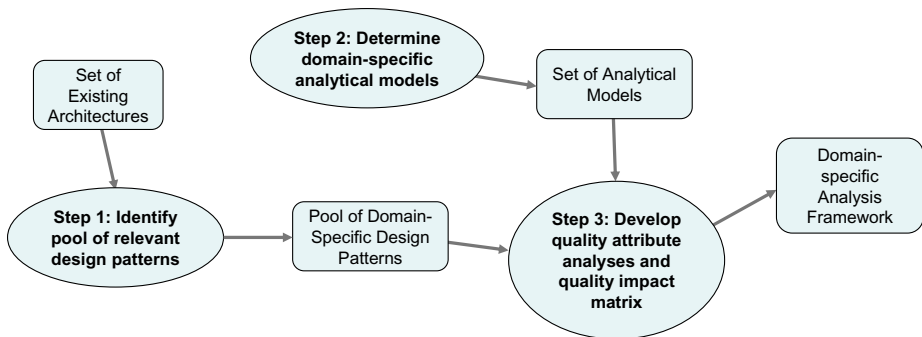
The choice of design patterns to populate the framework is itself challenging. There is a constant tension between specifying many orthogonal design patterns with limited functionality versus fewer design patterns with more functionality. The former approach is more general, allowing design patterns to be more easily combined, possibly even in ways that the framework developer did not foresee. The latter approach

makes it easier for users of the framework to pick patterns of interest and combine them into architectures. Over all, making design patterns too fine-grained can lead to an explosion of patterns, while too coarse a granularity may make them hard to combine and may lead to important cases being missed.

Our experience shows that analytical models may be quantitative or qualitative. For example, our Availability and Performance models are based on measurable phenomena, while our Usability model is more subjective. Even with quantitative models, our reasoning is ultimately qualitative: it is difficult to provide a numeric value capturing the effect of a design pattern. There has been some progress in creating and validating analytical models in the groupware area [10] and in performance in general [11], but substantially more work is required. Of these approaches, we favour work that validates analytical models over approaches that require architects to do extensive mathematical analysis of their designs, simply in order to obtain results in a timely fashion. Particularly, as the required analysis becomes more complex, there is likely diminishing return on investment.

Nevertheless, the approach is useful now, as QCAD frameworks support methodical reasoning about the properties of software architectures. For groupware developers, even the experience of thinking about how quality attributes such as availability and security affect the user experience is highly beneficial. The framework as it stands already represents a significant advance over ad-hoc design.

Throughout our work, we gained experience in the development of QCAD frameworks, of which our groupware framework is one example. Figure 8 summarizes the steps required to create a new framework for a new domain. Our approach is similar to Bass *et al.*'s Attribute-Driven Design method, differing primarily in our use of design patterns as the unit of design, rather than ADD's more abstract tactics.



**Fig. 8.** How a framework developer populates a quality-centered design framework

A framework developer must first mine a set of existing applications to isolate useful design patterns, resulting in a *pool of domain-specific design patterns*. It is important to emphasize that each QCAD framework is specific to a relatively narrow domain, such as the development of groupware.

In order to help designers evaluate the tradeoffs between design patterns, *analytical frameworks* must be developed. The analytical frameworks are used to develop analytical advice associated with each design pattern, as well as a *quality impact matrix* used to help navigate the pool of patterns.

## 7 Conclusion

In this paper, we have presented a quality-centered design framework for groupware applications. This framework is an example of a more general approach in which domain-specific frameworks can be developed to help architectural design. We have illustrated the framework through its application to a significant groupware application, the Software Design Board. We have shown how the Software Design Board is constructed by combining design patterns suggested by our QCAD framework.

## Acknowledgements

This work benefitted from the generous support of the Natural Science and Engineering Research Council of Canada, the Ontario Centres of Excellence, and the Network for Effective Collaboration Technologies through Advanced Research.

## References

1. Bass, L., Clements, P., Kazman, R.: *Software Architecture in Practice*, 2nd edn. SEI Series in Software Engineering. Addison-Wesley, Reading (2003)
2. Bass, L., Klein, M., Bachmann, F.: Quality attribute design primitives and the attribute driven design method. In: *Software Product-Family Engineering*. LNCS, pp. 169–186. Springer, Heidelberg (2001)
3. Bergery, J., Barbacci, M., Wood, W.: Using quality attribute workshops to evaluate architectural design approaches in a major system acquisition: A case study. Technical Report CMU/SEI-2000-TN-010, Software Engineering Institute (2001)
4. Boehm, B., Brown, J., Kaspar, H., Lipow, M., McLeod, G., Merrit, M.: *Characteristics of Software Quality*. North Holland, Amsterdam (1978)
5. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, Dordrecht (2000)
6. Ellis, C., Gibbs, S.: Concurrency control in groupware systems. In: *Proceedings of the ACM Conference on the Management of Data (SIGMOD 1989)*, Seattle, WA, USA, May 2–4, pp. 399–407. ACM Press, New York (1989)
7. Ellis, C., Gibbs, S., Rein, G.: Groupware: Some issues and experiences. *Communications of the ACM* 34(1), 38–58 (1991)
8. International Organization for Standardization, International Electrotechnical Organization. International Standard ISO/IEC 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use
9. John, B., Bass, L., Segura, M., Adams, R.: Bringing usability concerns to the design of software architecture. In: *Proc. Engineering Human-Computer Interaction/ Design, Specification and Verification of Interactive Systems*. LNCS, pp. 1–19. Springer, Heidelberg (2004)
10. Junuzovic, S., Chung, G., Dewan, P.: Formally analyzing two-user centralized and replicated architectures. In: *Proc. ECSCW 2005*, pp. 83–102. Springer, Heidelberg (2005)
11. Koziolok, H., Firus, V.: Empirical evaluation of model-based performance prediction methods in software development. In: *Quality of Software Architectures*. LNCS, pp. 188–205. Springer, Heidelberg (2005)



12. Phillips, W.G., Graham, T.C.N., Wolfe, C.: A calculus for the refinement and evolution of multi-user mobile applications. In: Gilroy, S.W., Harrison, M.D. (eds.) DSV-IS 2005. LNCS, vol. 3941, pp. 137–148. Springer, Heidelberg (2006)
13. Wu, J., Graham, T.C.N.: The Software Design Board: A tool supporting workstyle transitions in collaborative software design. In: Proc. Engineering Human-Computer Interaction/ Design, Specification and Verification of Interactive Systems. LNCS, pp. 363–382. Springer, Heidelberg (2004)
14. Wu, J., Graham, T.C.N., Smith, P.: A study of collaboration in software design. In: 2003 International Symposium on Empirical Software Engineering (ISESE 2003), Rome, Italy. IEEE Computer Society, Los Alamitos (2003)

## Questions

### **Prasun Dewan:**

*Question: Does your work allow for optimization of combinations of parameters/ For example, high awareness compensates for low consistency management. This is an apparent trade-off, but not a real one, as the usability does not degrade because of low consistency.*

Answer: The user will simply pick an architecture with high awareness and low consistency management.

### **Laurence Nigay:**

*Question: Would it be possible that design patterns re not compatible?*

Answer: It is a loop mechanism, back to the quality factor.

### **Phil Gray:**

*Question: This approach is based on the identification of requirements which drives the analysis and assessment. However, requirements are subject to change. How would/could you handle this fact?*

Answer: Requirements always subject to change. Basically, we should always do the best we can to anticipate potential change and design with that in mind.

*Question: What about “malleability” or “support for change” as a quality attribute for an architecture?*

Answer: Yes. We don't have that, but it would be a great idea.

# Programs = Data + Algorithms + Architecture: Consequences for Interactive Software Engineering

Stéphane Chatty

ENAC, Laboratoire Informatique et Interaction,  
7 avenue Edouard Belin, 31055 Toulouse Cedex, France  
and  
IntuiLab, Prologue 1, La Pyrénéenne, 31672 Labège Cedex, France  
<http://recherche.enac.fr/~chatty>

**Abstract.** This article analyses the relationships between software architecture, programming languages and interactive systems. It proposes to consider that languages, like user interface tools, implement architecture styles or patterns aimed at particular stakeholders and scenarios. It lists architecture issues in interactive software that would be best resolved at the language level, in that conflicting patterns are currently proposed by languages and user interface tools, because of differences in target scenarios. Among these issues are the contravariance of reuse and control, new scenarios of software reuse, the architecture-induced concurrency, and the multiplicity of hierarchies. The article then proposes a research agenda to address that problem, including a requirement- and scenario-oriented deconstruction of programming languages to understand which of the original requirements still hold and which are not fully adapted to interactive systems.

## 1 Introduction

Niklaus Wirth, renowned computer science teacher and programming language designer, wrote in 1975 a reference book entitled “Algorithms + Data structures = Programs” [1] that has influenced thousands of programmers. It may be that his equation was incomplete though. Software architecture, that is the way of organising software into interconnected parts, has progressively become recognized as a central issue in programming and software engineering, to the point where students now spend more time learning about patterns and frameworks than data and algorithms. Yet, software architecture is still mostly considered a separate issue from programming languages. We contend that this is a serious issue for the software engineering of interactive systems. Short of being able to write “Programs = data + algorithms + architecture” and addressing architecture issues at the language level, the architecture of interactive software may be doomed to inconsistency and complexity.

The architecture of interactive software has been heavily studied and many influential results in software architecture were obtained by researchers with a background in interactive software, or derived from their work. Compare for example the authors and topics in the following list of publications: [2-10]. Still, very few actors of the

domain consider that the situation of interactive software architecture is satisfactory: teaching these issues is still awkward, and programming interactive software remains complex as soon as one does not stick to common WIMP interfaces. The author's personal experience in selling interactive software design and solutions was a very instructive field study of that problem: most potential customers of interactive software technology are put off by perceived incompatibilities between the processes of user interface design and traditional software engineering, or even more explicitly by software incompatibilities [11]. For instance, customers had to renounce implementing the chosen design when finding that implementing it with Java Swing would cost four times the cost of a WIMP interface, just because of architecture mismatches.

In this article, we propose an analysis of the relationships between software architecture, programming languages and interactive software, based on the principles of requirements and usage scenarios. We highlight a strong coupling between languages and architecture, and propose that languages can be studied using the same methods. We then use this analysis to identify some requirements and scenarios where current programming languages and interactive software conflict and thus favour inconsistent or costly architecture solutions. User interface toolkits act as architectural patches to languages, but the result is not always consistent. Finally, we propose a research agenda for addressing that issue, considering that user interface development brings at the same time new problems and techniques for addressing them. Architecture issues can be addressed by identifying the underlying usage scenarios more explicitly before applying the body of knowledge created for programming languages. Doing so, in addition to helping to understand interaction architecture, could help improve programming languages.

## 2 Of Programming Tools, Scenarios and Architecture

The software engineering and the user interface design communities have come up with similar models of requirements engineering and design for software products. With some differences in vocabulary, they share the concepts of stakeholders, external requirements or goals, technological choices or constraints, scenario-or usecase-based design, task or process analysis, and iterative design [12,13]. These design models have proven effective over the years for designing tools and (in many cases) improving the efficiency of the final users.

These models can be applied to the design of a special category of tools: the tools made for software builders themselves. Programming languages are tools for programmers; development environments are tools for programmers and project managers; user interface toolkits are tools for programmers and interface designers; some specialized languages are aimed at non-professional programmers, and so on. Some of these tools are developed with a focus on a given technology and aimed at specific tasks, for instance logic programming for knowledge management. Some have to take into account constraints such as the performance of compilers or computers. But all of them were designed, explicitly or not, with stakeholders and usage scenarios in mind. That is, they take into account all the persons that are concerned with the product because they build, manage, or use it and they try to capture the multiple activities around the product through concrete stories called scenarios or use cases. Many

language designers used themselves as the target users, made their own scenarios mentally, and performed initial iterations by testing the candidate designs against their mental scenarios. Others, such as the designer of Perl, used the whole user community for a vast participatory design process. In all cases, understanding the underlying scenarios and requirements provides a powerful means for analysing architectures, languages and other tools.

In the following sections, we identify the types of stakeholders and scenarios that underlie the state of the art in software architecture, programming languages and interactive software architectures. We will later use that analysis to detect some plausible causes of the problem of interactive software architecture.

## 2.1 Software Architecture

One definition of software architecture is “the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their evolution over time” [14] or in other words, how to split programs in smaller parts and glue them together. In their seminal paper on software architecture, Garlan and Shaw analyse architectural styles by focusing on the nature of components and the glue that links them [15]. Software architectures are not tools for building software, but rather rules, guidelines, or patterns for the same purpose. Nonetheless, the above reasoning on scenarios applies, in that an architecture style is a design aimed at supporting some scenarios of software building for stakeholders of the software industry. Programming tools are complete and implemented designs, whereas architectures styles are partial designs. Some architecture styles come with supporting tools. Others are more theoretical and let their users choose how to implement them, either because they address issues orthogonal to those addressed by available tools, or because they conflict with them (see the section on Interactive software architecture below for examples).

Architectures, like tools, are aimed at sparing their users from some design choices by providing a good solution adapted to their goals. For instance, a “pipes and filters” architecture like that of the Unix shell focuses on the needs of three types of stakeholders involved in the production of data analysis software: the programmers of basic analysis algorithms, who are encouraged to isolate their algorithms in separate programs, thus avoiding the details about how their algorithms will be used; the shell programmers, who are encouraged to implement a simple interface for connecting program inputs and outputs, and know that their shell will be usable in various situations; and finally power users who can build custom analysis chains at a very low cost.

The role of scenarios is recognized by the software architecture community [16]. Admittedly, no architecture style is well adapted to all situations. The identified stakeholders include the end user, framework programmers, administrators, and maintainers. Scenarios include development, debugging, parameterising, all sorts of software reuse, and even off-shoring. It is recognised that the type of application (databases, interaction, AI, etc) is an important aspect of scenarios too [15]. It is interesting to note, however, that most of the literature on software architecture focuses on scenarios and techniques beyond a certain granularity of code. Most proposed definitions of software architecture suggest that it deals with medium and large-scale software components. Garlan and Shaw present software architecture as the third level of a scale where the first two levels are high-level programming languages and abstract data types.

## 2.2 Programming Languages and Hardware Design

It is also interesting to analyse languages and even computers through the looking glass of scenarios and architecture. Actually, many constructs in programming language are aimed at software architecture rather than algorithms or data structure. Most literature shows that all programming languages and even computing hardware enforce certain architecture styles and were built with certain stakeholders and scenarios in mind. It also hints that expressions in programmer lore such as “clean”, “elegant” or “orthogonal” are actually scenario-based architecture quality statements.

In the prehistory of computing, Jacquard looms were machines that executed programs coded on punch cards. The system was split into two components (machines and cards) so as to support a standard scenario involving two actors: the same machine built by a maker could afterward be used by an operator to produce different weaving patterns by changing cards. That architecture style where the central engine is fixed and smaller parts of the execution process can be changed at will was very influential on Ada Lovelace. She built upon the idea to propose that Babbage's analytical engine could be used to tabulate different mathematical functions by using different cards. She also used it to suggest that functions could be computed several times with different data [17]. Later Turing invoked similar computing scenarios to propose splitting the sequence of operations executed by the Automatic Computing Engine into “subsidiary operations” [18]. He also proposed instructions named BURY and UNBURY and a stack structure to support that architecture, thus laying out the foundations of the call stack. Support for implementing it was soon built into computers and since then has been present in the microcode of most processors.

Just like computing hardware, programming languages have been deeply influenced by these historical scenarios: a fixed engine executing interchangeable computations, or programmers splitting their code into several sequences so as to call the same sequence several times. The concept of function, procedure or subroutine borne from these scenarios is present in most languages. The design rationales written by language designers are dense with references to such scenarios. For instance Stroustrup [19] mentions “communication between designers and programmers” (p. 114) as a goal, states that “the issue of how separately compiled program fragments are linked together is critical” (p. 34), and that “C with Classes was explicitly designed to allow better program organisation; computation was considered a problem solved by C” (p. 28). Actually, languages such as Pascal, C++ or Java abound with features aimed at facilitating the splitting of programs into reusable parts: functions, name scoping, namespaces, typing, classes, etc. These features implement a style that is strongly suggested to programmers: split your programs in functions so that you can reuse them at will. Hence we claim that languages support the “Programs = data + algorithms + architecture” equation, and we observe that most languages are still based on the historical computation scenario.

True enough, the evolution of mainstream languages has been focused on supporting more and more complex software engineering scenarios. First it was observed that the functions paradigm could be used to support such uses as documenting, reading and maintaining code, or detecting errors. Then came more complex scenarios: a first programmer develops a library of functions that other programmers will reuse in their programs; or a programmer builds a computation engine in which other programmers

later insert their own computation functions; or a programmer builds a specialisation of a library and inserts it into a computation engine, etc. These scenarios are supported by features such as separate compilation, late binding, interfaces or exceptions. This evolution was possible because clever engineers always found how to extend the basic paradigm to support these scenarios: they were compatible with the historical architecture.

Alternate programming paradigms have been proposed: functional, logical, reactive or parallel programming. But usually the proposed justifications were about the expressive power of languages for a given programmer, not about architecture or scenarios involving multiple stakeholders. If some of these paradigms induce architecture styles that diverge from the historical style, this is apparently just a side effect. For instance, when Backus criticised “von Neumann languages” and proposed the functional style [20], he did it at the level of programming instructions, not at the level of combining larger parts of programs. Some of his arguments used architectural concepts (“language framework versus changeable parts”), but his concern was at a very fine grain and that did not lead him to challenge the functions paradigm. And the truth is that the ability of this paradigm to be applied to all situations is apparently unlimited.

### 2.3 Interactive Software Architecture

Nevertheless, after nearly 30 years of research history, interactive software does not seem to be part of that success story:

The user interface research community periodically debates about the reasons why so little of its successful research makes it to commercial products, and software issues are among the proposed explanations;

Programming rich user interfaces is still considered a highly complex task, and teachers still look for solutions to make their students able to create working interactive components during their courses;

Researchers working on new interaction styles often express frustration at current tools or build their own;

Many works have been devoted to software architecture, models and patterns for interactive software, which confirms that there are stills problems that need solving; the fact that research in the domain has considerably decreased is most likely not due to a sense of successful achievement;

Very few results have been integrated into programming languages, unlike with other software engineering works;

Industries in the defence, aerospace, automotive, or home automation industries are still looking for technologies that combine the results of user interface research and their current development tools;

The implementation of many interactive systems uses some sort of middleware, which frees architects from the constraints of languages by creating their own language (the middleware protocol) to glue components; the fast evolution of Web user interfaces is probably an example of this.

We propose to analyse causes of this situation by comparing the architecture styles induced by languages and those proposed for interactive software. We first try to identify the software engineering scenarios behind the proposed interactive software architectures, before identifying some conflicts in a later section.

One of the most cited reference is the Seeheim architecture model, proposed at a time when the problem at hand was retrofitting existing software with new graphical user interfaces [21]. This scenario was new because it required to organise software along two dimensions. The first axis was as usual a split into one fixed and one interchangeable parts: the functional core and the user interface. The second axis dealt with the varying location of execution control, which depends on the nature of the user interface: control is split between the functional core and the user interface for text user interfaces, and it resides within the user interface when it is graphical. These requirements led to propose a four-tier architecture pattern. However that was done at a very high level of abstraction, not explaining how that was related to programming constructs, probably because there was no obvious solution for that. When the Seeheim model was refined later into the Arch model, new tiers were added to accommodate more complex reuse scenarios including multiview user interfaces, but once again no relationship with programming languages was set forth [22]. This means that programmers are free to implement the architecture as they wish. But this freedom comes at a high cost, just as if programmers of classical programs had kept on coding in assembly language. More detailed architecture styles have been proposed. PAC [23] had the same aims as the Seeheim and Arch model, but with more concrete handling of concerns such as the hierarchical organisation of components. However it was no more based on programming language constructs.

In contrast with these architecture styles aimed at changing user interfaces, a series of architecture styles or patterns have been proposed and implemented as toolkits or frameworks to address more programmer-oriented needs [24]. The “Inversion of Control” (IoC) or “Dependency Injection” pattern recently gained popularity [26]; it captures the fact that containers are usually coded before the objects they contain even though they pass control to them at execution time. Earlier, a series of graphical toolkits have used the callback pattern or the late binding technique provided by object-oriented languages [4,5,25]. The MVC (Model-View-Controller) pattern focused on graphical rendering and input handling, relying on constructs of Smalltalk, a rare language built with user interaction scenarios in mind [9,27]. Some authors proposed to connect program components through one-way constraints [28] or dataflow connections [29] so as to support program readability and interchangeability of components, or even adaptation to execution platforms, in the context of direct manipulation and animation. With similar use scenarios in mind, but with a focus on graphical rendering, others have proposed to isolate graphical computations in components linked together by a hierarchical glue named a scene graph [30]. Others have proposed to isolate states and reactions to events in components based on finite state machines, Statecharts or Petri nets [31]. Others have noticed that architecture styles proposed by alternative programming styles matched some scenarios of interactive software development: tools were developed using the functional programming [32], the reactive programming [33], or the parallel programming paradigms. Some even tried to merge user interface programming deeply into the syntax of existing languages to try and force the compatibility of user interfaces and programming languages, see for instance the Ubit toolkit that makes heavy use of the operator overloading feature of C++ [34].

The theoretical architecture styles such as Seeheim, Arch or PAC could not fail: they represent real concerns and do not face “implementation details”. The more

implementation-oriented solutions were not as successful, even though most of them strike by their elegance. Apart from MVC and the Smalltalk environment, they all fall into one of these two categories:

The general purpose tools, which are widely used but considered as yielding complex architectures and limiting the evolution of user interfaces;

And the more specialized tools, which are not widely used, probably because the local help they provide conflicts with the requirements of the other parts of the software or the architecture style of the underlying language. In the rest of this paper we attempt to analyse the reasons behind this mixture of success and failure, and we propose a research agenda to address them.

### 3 A Multi-level View of Software Architecture

We observe that all the tools and architecture styles mentioned in the previous section are concerned with architecture at different levels of granularity. All levels propose to split applications into components in a way that efficiently supports scenarios where parts of the software are created at different times by different persons, but they deal with components of different sizes.

#### 3.1 Four Levels of Architecture

Architecture can be considered at four levels with growing component sizes:

1. The lowest level is that of *programming instructions*: how can they be grouped and reused, for instance in iterations? We are used to juxtaposing instructions, but Turing identified that as a design choice: “A simple form of logical control would be a list of operations to be carried out in the order in which they are given” [18] Lisp or Occam do not rely on that implicit semantics of grouping. As for control structures, patterns are proposed that favour different reuse scenarios (using an assignment in a test, for instance). This level of architecture is handled by languages and processors: they define a data model, a set of instructions, and ways of organising them. All underlying usage scenarios have one stakeholder: a programmer who writes, reads, and debugs a small piece of code, usually at the scale of one page.
2. The next level deals with *structuring chunks* of programs: how do I split my code in sequences that are at most one page long and that can be reused at several places? That level deals with the needs of programmers or groups of programmers working on the same part of a program. It deals with scenarios such as documenting code, communicating about it, optimising or debugging it. Most languages handle it through functions or classes, or through alternate constructions such as continuations.
3. Then comes the level of *software reuse*, customisation and extension by different actors. Common stakeholders are groups of programmers that either split work and integrate it later or reuse libraries and frameworks built earlier. Others are project managers, maintenance managers and technical writers. Recently, engineers who deploy and parameterise software, or even users, have become stakeholders at that level. For classical software, that level has been handled by tools like preprocess-



sors and linkers, then by languages, then more recently by architecture patterns and systems of plug-ins. For interactive software, it has been the focus of user interface management systems, toolkits or frameworks. Interactive software has been a great provider of research on that level, and the works listed previously are solutions pending for consideration. For instance, events were recently included in C# [35].

4. The highest level is *software planning*, concerned with reusing whole applications or groups of applications. It deals with stakeholders such as information directorates in companies, computer providers, software houses and scenarios such as product line management, deployment, etc. Expressions such as “software urbanism” [36] have been coined for this level, which we do not address here.

Taking the perspective of tool design, the first two levels are aimed at single users (the programmers), and the third level is more about groupware design (development teams). These levels cannot be handled independently.

### 3.2 Managing Compatibility

All levels cannot be addressed by a single tool. For instance it was decided to handle in operating systems issues that were best not handled in languages. However, a lot of research has been aimed at handling more and more of the higher levels in languages. The step from level 1 to level 2 was made very early; the step from level 2 to level 3 has started with FORTRAN II (the introduction of separate compilation) and is probably not over. Two probable reasons for that tendency are:

- A wish to minimise the number of concepts or patterns manipulated by programmers; once they are in a programming language or a processor, they can be used at all levels with no additional cost;
- Once a pattern has proved its value and compatibility with the language, a desire to encourage programmers to use that pattern rather than invent others which might prove incompatible and dangerous.

**Compatible patterns.** These two points highlight the importance of having compatible patterns throughout the four levels and especially within a given level. Patterns are compatible when they can be combined so that all scenarios they support individually are supported by the combination, without adding complexity. For instance, functions and object-oriented programming can be made compatible by deciding that object methods are functions. This allows to combine components written with either pattern. If compatibility is not retained programmers are led to creating code that has not the expected behaviour because the programmer had wrong expectations. At best this necessitates special documentation and training for programmers; at worst, programmers may try to introduce new concepts or syntaxes, succeeding only in masking the problems. For instance, message passing and functions can appear similar for architecture purposes but are based on different synchronisation models; mixing them is dangerous because the programmer's code may be executed in an unexpected way. Consequently, an architecture level should only use a subset of the connectors provided by the lower level (or compatible connectors), and its component types should be refinements of component types of the lower level. When incompatible patterns are identified at different levels, one can build middleware that adapts connectors: a RPC

library or a message bus, for instance. The additional cost is acceptable between levels 2 and 3, or 3 and 4, but not within level 2 or 3.

**Pattern lifecycle.** Another consequence of the two points above is the lifecycle of architecture patterns that they describe. Solutions are first proposed to programmers in tools that act as additions or modifications (“patches”) to the underlying language. When an addition or modification proves safe and beneficial to a large audience, it ends up being part of a new language. Most user interface toolkits or frameworks provide both additions and modifications. The additions are interactive objects and algorithms: graphics, interaction management, gesture recognition, etc. The modifications are new level 3 or even level 2 architecture patterns: data-flow, scene graph, continuations, etc. The same holds for operating systems. Consider for instance the `select` call of Unix or the message queues of Windows: they provide mechanisms that are not native to the C (resp. C++) language and that allow asynchronous communication.

In the above lifecycle, additions usually stay out of the language. As for modifications, three states are possible:

Compatible modifications waiting for inclusion in a language, if someone can devise a clever way of including them;

Modifications that have been identified as incompatible and either force the use of a middleware layer or limit the usefulness of the toolkit.

Modifications that have not been identified as incompatible, and make the toolkit difficult or even dangerous to use.

**Compatibility as a goal.** Ideally of course, one would be able to design compatible architecture patterns that answer all known software engineering scenarios of a given domain, and thus ultimately build a language that supports that domain. That language would offer a component model and a linking mechanism that would hold at all levels and allow to build “fractal” software where the architecture patterns would be the same at all levels of hierarchy of the software. That would, among other things, make middleware useless. That would also allow the implementation of multilanguage solutions at level 3, such as Microsoft’s .Net which allows the use of different languages for addressing different application parts. But it seems that the current situation today is that most proposed solutions for interactive systems are in the second or third state above. As stated before, this makes programming interactive systems more difficult and error-prone than necessary. This also has dire consequences on project management and user interface quality, encouraging to develop user interfaces at the end of projects when constraining architectures are already in place.

An exception to this situation would be the Smalltalk environment, which was explicitly designed along the lines of architecture consistency: “Smalltalk’s design — and existence— is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block (...)” [37] Even then, the limited industrial success of Smalltalk suggests that some key scenarios where not taken into account, the foremost being probably the interconnection with non-interactive software. C++ took the opposite stance, making it harder to develop interactive software. That shows how much understanding the possible architecture mismatches is important.

## 4 Understanding Mismatches

We now propose a few reasons why architecture patterns proposed at level 3 for interactive software display incompatibilities with those offered by most programming languages. Most reasons listed below stem from the same cause: interactive software involves new stakeholders and generates new engineering scenarios. If we except project managers, maintenance managers or technical writers, most scenarios described earlier in this article involved programmers who build their own programs by including components written by others, or insert their components into existing computation engines. User interface design and development multiplies the roles: it introduces interaction designers, graphical designers, developers of low fidelity prototypes, developer of the final application, framework developer, developers of device drivers, interactive component developers, users setting parameters of their application, etc. All these stakeholders have different backgrounds and use different tools, and they generate complex development scenarios. The complexity is similar to that of very large systems, even though a single program is produced. This partly comes from a new step of software engineering: it focused on programmers, then on software engineering groups, and now needs to focus on multidisciplinary software engineering groups [38].

### 4.1 New Reuse Patterns

Software reuse defines a partial order relation between components: to reuse a component, a programmer must know how to address it, and uses that in the newly written component. This relation fostered many constructs in programming languages: names given to functions or variables, typing, encapsulation to hide details, name rewrite to provide growing levels of abstraction, etc. This binary relation is well adapted to scenarios where programmers add layers upon layers of code. It is not to scenarios involving other types of stakeholders, because in that case there are more than one reuse relations. That challenges many mechanisms, starting with encapsulation:

An interface designer or a user who changes a font in an application accesses a property name defined by the programmer of a text field; that name is not accessible to other programmers; consequently, components should have several interfaces depending on the type of stakeholders: developers of new interaction modalities, interactive component developers, application programmers, graphical designers, users;

Even among programmers, the order relation may vary; for established concepts, the language and its core library reuse and encapsulate the operating system (see for instance the standard input in C); but with innovative user interfaces the application programmer is often also a device driver programmer, who for instance configures a wireless remote control to behave as a mouse; this requires framework developers to provide extension mechanisms for all operating systems, and breaks the traditional encapsulation hierarchy;

Encapsulation usually supposes that the reused component is complete, whereas interface skinning or the multidisciplinary development of components leads to splitting components in halves that are managed independently: a programmer will develop the behaviour and a graphical designer the looks, for instance. This lessens the added value of class derivation.

## 4.2 Contra-Variance of Reuse and Control

One of the most common reuse scenarios in interactive software is that of event sources: picking a target in graphics scenes or interpreting speech is hard enough that one prefers to reuse existing libraries. Reusing these components has led to event-driven programming and to the progressive replacement of graphical libraries by programming frameworks. This reuse pattern is fundamentally different from the historical reuse scenarios. Consider the partial order relation introduced in the previous section (reuse relation) and compare it with another partial order relation: that which relates two components when one transfers control to another one (control relation). In the historical reuse scenarios, the two relations are covariant: the caller knows the callee, because the main program is written after the libraries or at least linked later. With interactive software, the main program is still written last but initiative always comes from external sources: timers, network peers, or input devices. The two relations are thus contra-variant.

This contra-variance has been accounted for in diverse ways: event-driven dialogue, main loops, callbacks, programming frameworks, IoC pattern, are all toolkit-level solutions for supporting it. However, we believe that it should be handled at a more basic level, because it characterises the most important reuse pattern in interactive software. Apart from their initialisation, there are few situations where components are in a “covariant reuse” situation; actually, it is possible to describe fairly rich user interfaces without the concept of function, whereas it is impossible without a solution for the “contra-variant reuse”.

Apart from the additional cost and complexity induced by this inversion of priorities between languages and interactive software, it causes several problems:

Event emission is a good basis for encapsulating components: a button emits either `press` or `release`, a dialogue box with two buttons only emits `ok` or `cancel`, and so on; managing it outside of languages deprives programmers from that encapsulation;

There are solutions for providing both dataflow and event emission with a unified model; having function calls as the predominant paradigm in programs makes it difficult to implement, in particular because of diverging semantics as for sequencing;

Using the functions paradigm creates a misunderstanding with functional core programmers: it does not help them to detect that user interfaces cannot be programmed as mere function calls, and pushes many teams to restrain to interface components that can be used with the functions paradigm;

And finally it plays a role in the “inversion of calendar” problem that strikes many large projects: when a user interface design is chosen towards the end of a project, managers realise that the architecture chosen years before does not allow it. Indeed, it is logical to choose an architecture early enough: at the beginning, the interface is still in the iterative design phase and there are other developments to start. But with no knowledge of the interaction styles that will be chosen one can only resort to the common denominator, which currently appears to be the function call, whereas the only certain thing is that it will not be the function call. It is therefore necessary to promote a basic pattern that accommodates the contra-variant reuse pattern, and if possible the covariant one for the commodity of functional core development.

### 4.3 Locality of State and Computations

When reading software or locating errors, locality of behaviours is an important feature: having one page per algorithm makes it easier to use a divide-and-conquer approach. Functions are fit for that purpose when programs mostly consist of algorithms: each function implements a computation, which in addition makes computations reusable. However, computations and algorithms play a more minor part in interactive software. Most behaviours consist in managing a state, its modifications upon events, and the associated actions. For instance, leaving the graphical objects aside, a visual button is essentially made of a state (disabled, idle, pushed, etc) and ways of changing it. In computation-oriented programs functions are essential and data can be hidden in the call stack, and that led to functional programming. With interaction, state is essential in behaviours and the locality principle would require that all code that changes it is grouped. That pushed researchers to propose programming patterns based on finite state machines, Statecharts or Petri nets, but:

When using a computation-oriented language, the transitions are implemented as functions or methods and the principle of locality is not met;

Functions and transitions are not as easy to match as functions and methods: all uses of function arguments do not easily transpose to transitions, and the expected sequencing properties are not always the same;

In the same way as functions can be combined in complex ways, many development scenarios involve the combination of several behaviours; for instance, a blinking icon has two orthogonal behaviours: the blinking, and the ability to be dragged across the screen; state management should allow to separate and combine states at will, just like for functions;

States and behaviours are an important part of reuse scenarios and thus should be part of the reuse patterns: with interactive systems, programmers do not reuse components by adding functions to them; they add event reactions or animations as much as they would change the graphical looks;

In addition to be combined or reused, behaviours sometimes need to be structured hierarchically: levels in a game or steps in a wizard are high level states that influence lower level behaviours such as the speed of targets or the enabling/disabling of buttons; hierarchical state machines are a local solution that mixes badly with the software reuse scenarios;

Finally, not all behaviours have the same focus on state transitions; some, often represented by dataflows, are made of successive computations that alter quantitative states. Animation, for example, relies on combining algorithms to compute the positions of graphical objects. This creates a continuum between computations, dataflows, and state-transitions that would require a uniform organisation pattern.

### 4.4 Architecture-Related Concurrency

Interactive systems require concurrency in few situations only. When reading large documents, the user should be able to interact with the system even when the program is busy loading the file. For most other situations, one only needs to rely on the interleaving of external events which all occur asynchronously. However, software engineering scenarios and architecture induce some form of concurrency that needs to be handled properly.

Consider a program that emits events when the user clicks on an icon. Classical interactive software engineering scenarios lead to providing that component in a library, so that programmers can reuse it and bind their code to events it emits. It may happen that several components are connected to this event source. For instance, an application programmer can bind both the modification of a text field and the opening of a dialogue box, both obtained from two widget programmers. Suppose the box emits a sound then an animated feedback when opening, and the text changes with an animation. Then for all purposes, these two widget programmers are in a concurrent situation: neither knows about the actions coded by the other, and nevertheless the application programmer may want to ensure a sequencing order: sound first then animations, for instance. That requires that the programming environment allows to express sequencing constraints on the actions triggered by events. This requirement is rarely fulfilled, and many commercial programs exhibit strange behaviours with that regard.

As usual, one may be tempted to handle this requirement with the concepts or the syntax of the underlying language. For instance, the author used an animation library that encapsulated sequencing in a functional programming style. It was very elegant to use, except that it had to be implemented through nested event loops, and when sequencing more than two animations, the first animation might get stuck and the program continued its execution with two nested mainloops. Trying to hide the concurrency only made it bite programmers later. The safe solution is to use a concurrent language or a system of threads and semaphores, which forces user interface programmers to absorb complex concepts and does not make it easy to explicit sequencing properties of their code.

## 4.5 Multiple Hierarchies

Programming languages manage two hierarchies in programs. First, they give an important role to the lexical hierarchy of code to manage components. Most names are visible only within a given lexical scope, which plays an important role in defining reusable functions and components. Languages like C++ associate the lifecycle of objects to their lexical scope. Some languages, like Occam, even use lexical scopes to define the concurrent or sequential execution of instructions. Second, most languages introduce a hierarchy of types or classes that is often used to represent a hierarchy of domain concepts. Interactive systems require that other hierarchies are managed by the language or toolkit, and can rely very little on syntax. When a component is made of sub-components, these can:

- Be created in a given lexical scope and use the names defined in that scope;
- Be derived from another type of components, using the class hierarchy proposed by object-oriented languages;
- Belong to a given modality (graphics, speech, etc) and occupy a certain position in a modality-specific hierarchy (scene graph or widget containment for instance); that is the hierarchy seen by the specialist of that modality;
- Influence the execution of their parent and sibling components, for instance because their sizes is used by the layout algorithm, because their current state influences the behaviour of another component, or because their mere presence changes the nature of the user interface: consider for instance a graphics layer that removes

all colours from the interface whenever a modal dialogue box is displayed. There are multiple independent behaviour hierarchies, relatively independent from each other. For all these hierarchies, it is tempting for programmers either to map them to the existing hierarchies in languages, or to build one's own set of graphs. The first option often yields conflicts. For instance, it is tempting to use a class hierarchy to represent the nature of components: a hierarchy of graphical object classes, a hierarchy of speech object classes, etc. This potentially leads to very complex class hierarchies when containers are present: can graphical groups contain speech objects? can windows contain animation trajectories? The latter option creates less complexity but forces programmers to build their own hierarchy management system, which cannot benefit from services provided by the language for its own hierarchies, such as renaming and encapsulation.

Furthermore, language hierarchies are limited to the scope of programs. They do not scale up to applications built as several programs. To do so, one needs to use middleware such as Corba, which provides a multi-program class hierarchy but at a very high cost. Ideally, a language should provide a hierarchy management that supports the hierarchies found in interactive systems, and valid at all levels of granularity, thus enabling to handle programs like components.

## 5 Related Work and Research Agenda

This is not, by far, the first attempt at analysing the nature of programming languages and their issues. To begin with, all language designers appear to have carried out a critical analysis of existing languages. As already discussed in this paper, most did it with programmers in mind. Examples include Backus on functional programming [20], Kay on Smalltalk [37] or Stroustrup on C++ [19]. Prominent software engineering essayists often carry out the same type of analysis, based on their experience of industrial development; see Graham for a recent example [39]. Some researchers have tackled the issue of dealing with more complex software engineering scenarios. Aspect programming [40] and the meta-object protocol [41] are examples of that approach. Software architecture specialists have identified the problem of architecture mismatch [14] and analysed their causes and consequences, at a generic level. Several researchers from the interactive software community worked on resolving some mismatches posed by interactive software. For instance, Prospero is aimed at solving issues between different levels of tools in CSCW software development [42]. Wegner even goes further and challenges the very fact that algorithms should be central in programming, proposing interaction as the key construction [43].

Our approach focuses on architecture and relies on the conviction that user interface development brings both problems and techniques for addressing them. A first list of problems has been presented in this article. The techniques are those of user interface design: requirements engineering and design techniques for usercentred design. We are convinced that an explicit use of these techniques, often used implicitly by language designers, can help understand the needs of interactive software stakeholders, the solutions proposed, and how to match them. Our experience with the user-centred design of the graphics module of a user interface environment [38] strengthens that conviction. We therefore propose a research agenda that could help

understand to what extent solutions currently proposed by programming languages can be used for or adapted to the efficient development of interactive systems, or how they could be modified to support the expected development scenarios without forfeiting their other qualities. This agenda includes:

- Reviews of the software engineering and programming language literature to identify all stakeholders and scenarios taken into account in these domains;

- Identification of stakeholders and scenarios with modern and/or future interactive software;

- Measurements of how these scenarios are handled in current software;

- Identification and classification of requirements and properties expected from interactive software development tools and languages;

- Deconstruction of programming languages and theories to identify the supported architecture patterns and the underlying scenarios;

- Identification of the patterns in traditional or alternative languages that support the desired scenarios, and those that potentially conflict with them; this may lead us to discover that some works in interactive software architecture have exact equivalent in programming language research;

- Research of compatible patterns that support the scenarios from interactive software; in other words, re-application of the working methods of the language and software engineering communities once the deconstruction has been performed, including formal methods;

- Construction of a set of basic instructions and patterns adapted to interactive software, so as to build the equivalent of Microsoft .Net for developing interactive software with languages adapted to each part (graphical interface, functional core, speech interface, dialogue, etc).

## 6 Conclusion

In this paper, we have proposed to analyse programming languages and interactive software in terms of software architecture and in terms of stakeholders and scenarios supported by architectures. We have suggested that software architecture is present at several levels of granularity, the finest grain being handled by programming languages. We have described user interface toolkits as providing modifications to the architectures proposed by languages. We have listed several issues where languages and interactive software bring conflicting patterns, causing complexity that must be managed by programmers and that impedes innovation in user interaction. Finally, we have proposed a research agenda based on the identification of stakeholders, scenarios and architecture patterns that involves the application of language design techniques to interactive software tools or even interactive software languages. User interface design teaches us that humans are able to adapt to various designs, sometimes accepting systems that make them relatively inefficient. How much of this coadaptation is at work when we build user interface tools based on languages? So far, the user interface community has mostly focused on “getting the job done with the tools provided”, that is producing the expected user interfaces and taking the rest of software tools as immutable. Maybe we need some usability experts for ourselves!



**Acknowledgements.** This article finds its roots in a long conversation with M. Beaudouin-Lafon in Palos Verdes, CA in 1994. Some ideas came from there or my later work with P. Palanque and J. Accot at CENA. The rest came from an extreme experience at IntuiLab in 2002-2004: trying to apply to ourselves participatory design as taught by W. Mackay in the design of IntuiKit. Finally, marketing work with D. Figarol helped me articulate the arguments. S. Conversy, P. Dragicevic and M. Beaudouin-Lafon helped to improve the paper.

## References

1. Wirth, N.: Data structures + algorithms = programs. Prentice Hall, Englewood Cliffs (1975)
2. Kruchten, P., Sotirovski, D.: Implementing dialogue independence. *IEEE Software* 12(6), 61–70 (1995)
3. Kruchten, P.: The Rational Unified Process — an Introduction. Addison-Wesley-Longman (1999)
4. Linton, M.A., Vlissides, J.M.: The design and implementation of InterViews. In: Proceedings of the USENIX C++ Workshop (1987)
5. Weinand, A., Gamma, E., Marty, R.: ET++ -an object-oriented application framework in C++. In: OOPSLA 1988 Proceedings (1988)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
7. Bass, L., Coutaz, J.: Developing software for the user interface. The SEI Series in Software Engineering. Addison Wesley, Reading (1991)
8. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley, Reading (1998)
9. Krasner, G., Pope, S.: A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk 1980. *Journal of Object-oriented programming* 1(3), 26–49 (1988)
10. Barrett, R., Delany, S.J.: OpenMVC: a non-proprietary component-based framework for web applications. In: Proceedings of the 13th international WWW conference (2004)
11. Chatty, S., Sire, S., Lemort, A.: Vers des outils pour les équipes de conception d'interfaces post-WIMP. In: Actes d'IHM 2004, pp. 45–52. ACM Press, New York (2004)
12. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: ICSE 2000: Proceedings of the Conference on The Future of Software Engineering, pp. 35–46. ACM Press, New York (2000)
13. Muller, M.J., Kuhn, S.: Participatory design. *Commun. ACM* 36(6), 24–28 (1993)
14. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch: Why reuse is so hard. *IEEE Software* 12(6), 17–26 (1995)
15. Garlan, D., Shaw, M.: An introduction to software architecture. In: Ambriola, V., Tortora, G. (eds.) *Advances in Software Engineering and Knowledge Engineering. Series on Software Engineering and Knowledge Engineering*, vol. 2, pp. 1–39. World Scientific Publishing Company, Singapore (1993)
16. Kazman, R., Abowd, G., Bass, L., Clements, P.: Scenario-based analysis of software architecture. *IEEE Software* 13(6), 47–55 (1996)
17. King, B., Lovelace, A.: Notes by the translator of the Sketch of the Analytical Engine invented by Charles Babbage, by L.F. Menabrea. *Scientific Memoirs* 3, 666–731 (1843)
18. Turing, A.M.: Proposals for the development in the mathematics division of an Automatic Computing Engine (ACE). Technical Report E882, Executive Committee, NPL (1946)

19. Stroustrup, B.: *The Design and Evolution of C++*. Addison-Wesley, Reading (1994)
20. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM* 21(8) (1978)
21. Pfaff, G.E. (ed.): *User Interface Management Systems*. Eurographics Seminars. Springer, Heidelberg (1985)
22. Bass, L., Pellegrino, R., Reed, S., Seacord, R., Sheppard, R., Szezur, M.R.: The Arch model: Seeheim revisited. In: *CHI 1991 User Interface Developers Workshop* (1991)
23. Coutaz, J.: PAC, an implementation model for dialog design. In: *Proceedings of the Interact 1987 Conference*, pp. 431–436. North Holland, Amsterdam (1987)
24. Myers, B.A.: A brief history of human-computer interaction technology. *Interactions* 5(2), 44–54 (1998)
25. Beaudouin-Lafon, M., Berteaud, Y., Chatty, S.: Creating direct manipulation interfaces with X<sub>TV</sub>. In: *Proceedings of EX 1990, London*, pp. 148–155 (1990)
26. Martin, R.C.: *Agile Software Development: Principles, Patterns and Practices*. Pearson Education, London (2002)
27. Goldberg, A.: *SMALLTALK 1980, the Interactive Programming Environment*. Addison-Wesley, Reading (1984)
28. Myers, B.A.: Creating user interfaces using programming by example, visual programming and constraints. *ACM Transactions on Programming Languages and Systems* 12(2), 143–177 (1990)
29. Chatty, S.: Defining the behaviour of animated interfaces. In: *Proceedings of the IFIP WG 2.7 working conference*, pp. 95–109. North-Holland, Amsterdam (1992)
30. Strauss, P.S.: Iris inventor, a 3d graphics toolkit. In: *OOPSLA 1993: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pp. 192–200. ACM Press, New York (1993)
31. Palanque, P., Bastide, R.: Petri net based design of user-driven interfaces using the interactive cooperative object formalism. In: *Proceedings of the DSV-IS 1994 workshop*, pp. 383–401. Springer, Heidelberg (1994)
32. Dannenberg, R.B.: Arctic: A functional language for real-time control. In: *Proceedings of the ACM Conference on Lisp and Functional Languages*, pp. 96–103 (1984)
33. Clement, D., Incerpi, J.: Programming the behavior of graphical objects using Esterel. In: Díaz, J., Orejas, F. (eds.) *TAPSOFT 1989 and CCIPL 1989*. LNCS, vol. 352. Springer, Heidelberg (1989)
34. Lecolinet, E.: A molecular architecture for creating advanced GUIs. In: *Proceedings of the ACM UIST*, pp. 135–144 (2003)
35. Venners, B., Eckel, B.: The C# design process. a conversation with Anders Hejlsberg (2003), <http://www.artima.com/intv/csdes.html>
36. Desreumaux, M., Oudrhiri, R.: Information and software systems: from architecture to urbanism. In: *Proceedings of the 1st IFIP Working Conference on Software Architecture*. Chapman & Hall, Boca Raton (1998)
37. Kay, A.C.: The early history of Smalltalk. *ACM SIGPLAN* (3), 69–75 (1993)
38. Chatty, S., Sire, S., Vinot, J., Lecoanet, P., Mertz, C., Lemort, A.: Revisiting visual interface programming: Creating GUI tools for designers and programmers. In: *Proceedings of the ACM UIST*. Addison-Wesley, Reading (2004)
39. Graham, P.: *Hackers and Painters: Big Ideas from the Computer Age*. O'Reilly Media, Sebastopol (2004)
40. Kiczales, G.: Aspect-oriented programming. *ACM Computing Surveys* 28(4) (1996)
41. Kiczales, G., des Rivières, J., Bobrow, D.G.: *The art of the meta-object protocol*. MIT Press, Cambridge (1991)

42. Dourish, P.: Using metalevel techniques in a flexible toolkit for CSCW applications. *ACM Transactions on Computer-Human Interaction* 5(2), 109–155 (1998)
43. Wegner, P.: Why interaction is more powerful than algorithms. *Communications of the ACM* 40(5) (1997)

## Questions

### ***Prasun Dewan:***

*Question: Regarding the influence of programming languages, all programming languages are Turing complete. Just because you find a language difficult to use could mean you don't know how to use the language.*

Answer: I have lots of examples, but indeed it is really hard to prove it.

### ***Helmut Stiegler:***

Answer: The language related notion of a control stack goes beyond a data-driven way of a processing model according to “last-in-first-out”. The notion is based on a “processing context” of a unit of processing (usually called a “procedure”) being automatically handled by an implicit mechanism and being independent from data visibly accessed by the unit of processing. This kind of control stack was introduced by Bauer and Samualtou in the Algol language, and a patent was granted to them.

# Towards an Extended Model of User Interface Adaptation: The ISATINE Framework

Víctor López-Jaquero<sup>1</sup>, Jean Vanderdonckt<sup>2</sup>, Francisco Montero<sup>1</sup>,  
and Pascual González<sup>1</sup>

<sup>1</sup>Laboratory on User Interaction & Software Engineering (LoUISE)  
Universidad de Castilla-La Mancha, 02071 Albacete, Spain  
{victor, fmontero, pgonzalez}@dsi.uclm.es

<sup>2</sup>Belgian Laboratory of Computer-Human Interaction (BCHI)  
Université catholique de Louvain, 1348 Louvain-la-Neuve, Belgium  
jean.vanderdonckt@uclouvain.be

**Abstract.** In order to cover the complete process of user interface adaptation, this paper extends Dieterich's taxonomy of user interface adaptation by specializing Norman's theory of action into the ISATINE framework. This framework decomposes user interface adaptation into seven stages of adaptation: goals for adaptation, initiative, specification, application, transition, interpretation, and evaluation. The purpose of each stage is defined and could be ensured respectively by the user, the interactive system, a third party, or any combination of these entities. The potential collaboration between these entities suggests defining additional support operations such as negotiation, transfer, and delegation. The variation and the complexity of adaptation configurations induced by the framework invited us to introduce a multi-agent adaptation engine, whose each agent is responsible for achieving one stage at a time (preferably) or a combination of them (in practice). In this engine, the adaptation rules are explicitly encoded in a knowledge base, from which they can be retrieved on demand and executed. In particular, the application of adaptation rules is ensured by examining the definition of each adaptation rule and by interpreting them at run-time, based on a graph transformation system. The motivations for this multi-agent system are explained and the implementation of the engine is described in these terms. In order to demonstrate that this multi-agent architecture allows an easy reconfigurability of the interactive system to accommodate the various adaptations defined in the framework, a case study of a second-hand car-selling system is detailed from a simple adaptation to progressively more complex ones.

**Keywords:** Adaptation, adaptation configuration, delegation, isatin, ISATINE framework, mixed-initiative user interface, multi-agent system, negotiation, re-configuration of user interface, transfer, user interface description language.

## 1 Introduction

We are witnessing a paradigm shift in the interaction with computers. The progressive migration of applications from desktop PCs to mobile platforms is changing the habits of user in interaction. Furthermore, a new mass of computer interaction neophytes is

becoming attracted to the possibilities of using computer applications to support many daily tasks, such as buying flight or theater tickets. At the same time, as communications and hardware sensors cost gets cheaper the availability of information to the applications is quickly increasing. To take advantage of this increase in the information available to the application from the context of use where they are executed, adaptation mechanisms that adjust the application according to the data received from the context of use need to be devised. For this purpose, a multitude of adaptation techniques are been used [3,11,14].

Currently, the most widely accepted understanding of the adaptation process comes from Dieterich's survey of adaptation techniques [5], despite that it has been produced in 1993. In addition to its age, Dieterich's taxonomy suffers from several shortcomings: it is constrained by only entities (e.g., the user and the system) in each stage of the adaptation process, it does not handle an explicit collaboration and it is restricted to the execution only. Furthermore, some of the most relevant issues in the adaptation process such as how the adaptation is specified were left out of the framework. In particular, Dieterich's taxonomy is incomplete with respect to the seven stages of Norman's theory of action [14]. This model describes how a user interacts with an application from the beginning, when the user is forming his intention to reach a goal, until the end, when the user evaluates the results from the actions taken to achieve the goal.

This paper expands Dieterich's framework by incorporating some extra stages adapted from the mental model proposed by Norman. These extra stages improve user involvement in adaptation process and foster a more detailed description of how the adaptation process is carried out. To validate the proposed framework, an architecture supporting the framework is also presented. The architecture has been designed as a multi-agent system to enable easy extensibility and to make more natural the design of the negotiation, transferring and delegation capabilities required for the adaptation stages proposed in our framework to be executed collaboratively.

This paper starts by describing the ISATINE adaptation framework (Section 2), along with the antecedents that have motivated and inspired it. Section 3 introduces a multi-agent architecture that supports the proposed framework and describes how each adaptation stage proposed in the framework is supported. A discussion on how the designed multi-agent architecture has been implemented is delivered in Section 4. Section 5 exemplifies the framework by applying the framework and the architecture on a running example: a second-hand car selling application with various levels of adaptation. Some conclusions and future work are reported in Section 6.

## **2 The ISATINE Framework for User Interface Adaptation**

This section first introduces Dieterich's taxonomy of user adaptation in order to identify its shortcomings, thus initiating an extension according to Norman's theory of action for user interaction [14] resulting in the ISATINE framework. This framework is defined in the second subsection.

### **2.1 Dieterich's Taxonomy of User Adaptations**

On the one hand, Dieterich's taxonomy of user adaptations has always been considered as a seminal reference for classifying different types of user interface adaptation

configurations and techniques. This paper sorted more than 200 papers dealing with various forms of user interface adaptation and summarized them into four stages needed to perform any form of adaptation, in principle:

1. **Initiative:** one of the entities involved in the interaction suggests its intention to perform an adaptation. The main entities are usually the user and the system.
2. **Proposal:** if a need for adaptation arises, it is necessary to make proposals of adaptations that could be applied successfully in the current context of use for that need for adaptation detected.
3. **Decision:** as we may have different proposals from the previous stage, which adaptation proposal best fit the need for adaptation detected should be decided, and whether it is worth applying each proposal.
4. **Execution:** finally, the adaptation proposal chosen will be executed. One important factor when making any changes in the UI is how the transition from the original UI to the adapted one is performed. Before the execution stage, a prologue can be executed to prepare the UI for the adaptation. For instance, if the adaptation includes switching from one code to another code, the prologue function should store the current state of the application, so it can be resumed after the adaptation takes place. On the other hand, an epilogue function can be provided to restore the system after adaptation takes place. This epilogue will take care of restoring application state and resuming the execution of the application.

On the other hand, we are considering Norman's mental model of user interaction which decomposes any user interaction into seven Stages of Action:

1. Forming the Goal: the user shapes a goal in her mind.
2. Forming the Intention: to reach the goal, the user is forming some intention.
3. Specifying an Action: the intention is turned into a series of actions.
4. Executing an Action: one action at a time is selected and executed.
5. Perceiving the State of the World: after that the action has been executed, the results produced by this action are perceived.
6. Interpreting the State of the World: the results perceived trigger an interpretation in the user's mind on how the World has changed.
7. Evaluating the Outcome: depending on this interpretation, the user evaluates whether the action she executed matches her initial goal or not.

If we attempt to match Dieterich's four stages of adaptation on Norman's model, it can be observed straightforwardly that the initiative corresponds to the intention, that the proposal and the decision are two steps involved in the action specification, and that both execution stages match (Fig. 1). Therefore, only some portion of the whole process, the left part of Norman's model, is covered, thus creating a need for covering the remaining uncovered portion. This expresses some current shortcomings such as: the results of adaptation should be made perceivable in a way that is appropriate enough for the user to understand it. Not only the adaptation results could be made perceivable, but also the adaptation execution itself. Too often interactive systems supporting some adaptation do not convey properly the idea and the meaning of the adaptation process. Empirical studies have shown that users are always confused to some extent when they face some adaptation. If nothing is implemented in the system to minimize this effect, the adaptation process is likely to be rejected. Fig. 1 does not reveal when the adaptation is performed by the user (adaptable user interface) vs. by

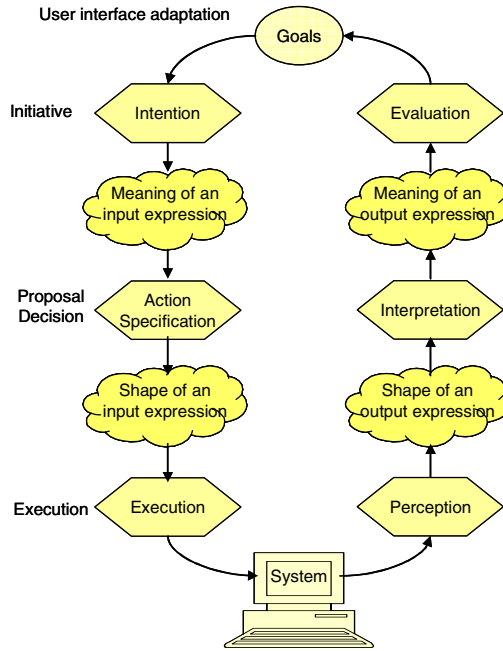


Fig. 1. The four steps of Dieterich’s taxonomy located on Norman’s mental model

the system (adaptive user interface). In Norman’s model, goals are typically expressed by a human trying to interact with the system. Therefore, there is a need to better identify the roles of each entity. Dieterich’s model does not decompose very much the adaptation process into sub-processes, thus leaving some room for more expressivity.

**2.2 Definition of the ISATINE Framework**

The shortcomings identified in the previous subsection lead us to expand Dieterich’s taxonomy by trying to express the adaptation process according to all the Seven Stages of Norman’s model. In this way, it is expected that no adaptation stage will be left out. Basically, we state that three entities are involved in the adaptation process: the *user* (U), the *interactive system* (S), or any *third party* (T), which may substitute the two previous entities in case of need (e.g., request for help, further support, support for some operation which is impossible to achieve otherwise, failure). When at least two entities share the responsibility of a stage, there is a need for coordinating the input and output of these entities. For instance, mixed-initiative [9] represents a typical configuration when U and S collaborate to determine the best option possible for ensuring a stage. We distinguish three forms of coordination:

1. *Negotiation*: options could be presented by each entity and the final result is negotiated between the entities so as to reach a consensus. T could serve for this purpose when, for instance, contradicting output are produced by U and T. Or for stating which entity has the higher priority.

2. *Delegation*: when an entity estimates that it does not have information or responsibility enough to achieve the adaptation stage, it may request help/support from any other entity to achieve its purpose. When the results come back to the requesting entity, it may then decide the final option, therefore keeping the control over the decision process.
3. *Transfer*: this form is the same as delegation, but without any return to the requester. The requested entity takes the decision and may send a notification.

The specialization of Norman's model for adaptation results into the ISATINE<sup>1</sup> framework, so-called because the Seven Stages became seven adaptation stages, each one being specialized for each entity (Fig. 2):

1. *Goals for user interface adaptation*: any entity (U, S, or T) may be responsible for establishing and maintaining up-to-date a series of goals to ensure user interface adaptation. Although this adaptation is always for the final benefit of the user, it could be achieved with respect to any aspect of the context of use (with respect to the user herself, the computing platform used by the user, or the complete physical and organizational environment in which the user is carrying out her task). The goals are said to be *self-expressed*, *machine-expressed*, *locally* or *remotely*, depending on their location: in the user's head (U), in the local system (S), or in a remote system (T). A typical example of machine-expressed goals is encountered when the system is made responsible for maintaining a certain level of fault-tolerance depending on varying network or hardware conditions. This main goal could be further decomposed into sub-goals, like keeping a minimal amount of information, ensuring a graceful degradation [7] of the user interface, or avoiding any task disruption.
2. *Initiative for adaptation*: this stage is further refined into formulation for an adaptation request, detection of an adaptation need, and notification for an adaptation request, depending on their location: respectively, U, S, or T. For example, T could be responsible for initiating an adaptation when an update of the UI is made available or there is a change of context that cannot be detected by the system itself (e.g., an external event).
3. *Specification of adaptation*: this stage is further refined in specification by demonstration, by computation, or by definition, depending on their origin: respectively, U, S, or T. When the user wants to adapt the UI, she should be able to specify the actions required to make this adaptation, such as by programming by demonstration or by designating the adaptation operations required. When the system is responsible for this stage, it should be able to compute one or several adaptation proposals depending on the context information available. When the third party specifies the adaptation, a simple definition of these operations could be sent to the interactive system so as to execute them.

---

<sup>1</sup> An orange-red crystalline substance, C<sub>8</sub>H<sub>5</sub>NO<sub>2</sub>, obtained by the oxidation of indigo blue. It is also produced from certain derivatives of benzoic acid, and is one important source of artificial indigo (Source: <http://dictionary.reference.com/>)



4. *Application of adaptation*: this stage specifies which entity will apply the adaptation specified in the previous stage. Since this adaptation is always applied on the UI, this UI should always provide some mechanism to support it. If U applies the adaptation (e.g., through UI options, customization, personalization), it should be still possible to do it through some UI mechanisms.
5. *Transition with adaptation*: this stage specifies which entity will ensure a smooth transition between the UI before and after adaptation. For instance, if S is responsible for this stage, it could provide some visualization techniques, which will visualize the steps, executed for the transition, e.g., through animation, morphing, progressive rendering [15].
6. *Interpretation of adaptation*: this stage specifies which entity will produce meaningful information in order to facilitate the understanding of the adaptation by other entities. Typically, when S performs some adaptation without explanation, U does not necessarily understand why this type of adaptation has been performed. Conversely, when U performs some adaptation, she should tell the system how to interpret this evaluation. For instance, [6] develops a machine-learning algorithm where the system first proposes some adaptation to be applied. If this adaptation does not correspond to users' needs, the user provides the alternate adaptation instead and tells the system how to incorporate this new adaptation scheme for the future. The system updates the knowledge base by interpreting this explanation.
7. *Evaluation of adaptation*: this stage specifies the entity responsible for evaluating the quality of the adaptation performed so that it will be possible to check whether or not the goals initially specified are met. For instance, if S maintained some internal plan of goals, it should be able to update this plan according to the adaptations applied so far. If the goals are in the users' mind, they could be also evaluated with respect to what has been conducted in the previous stages. In this case, the explanation of the adaptation conducted also contributes to the goals update. Collaboration between S and U could be also imagined for this purpose.

The only stage, which could not be a priori ensured by U or T, is the execution, unless the user is a programmer or the third part supports dynamic programming.

The deviation between the initial expression of goals for UI adaptation and those specified in terms of the system is referred to as the *adaptation semantic distance in input*. When an adaptation operation is adequately specified, the deviation between this specification and the operations required to achieve the adaptation step is referred to as the *adaptation articulatory distance in input*. The sum of these two deviations denotes the gulf of adaptation execution and represents how complex it could be to represent and execute the adaptation operations in the system's terms. Similarly, the deviation between the perception of the adaptation as performed on the UI and the perception of the user denotes the *adaptation articulatory distance in output*. The difference between the goals reached so far in the system and the initial goal denotes the *adaptation semantic distance in output*. The sum of these two deviations denotes the gulf of adaptation evaluation and represents how complex it could be to evaluate the results of the adaptation. This second gulf is too often forgotten in adaptation algorithms, although it is largely reported (e.g., in [3,5]) that any adaptation, however good and adequate it could be, always provokes some perturbation in the user's mind. By reducing this gulf, the perturbation should be able to be minimized.

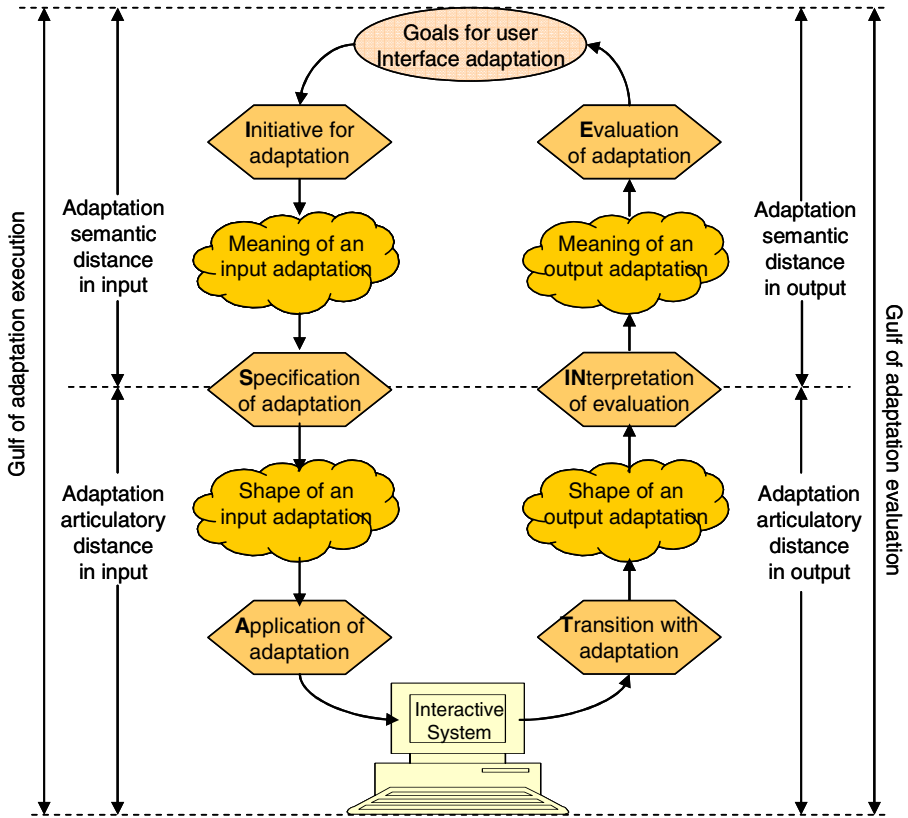


Fig. 2. The seven stages of the Isatine framework for user interface adaptation

### 3 A Multi-agent Architecture Supporting ISATINE Framework

The previous section identified some holes in the support of a complete adaptation process, which is also reflected in some lacks of system support for these stages. Indeed, the lack of general techniques, methods and tools for adaptation design produces systems where the support for adaptation is rather inflexible, and the knowledge injected into the adaptation engine is very hard to be reused. In the design of a general technique that supports adaptivity in a flexible manner, where knowledge can be reused and integrated with a user interface design method that provides the required formalism to build UIs in a systematic way, a software architecture is required able to cope with all these requirements. However, this software architecture should be able to decide which adaptation could be applied, when they should be applied, etc. Therefore, a dedicated software architecture is required, where the system is able to make some reasoning and to decide what to do next (which adaptation to apply, if any).

Different reasoning models have been proposed so far: rule based systems, neural networks, Bayesian networks, etc. However, a great interest has appeared for software agents [19] as a means to represent reasoning capabilities in an abstract manner

similar to human reasoning. Most of them use the BDI model (*Beliefs, Desires, Intentions*) [8,18], which is inspired by human reasoning theories. *Beliefs* represent the view the agent has of itself and the world where it is immersed. *Desires* describe the goals that the agent is trying to achieve. Finally, *Intentions* are the plans the agent is executing to achieve the goals it pursues. Because the designed architecture supporting the ISATINE framework should be able to manage negotiation, delegation, and transferring between the different stakeholders in adaptation process (the user, the system or a third-party) multi-agent systems are especially suitable, since there is already some work done within agents research community regarding how the different agents involved in a multi-agent system collaborate or compete negotiating, delegating or transferring duties. Another advantage found in multi-agent systems is the natural distribution of computation, which supports the integration of the implemented multi-agent system with exiting services easily. Furthermore, software agents have already proved useful in the interaction between the user and the UI in some projects such as [8,18]. Those were our motivations to design an architecture to support the ISATINE framework as a set of agents collaborating in a multi-agent system to achieve the final goal: adaptation. Next, how the different stages of the adaptation process defined in ISATINE framework are carried out by the multi-agent system created will be addressed.

### 3.1 Goals for User Interface Adaptation

The goals for user interface adaptation express the motivations to initiate an adaptation process. When these goals are in the user's head, our system cannot directly achieve them, however the system supports it by means of the adaptability facilities included. Although, not every user goal can be supported, including support for some of them actually increases user's confidence in the adaptation capabilities of the system. When the goals are kept by the system, they should be expressed in terms of the context of use characteristics considered during the design of the system and the usability criteria to be preserved. Thus, no goal can be stored that makes use of context of use characteristics that the system is not able to either query or store. The goals for adaptation kept by the system are represented in two different components in our system. On the one hand, these goals are partially expressed as part of the adaptation rules that will finally produce the adaptations required to fulfill those goals. On the other hand, they are expressed as a usability trade-off. This usability trade-off specifies relatively the usability criteria that should be preserved while adapting the user interface. For instance, if in the usability trade-off we specify that continuity should be maximized, the system will always choose those adaptations producing a lesser disruption in continuity, unless the user forces the execution of another adaptation. This usability trade-off is expressed by using  $I^*$  [18] notation.  $I^*$  notation was originally designed to specify system goals in early requirements analysis stage. In section 3.4 how this trade-off is actually applied is described. The multi-agent system supports also those goals remotely-expressed. In this last case, the new remote goals should be expressed in terms of new adaptation rules that can be plugged into the adaptation engine seamlessly. In section 3.3 we elaborate more on how these adaptation rules are designed and specified.

### 3.2 Initiative for Adaptation

In ISATINE multi-agent architecture, the adaptation process can be initiated by either the user, the system or a third-party. The user is allowed to do it by clicking or typing (auditory user interfaces are not supported by now) an option available in every user interface generated by the system. The system can decide that an adaptation is needed by inferring it from the incoming information from the context of use. The agents in charge of detecting context of use changes (*AgentContextPlatform*, *AgentContextEnvironment*, *AgentContextUse* and *AgentDetectContextOfUse*) notice those changes by means of sensors. These sensors can be either software or hardware sensors. Hardware sensors are built or plugged into the hardware platform where the application is running, while software sensors are programmed, and included into the applications supported by the multi-agent system. The designer of the adaptation facilities of every application can define his own software sensors provided that the implementation is compliant with the defined interface for sensors. Most data incoming from sensors is directly linked with a piece of information in the context model, although it is not mandatory. In this architecture, the current task the user is carrying out is also included within the context of use, since it is necessary quite often to guess user needs. To guess the user's current goal, this agent uses the task model created at design time. This task model is a tree where the designer specifies the tasks the user will be able to perform along with some temporal constraints (for instance, a sequential relationship between two tasks). Thus, at any time, taking into account the tree structure and the temporal constraints between the tasks, there will be just a set of possible tasks that the user is allowed to perform through the UI (called enabled tasks set). Therefore, the agent just needs to guess which one among the tasks included in the enabled tasks set is the current task. To help in this problem, the agent uses the usage data collected from interaction, especially taking into account the last components of the UI that have been manipulated and the mapping between the widgets of the user interface and the tasks in the task model.

### 3.3 Specification of Adaptation

Given an initiated adaptation process it is necessary to decide which adaptation will be applied (if any). Whether the user, the system or a third-party has initiated the adaptation process, *AgentAdaptationProcess* Agent proposes the set of adaptation rules that best fit the current context of use. The specification of the set of available adaptations to choose from is built in different ways. The user can demonstrate how he would like the user interface to be adapted. Currently, the user is allowed to demonstrate the colors for each kind of widget, the sizes of the different types of widgets and some kinds of widgets replacements. The agent supports also the specification of rules by computation, although it is currently constraint to the refinement of rules previously defined. However, the main corpus of adaptation rules is provided by the application designer by defining how the system should react to the different situations arising from the interaction.

### 3.4 Application of Adaptation

By default regardless on who was the one that started the adaptation the system will automatically choose which application to apply. If it was either the user or a third

party the one who initiated the adaptation the agent will ask the user or the third party which adaptation between the eligible ones he would like to apply. Otherwise, or if the user or the third-party delegate the task of choosing the adaptation the *AgentAdaptationProcess* agent will choose the most appropriate ones, creating a ranking of rules. To make this selection the rules are evaluated by means of a set of metrics. Afterwards, the agent will try to execute the rules starting from the highest one in the ranking. If the application of the rule does not meet the usability trade-off specified in the goals for user interface adaptation that rule will be discarded and the agent will try to apply the next rule in the ranking following the same process as for the first rule in the ranking. This process is made until no rule is left in the ranking list or until the agent finds that a ranking has been reached in the list too low for that rule to be applied. The agent has been designed so it will not apply any adaptation rule it does not find good enough (unless the user forces its execution). Most of the time is better inaction than applying a rule that is not good enough, producing a degradation of user interface usability and damaging user confidence in the system.

### 3.5 Transition with Adaptation

Making smoother and clearer the transition between the original user interface and the adapted one is very important to avoid confusing the user, and therefore to avoid degrading the user's confidence in the system. ISATINE architecture has been extended with a new agent to support this stage. Although many different kinds of transitions [15] from the original user interface to the adapted one can be imagined, in our case we are just supporting those being general enough to be applied to many different user interfaces, since our transitions are generated at run-time on-the-fly. In section 5.4 an example of how this stage is applied and how our architecture was extended to support it is shown.

### 3.6 Interpretation of Adaptation

One of the issues we found when testing adaptive systems is that sometimes the users were not actually aware that an adaptation had been done, and even what the adaptation was for. The same happens when the user makes an adaptation and the system does not understand why the user wanted to perform that adaptation. To address the first issue transition stage can be used. However, to address the second issue another sub-stage is required to help the user in evaluating what the result of the adaptation was. In this sense, if the user is the one leading the adaptation process, she is allowed to provide a description of what the adaptation was useful for. It allows the system to extract some keywords used to relate this new adaptation with other adaptations stored in adaptation rules pool. On the other hand, if the system leads the adaptation process, it always adds a tooltip to the adapted user interface with a short description of the adaptation made.

### 3.7 Evaluation of Adaptation

An adaptation quality assessment is essential to any good adaptation process, because it should be adaptive itself. The system assesses the adaptation performed by applying heuristics to evaluate a migration cost [13]. This assessment is made at specification

of adaptation stage to create a ranking with the potential applicable rules. However, it is not enough. Since it is impossible to foresee every combination of factors in the context of use, the system can apply a rule not good enough, or simply it can apply a rule the user dislikes. Thus, in the architecture the user can undo any adaptation applied expressing he did not like it. This feedback from the user is injected into the adaptation evaluation mechanism applying a Bayesian approach where rules can improve or worsen their ranking.

## 4 Implementing the Multi-agent Architecture

In this section we will show an overview of the technologies used in the implementation of the multi-agent architecture to support ISATINE framework.

For the multi-agent system implementation we have used JACK<sup>2</sup> [2]. JACK is an agent programming language based on BDI paradigm. This language generates Java language code out of a set of templates that is executed within an execution environment supplied with the language. To maximize platform independence we have wrapped the multi-agent java based system within an HTTP server interface.

The HTTP server interface allows any platform capable of networking using TCP/IP protocol to access the ISATINE adaptation engine. This HTTP server has been implemented as a servlet (server side applet) that runs on top of a TOMCAT server.

Internally, the user interface knowledge gathered at design-time, and later at run-time by means of sensors is stored by using the XML-based user interface description language UsiXML<sup>3</sup>. By means of UsiXML we are able to achieve the specification of a user interface in a representation abstract enough to be presented in different platforms. The model in this language, which is closer to the actual user interface the user interacts with, is the concrete model.

The concrete UI model describes a UI in a manner independent from the platform where it will run on (although it is dependent on modality). Therefore, a renderer is needed so the user can visualize the UI. For this purpose, a renderer for the concrete UI level of UsiXML has been written for several languages. Currently, there is basic rendering support for XUL, Java 2, J2ME, and OpenLaszlo<sup>4</sup> languages, what allows us to run the developed adaptive applications in almost every platform.

By now, we have just implemented sensors for collecting interaction usage data from the client platform and the user. Other data, such as environment physical conditions changes are being simulated via an agent called *AgentStimuliGenerator*. This agent is able to process an input XML file containing a specification of events and their timing, so it can simulate the arrival of changes in the context of use from hardware or software not currently available. This is especially useful during adaptation rules design process.

The real adaptation the user interface undergoes as a result of the application of the adaptation rules is specified by using Attributed Graph Grammars [17]. A detailed description of how this approach is used to generate a user interface can be found in [12]. The transformation engine to execute the transformations associated to the

<sup>2</sup> <http://www.agent-software.com/shared/products/index.html>

<sup>3</sup> <http://www.usixml.org>

<sup>4</sup> <http://www.openlaszlo.org>

adaptations uses the API from AGG (Attributed Graph Grammars) tool<sup>5</sup> to perform the transformations. It provides a programming language enabling the specification of graph grammars and a customizable interpreter enabling graph transformations.

Next, a description of the main processes within the implemented multi-agent system will be described.

### 4.1 Receiving Context Changes from the Sensors and Adapting the UI

When a sensor wants to communicate any change in the context of use it has detected, it opens a communication with a specific URL belonging to the *webAdaptationEngine* servlet. Then the sensor will send the information using the XML format designed for this purpose. This information will be passed to *AgentDispatcherAgent* by the servlet. This agent acts as a mediator between the multi-agent system and the servlet. This agent will detect that it is a context communication act, and it will use its plan *ContextEventGenerator* to send the information to the agent *AgentDetectContextOfUse*. This agent will perform two steps: it processes the XML information received and dispatches each piece of information to the corresponding agent (*AgentContextPlatform*, *AgentContextEnvironment* or *AgentContextUser*). *AgentContextPlatform*, *AgentContextUser* and *AgentContextEnvironment* will update the context model to reflect the changes they have received from *AgentDetectContextOfUse*. Notice that not every piece of information received from *AgentDetectContextOfUse* will produce a change in the context model. The new values received can be equal to the values stored in context model, or the changes in the values might not be significant. When these agents update the context model (represented as agents' beliefs - called *PlatformContextModel*), an event will be generated automatically by the agent's beliefs to indicate to *AgentDetectContextOfUse* that it should throw events of the type *ContextChanged*. These events will be handled by *AgentAdaptationProcess*, which will generate the feasible adaptivity rules (plans) for the new context of use. Finally, a meta-reasoning method will be used to choose the rules to be applied using the adaptation rules selection policy chosen. For the execution of the rules, the agent first gets the up-to-date usiXML version of the running UI to be adapted. Next, it transforms the usiXML specification into a graph representation, and it applies the selected rules using AGG API. Finally, the adapted graph is transformed back to usiXML and the target language at the same time. Thus, the adapted UI is made available to the *AgentDispatcherAgent*, so it can be delivered to the client. This process is illustrated in Fig. 3.

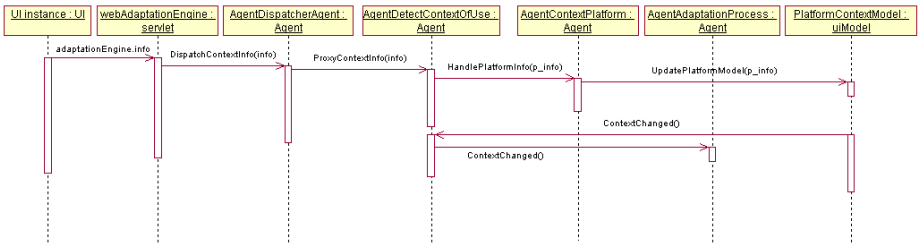


Fig. 3. Receiving Context Changes Info from the Sensors and Adapting the UI

<sup>5</sup> <http://tfs.cs.tu-berlin.de/agg/index.html>

### 4.2 Getting the Adapted User Interface

When any of the sensors communicate information to the adaptation engine, they always receive an answer about whether there is a newly adapted UI ready or not. If there is a new adapted UI ready, it will connect to a specific URL belonging to the *webAdaptationEngine* servlet. Then, *AgentDispatcherAgent* will send the adapted UI to the client. Thus, the user will get an adapted version of the UI that matches the changes in the context of use detected by sensors.

## 5 A Second-Hand Car Selling Case Study

To demonstrate how the architecture supports ISATINE framework for a real example, and the flexibility introduced by designing the architecture as a multi-agent system, a case study is presented next. The case study is based on the main searching facilities form of a real second-hand car selling website. In this form the user is allowed to select the different data required to filter the kind of second-hand car he is searching for. In this sense, the user can provide the car brands he would like the car to be, the maximum amount of money he is willing to spend or the mechanical and physical characteristics of the car. The examples in this section will be presented in growing



Fig. 4. Original main form for the second-hand car-selling example



complexity to illustrate the features starting from the more simple adaptations to the more complex ones. In Fig. 4, a screenshot of the main form for our example is shown. In this case we have used the OpenLaszlo renderer of our architecture to generate the final running user interface (<http://www.usixml.org/index.php?view=page&idpage=120>). On the upper part of the form the user selects the car brands he is interested in, while in the bottom part the user selects the features and constraints for the cars he is searching for.

## 5.1 Adaptability in ISATINE Framework Architecture

One of the main issues in adaptive systems is that if the adaptations are not properly carried out, and the user feels a sense of losing control, the adaptation engine might be rejected. Therefore, it is really important for an adaptation architecture to support the user in taking control of the adaptation engine, because mental model and tastes for different users might differ. In our example the user is querying the system database for the different car brands he is interested in.

To take this decision he is getting additional information from the web pages of different branches. However, the user in his current context of use is a little bit annoyed with the way the interaction is made, because the form takes too much screen display space. By occupying so much space the form is preventing the user from browsing the car brands web sites while selecting the different car features, forcing the user to switch between the second hand car selling application and the car's web-sites. At his moment has a goal on her mind: reducing the displaying space required to interact with the application.

Because of that goal the user wants to adapt the user interface to reduce screen space required by the form of the second hand selling application to be shown. To do so, the user clicks on the "ADAPT" button to express her intention to adapt the user interface. Next, according to the ISATINE framework, the adaptation to be performed needs to be specified. In this case, in order to specify which adaptation is executed, the user selects the adaptation from the available adaptation rule pool. An adaptation rule replaces a set of checkboxes with a multi-select combo box. In this selection activity, the user is supported by providing a meaningful description of the results achieved by applying the adaptation. The application of the adaptation is made by the system. Since it is the user the one who chose to apply the adaptation it will be applied regardless of the ranking of the rule. Because it was the user who led the adaptation process, it is not necessary to help him to interpret the adaptation. The adaptation in this case is considered to be successful unless the user undoes it.

## 5.2 Platform Adaptation in ISATINE Framework Architecture

In the same manner as for the user-initiated adaptation previously described, the architecture supports platform adaptation. In this second example, the user is now using the second hand car selling application in a PDA. In this case, the adaptation is triggered as a result of a goal specified by the designer: "every form displayed in the target platform must show, or at least allow browsing, the data required to carry out the task the form is intended to". The initiative in this case is taken by the system. The system detects a change in the target architecture by means of software sensors

reporting the new characteristics of the platform. To face this situation the system uses the set of rules created by the designers. To reinforce user's trust in the system it shows to the user the possibilities to achieve this platform shifting. In our example, the user selects the application of the same rule as in the previous example, so the checkbox group is replaced with a multi-select combo box. In general, one could imagine to provide the user with different sets of rules applicable for each specific platform.

### 5.3 Context Adaptation in ISATINE Framework Architecture

The user is now at a motor show where many different brands are available. The user is using the application in a PDA equipped with a web cam. The user is making a videoconference to decide which car to buy. So the user stands on the center of the exhibition center and he would like to show to other person each car in the conference, and then by using the second hand car selling application find out if there are any of those cars available and what its characteristics and price are. The user takes the initiative by clicking on the "ADAPT" button. The system now offers to the user the list of possible adaptations to apply. In this case, the user chooses an adaptation called "minimum presentation" that transforms the searching form of the application into a minimal set of widgets to allow querying the site for second hand cars. The adaptation application stage is made in this case in collaboration between the user and the system. The system applies the adaptation to produce a minimal presentation, however, it is the user in charge of positioning the brand new generated presentation in the best place of the screen to support his activities.

### 5.4 Extending ISATINE Framework Architecture to Support Transition Stage

In the previous example, there is an abrupt change between the original user interface in Fig. 5 and the adapted user interface shown in Fig. 6. Thus, a big disruption appears in the change from the original user interface to the adapted one, drastically reducing continuity usability property.

To improve the continuity in the adaptation process a new stage should be included in the adaptation process in charge of making smoother the transition from the original user interface to the adapted one. This stage is one of the extra stages in ISATINE adaptation framework with respect to Dieterich's one. One of the key features of the designed architecture is its extensibility. As long as it was created by using agent's paradigm, it can be easily extended by just adding some extra new agents and rerouting some messages from some agents to other agents. For instance, for the transition state we added a new agent called *AgentTransition*. *AgentAdaptationProcess* was modified so as to send the adapted user interface generated during the application of the adaptation rules to this brand new agent, instead of sending it directly to *AgentDispatcherAgent* to be delivered to the user. *AgentTransition* takes the adapted user interface and it creates smooth transitions depending on the kind of adaptation the user interface has undergone. Right now, this agent is able to highlight the adapted widgets in different ways to guide the user, by changing the background color of some components, changing the panel containing the adapted components or adding word balloons to explain the user what happen during the adaptation. Other techniques such as image animation or morphing could be implemented also. The new adapted user



Fig. 5. Adapted user interface reducing displaying space

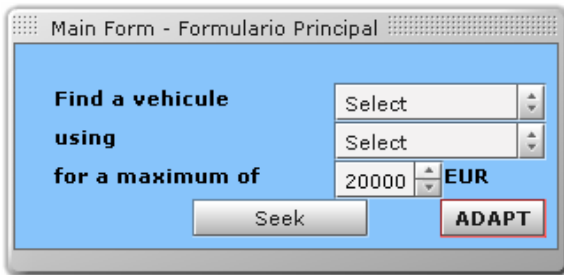


Fig. 6. Second-hand car selling user interface after a context adaptation

interface with the transition effects added is sent to *AgentDispatcherAgent* to be finally delivered to the user. In Fig. 7 a screenshot of the application of the transition stage by *AgentTransition* can be found. A tooltip has been added by *AgentTransition* to remind the user that he can change the view to show some more extra filtering options by clicking on “ADAPT” button. In the same manner that *AgentTransition* agent has been added, other extra agents could be added almost seamlessly to extend the architecture to better attend adaptation requirements.

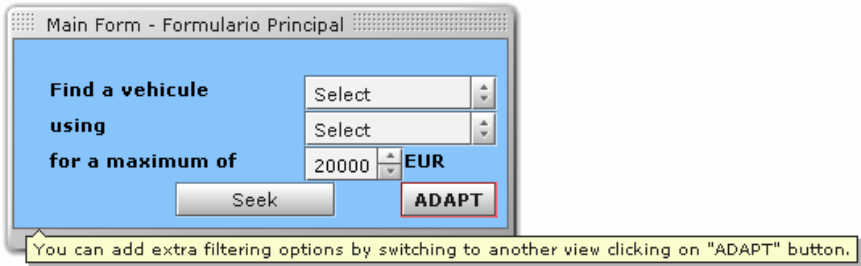


Fig. 7. An example of the output for the transition stage applied to the UI in Fig. 6

## 6 Conclusion and Future Work

This paper was initially motivated by the need for supporting more than just the adaptation execution, which is addressed in Dieterich's taxonomy. This taxonomy has therefore been expanded according to the Seven Stages of Norman's theory of action, thus, leading to the ISATINE framework for UI adaptation. This framework not only decomposes the whole adaptation process into seven corresponding stages, but it also shows how to decompose each stage into sub-stages depending on the collaboration between the entities involved in each stage: the user, the system, an external third party or any collaboration between them. A multi-agent software architecture has been motivated, justified, and defined so as to support the stages of the framework defined. The BDI paradigm has been used for this purpose. A graph transformation system, consisting of steps of graph transformations, has been developed to support the execution of the adaptation on a UI model. A running example has demonstrated how this architecture should be modified in order to accommodate a series of progressively more complex adaptation schemes, thus validating the approach.

A first area for future work consists in exploring other forms of collaboration such as competition (where at least two entities should compete to find out the best solution and a judge entity then keeps the best one assessed according to some criteria) or coopeition (where at least two entities should compete while cooperating at the same time because their knowledge is perhaps complementary). Coopeition is the combination of cooperation and competition. These new forms do not disrupt the multi-agent architecture defined in this paper. A new agent could be incorporated and new relationships defined according to the BDI paradigm could be defined. This greatly simplifies updating the software architecture for accommodating new forms of adaptation, even perhaps the unknown ones.

A second area for future work is to pursue research and development for the agent responsible for conducting the transition. Many techniques proposed in [15] are very promising for this purpose, but they are built-in. The advantage of having the UI model maintained at adaptation time enables us to develop some of these techniques specialized for the UI widgets.

A third area for future work consist of examining how IFIP quality properties (e.g., honesty, observability, browsability [8]) could be preserved by applying this or that adaptation technique and how controllability and traceability of the stages (especially transition and evaluation) could be achieved.

## Acknowledgements

This work is partly supported the Spanish CICYT TIN2004-08000-C03-01 grant and the PBC-03-003 and PAI06-0093-8836 grants from the Junta de Comunidades de Castilla-La Mancha. Also, we gratefully acknowledge the support of the SIMILAR network of excellence (<http://www.similar.cc>), the European research task force creating HCI similar to human-human communication of the Sixth Framework Program.

## References

1. Bratman, M.E.: *Intention, Plans, and Practical Reason*. Harvard University Press, Cambridge (1987)
2. Busetta, P., Ronnquist, R., Hodgson, A., Lucas, A.: Jack intelligent agents - components for intelligent agents in java. AgentLink News Letter (January 1999) White paper accessible, <http://www.agent-software.com>
3. Calvary, G., Coutaz, J., Thevenin, D.: Supporting Context Changes for Plastic User Interfaces: a Process and a Mechanism. In: Blandford, A., Vanderdonckt, J., Gray, P. (eds.) *Interactions sans frontières – Interactions without frontiers*, Proc. of the Joint AFIHM-BCS Conf. on Human-Computer Interaction IHM-HCI 2001, Lille, 10-14 September 2001, vol. I, pp. 349–363. Springer, London (2001)
4. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A Unifying Reference Framework for Multi-Target User Interfaces. *Interacting with Computers* 15(3), 289–308 (2003)
5. Dieterich, H., Malinowski, U., Kühme, T., Schneider-Hufschmidt, M.: State of the Art in Adaptive User Interfaces. In: Schneider-Hufschmidt, M., Khüme, T., Malinowski, U. (eds.) *Adaptive User Interfaces: Principle and Practice*. North Holland, Amsterdam (1993)
6. Eisenstein, J., Puerta, A.: Adaptation in Automated User-Interface Design. In: Proc. of ACM Conf. on Intelligent User Interfaces IUI 2000, New Orleans, 9-12 January 2000, pp. 74–81. ACM Press, New York (2000)
7. Florins, M., Vanderdonckt, J.: Graceful Degradation of User Interfaces as a Design Method for Multiplatform Systems. In: Proc. of ACM Conf. on Intelligent User Interfaces IUI 2004, Funchal, 13-16 January 2004, pp. 140–147. ACM Press, New York (2004)
8. Gram, C., Cockton, G.: *Design Principles for Interactive Software*. Chapman & Hall, London (1996)
9. Kolp, M., Giorgini, P., Mylopoulos, J.: An Organizational Perspective on Multi-agent Architectures. In: Meyer, J.-J.C., Tambe, M. (eds.) *ATAL 2001*. LNCS, vol. 2333, pp. 128–140. Springer, Heidelberg (2002)
10. Horvitz, E.: Principles of Mixed-Initiative User Interfaces. In: Proc. of ACM Conf. on Human Factors in Computing Systems CHI 1999, Pittsburgh, 15-20 May 1999, pp. 159–166. ACM Press, New York (1999)
11. Langley, P.: User Modeling in Adaptive Interfaces. In: Kay, J. (ed.) Proc. of the 7<sup>th</sup> Int. Conf. on User Modeling UM 1999, pp. 367–371. Springer, Berlin (1999)
12. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., López-Jaquero, V.: USIXML: A language supporting multi-path development of user interfaces. In: Bastide, R., Palanque, P., Roth, J. (eds.) *DSV-IS 2004 and EHCI 2004*. LNCS, vol. 3425, pp. 200–220. Springer, Heidelberg (2005)

13. López-Jaquero, V.: Adaptive User Interfaces Based on Models and Software Agents, Ph.D. thesis, University of Castilla-La Mancha, Albacete, Spain (in Spanish) (October 14, 2005), <http://www.isys.ucl.ac.be/bchi/publications/Ph.D.Theses/Lopez-PhD2005.pdf>
14. Norman, D.A.: Cognitive Engineering. In: Norman, D.A., Draper, S.W. (eds.) User Centered System Design, pp. 31–61. Lawrence Erlbaum Associates, Hillsdale (1986)
15. Rogers, S., Iba, W.: Adaptive User Interfaces: Papers from the 2000 AAAI Symposium, Technical Report SS-00-01. AAAI Press, Menlo Park (March 2000)
16. Schlienger, C., Dragicevic, P., Ollagnon, C., Chatty, S.: Les transitions visuelles différenciées: principes et applications. In: Proc. of the 18th Int. Conf. on Association Franco-phone d'Interaction Homme-Machine IHM 2006, Montreal, 18-21 April 2006. ACM Int. Conf. Proc. Series, vol. 133, pp. 59–66. ACM Press, New York (2006)
17. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
18. Yu, E.: Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering. In: Proc. of the 3rd IEEE Int. Symp. on Requirements Engineering RE 1997, Washington, 6-8 January 1997, pp. 226–235. IEEE Computer Society Press, Los Alamitos (1997)
19. Wooldridge, M., Jennings, N.R.: Agent Theories, Architectures, and Languages: A Survey. In: Proc. of ECAI-Workshop on Agent Theories, Architectures and Languages, Amsterdam, pp. 1–32 (1994)

## Questions

### *Philippe Palanque:*

*Question: According to the fact that you are using a multi-agent technology (that is by definition continuously evolving), how can you assess the results and, for instance, guarantee that the adaptation that was a success once, will be a success again?*

Answer: this is a real problem and the definition of metrics on a multi-agent platform is still a research topic. Now that the platform is ready and that the architecture is defined this is one of the things we will be working on.

# Towards a Universal Toolkit Model for Structures

Prasun Dewan

Department of Computer Science, University of North Carolina  
Chapel Hill, NC 27516, U.S.A.  
dewan@cs.unc.edu

**Abstract.** Model-based toolkit widgets have the potential for (i) increasing *automation* and (ii) making it easy to *substitute* a user-interface with another one. Current toolkits, however, have focused only on the automation benefit as they do not allow different kinds of widgets to share a common model. Inspired by programming languages, operating systems and database systems that support a single data structure, we present here an interface that can serve as a model for not only the homogeneous model-based structured-widgets identified so far – tables and trees – but also several heterogeneous structured-widgets such as forms, tabbed panes, and multi-level browsers. We identify an architecture that allows this model to be added to an existing toolkit by automatically creating adapters between it and existing widget-specific models. We present several full examples to illustrate how such a model can increase both the automation and substitutability of the toolkit. We show that our approach retains model purity and, in comparison to current toolkits, does not increase the effort to create existing model-aware widgets.

**Keywords:** Tree, table, form, tab, browser, hashtable, vector, sequence, toolkit, model view controller, user interface management system.

## 1 Introduction

User-interface toolkits strongly influence the nature of a user-interface and its implementation. Programmers tend to incorporate components into a user-interface that are easy to implement. For example, programmers use the buttons directly supported by a toolkit rather than define their own buttons using the underlying graphics and windows package. Moreover, the implementation of the user-interface typically follows the architecture directly supported by the toolkit. For example, in the early versions of the Java AWT toolkit, programmers attached semantics to widgets by creating subclasses of these widgets that trapped appropriate events such as button presses. As the newer version of AWT supports delegation, programmers now associate callbacks with these widgets.

One of the major recent advances in toolkits is support for model-aware widgets, that is, widgets that understand the interface of the semantic or model object being manipulated by them. Model-aware widgets have the potential for (i) increasing *automation* and (ii) making it easy to *substitute* a user-interface with another one. Current toolkits, however, have focused only on the automation benefit as they do not allow different kinds of widgets to share a common model. For example, in Java's

Swing toolkit, the `JTable` and `JTree` model-aware widgets understand different kinds of models. As a result, it is not possible to display the model of a `JTable` widget as a tree, and vice versa.

Therefore a data structure that serves as a universal model for different widgets is an attractive idea. It is not possible to develop such a model for all possible widgets as some widget models assume fundamentally different semantics. For example, the model of a slider must be a numeric value and not, for example, a string or a list. In this paper, we show that is possible, however, to develop a universal model for all existing structured model-unaware widgets and several new structured components such as browsers for which no appropriate model interface has been defined so far. Thus, such a universal structured model increases both the automation and substitutability of the toolkit. It increases automation as it directly supports user-interface components such as browsers that have to be manually composed today. It increases substitutability as it allows the model to be displayed using any of the existing and new model-aware structured-widgets.

In the rest of the paper, we expand on this idea. We first show the relationship between the MVC (Model-View-Controller) architecture [1] and model-aware widgets. Once this relationship is understood, then the substitutability limitation of current toolkits becomes apparent. We then present requirements of a universal structured-model. Next we take a top-down approach to identifying such a model based on the work done in programming languages, operating systems and database systems that support a single data structure. We then do a bottom-up analysis of this model by exploring how it could be attached to existing and new structured user-interface components, extending it as necessary. We end with conclusions and directions for future work.

## 2 MVC and Toolkit Widgets

The MVC framework, as presented in [1], requires the semantics of a user-interface to be encapsulated in a model, the input processing to be performed by one or more controllers, and the display to be defined by one or more views. In response to an input command, a controller executes a method to write the state of the model, which sends notifications to the views, which, in turn, read appropriate model state, and update the display.

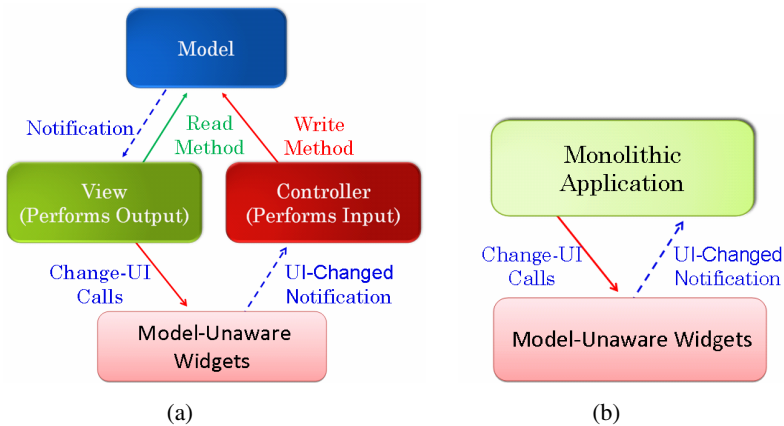
One issue not explicitly addressed by MVC, or any other paper with which we are familiar, is: what is the relationship between MVC and toolkits? The architecture could be implemented (i) from scratch, without using a toolkit, (ii) using model-unaware widgets, or (iii) using model-aware widgets. As (i) does not inform toolkit design – the focus of this paper – let us ignore this approach. To contrast (ii) and (iii), we must precisely distinguish between model-aware and model-unaware widgets.

A model-unaware widget talks to its client in a syntax-centric language. It defines calls allowing the widget client to set its state in display-specific terms, and sends notifications to the client informing it about changes to the state, again in display-specific terms. For example, a model-unaware text-box displaying a Boolean value talks to its client in terms of the text it displays. It defines calls that allow the client to set the text and sends notifications informing the client about changes to the text.



A model-aware widget, on the other hand, talks to its clients in a semantics-centric language. It receives notifications regarding changes to the client state in model-based terms, and converts these changes to appropriate changes to the display. When the display changes, the widget calls methods in the client to directly update its state. For example, a model-aware text-box displaying a Boolean value would talk to its client in terms of the Boolean it displays. When the user edits the string, it directly updates the Boolean, and conversely, it responds to a notification by automatically converting the Boolean to a string.

Given model-unaware widgets, Figure 1(a) shows how the user-interface *should* be implemented and Figure 1(b) shows how it *can* be implemented. In Figure 1(a), the view translates a model notification into an operation on the widget; and the controller translates a widget notification to a call in the model. In Figure 1(b), the widget client is a monolithic application that performs semantics, input and output tasks. Often, programmers follow the architecture directly supported by a toolkit, which in this case means that the architecture shown in Figure 1(b) is used, resulting in a spaghetti of callbacks [2] mixed with semantics.



**Fig. 1.** Using model-unaware widgets with (a) and without (b) MVC

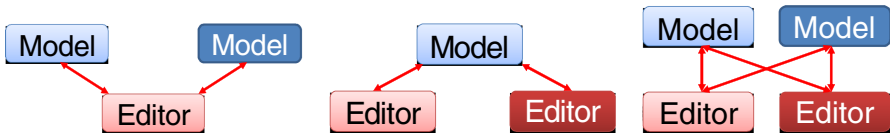
This problem does not, of course, occur with model-aware widgets. These widgets do not directly support the MVC architecture. Instead, they support a model-editor architecture (called subject-view in [3]), in which the editor combines the functionality of a view and controller, receiving notifications from the model and calling both read and write methods in the model. A model-aware widget is essentially an editor automatically implemented by the toolkit that is based on some model interface. As it is based on an interface rather than a class, it can be reused for any model class that implements the interface, as shown in Figure 2(a). It is this model substitutability that increases the automation of the toolkit – for all models displayed using the widget, no UI code needs to be written.

Model substitutability was not an advertised advantage of the original MVC framework, which, as mentioned earlier, did not address toolkits or automation. This substitutability is the dual of the UI/editor substitutability for which the MVC

architecture was actually created, which is shown in Figure 2(b). Given a model, it is possible to attach multiple editors to it, concurrently or at different times. Attaching a new editor to a model does not require changes to the model or other editors – the only requirement is that the editor understand the model interface. Thus, given a model displayed as a bar-chart, adding an editor that displays it as a pie-chart does not require changes to the model or the existing editor.

While toolkits have made an important advance to the MVC architecture by using it for automation, as designed currently, they have done so by sacrificing the original advantage of the architecture. The reason is that different editors supported by a toolkit assume different model interfaces. For example, the tree and table widgets in Swing assume different models. As a result, it is not possible to display the same model as a tree and/or a table. It is possible to display a tree or table model using a programmer-defined user-interface, but that involves sacrificing automation. The Windows/Forms toolkit has a similar problem. As our implementation is based on Java, we shall focus only on the Java Swing toolkit in the remainder of the paper.

What is needed, then, is a technique that combines both kinds of substitutabilities, which is shown in Figure 2(c). Here, a toolkit-provided editor can be attached to instances of multiple model classes. In addition, a model can be attached to instances of multiple editor classes. In the next section, we describe what this means in more depth.



(a) Toolkit Model Substitutability (b) MVC UI Substitutability (c) Model/UI Substitutability

**Fig. 2.** Three forms of substitutability possible with model-aware widgets

### 3 Requirements

To remove the limitations of previous work mentioned above, we need a new toolkit design that meets the following requirements:

1. *Reduced model set:* The current set of models should be replaced with a smaller set of models.
2. *Same or increased model-aware widget sets:* The set of model-aware widgets automatically supported by the toolkit should not be reduced.
3. *Same or decreased programming effort:* It should not be harder to create models and bind them to existing editors.
4. *Model purity:* The models must have only semantic state.

It is important to meet all of these requirements. It is easy to meet the first requirement by, for instance, simply eliminating the table model from Swing. However, this approach does not meet the second requirement, as the set of model-aware widgets is

also reduced. It is easy to meet both requirements by requiring a model to implement the interfaces of multiple existing model-aware widgets. For instance, combining the model interfaces defined by the tree and table widgets reduces the set of model interfaces, but requires programmers using the interface to implement both sets of methods, instead of only one of the sets, which does not meet the third requirement. Existing “models” in toolkits sometimes have user-interface information. For example, the `JTable` model indicates the label to be used as a column name. Therefore, we have put the fourth requirement to ensure the purity of models. It is possible to meet the first three requirements to different degrees depending on the extent to which the (1) model set is reduced, (2) set of model-aware widgets is increased, and (3) programming effort is changed. In the following sections, we present an approach that meets these requirements and evaluate it based on the above metrics.

## 4 Top-Down Identification of a Universal Structured Model

The ideal approach to meeting the above requirements is to define a universal model for all widgets. However, as mentioned before, it is not possible to develop such a model as there are widget models with fundamentally different semantics. Thus, we must set our sights lower and aim simply for a reduced model set rather than a single model.

There are well known techniques for reducing the model set in existing toolkits. Previous work has shown how a model can be mapped to multiple *unstructured-widgets* [4, 5], that is, widgets displaying a single editable atomic value. In particular, a discrete number can be mapped to a slider or textbox, an enumeration can be mapped to combobox or textbox, and a Boolean can be mapped to a textbox, combobox, or checkbox. These techniques are gradually being implemented in existing toolkits. However, there has been no work for mapping a model to multiple *structured-widgets* such as tables and trees, which display composite (non-atomic) values. Therefore, we will focus only on such widgets in this paper.

Can we define a single universal model for all model-aware structured-widgets supported so far? If so, can it also be bound to other user-interface components that are not automatically supported by existing toolkits? These are the two questions we address in this paper. While they have not been addressed before in the user-interface arena, analogous questions have been posed in other fields such as database management systems, operating systems, programming languages, and integrated systems.

Research in database management systems has tried to determine if a single data structure can be used to store all data that must be searched. A practical answer has been the relational model [6]. Similarly, research in operating systems has tried to determine if a single data structure can be used to store all persistent data, and a practical answer has been the Unix “file”, which models devices, sockets, text files, binary files, and directories. Research in programming language has tried to answer an even more complex question: can a single structured object be used for all computation? The answer in Lisp (and later functional languages such as ML) is an ordered list, and in Snobol (and later string processing languages such as Python) a

hashtable. Designers of EZ [7] have proposed using a nested hashtable as the only structured object in a programming language that is integrated with the underlying operating system. For example, a directory is simply a persistent table, and changing to sub directory, *sd*, corresponds to looking up the table value associated with key *sd*.

Of course, the reduced abstraction set is not without limitations. Therefore, object-oriented database management systems have been proposed as alternatives to traditional relational systems; IBM has supported structured files in its operating system (an idea that was supposed to be extended by the Longhorn Microsoft operating system); and object-oriented languages are preferred today to Lisp and Snobol. It is for this reason that we have added the other three requirements in addition to the requirement of a reduced model set. If we meet all four requirements, we improve the state of the art without introducing any limitations.

We mention the research in other fields to motivate a top-down search for a universal structured model that is based on data structures that have been found to be sufficient for defining a variety of semantic state, which is the kind of state managed by a model. The alternative is a bottom-up approach in which we try to generalize models of existing structured-widgets. As the nature of the models should be independent of the nature of user-interfaces, the result of the top-down approach seems more likely to last in the long-run. In particular, as it is not based on specific user-interfaces, it should make it possible to automatically support new kinds of structured-widgets. On the other hand, this approach does not distinguish between displayed and internal semantic state. The second approach can identify aspects of displayed semantic state not captured by existing display-agnostic data models.

For these reasons, we take an approach in which we: (1) first use the top-down approach of creating an interface that models the universal semantics structures proposed in other fields; (2) and then take the bottom-up approach of generalizing this interface to connect it to existing model-aware widgets.

The first step above requires an interface that combines elements of relations, nested hastables, and lists. A relation is simply a set of tuples, where each tuple is a record. Thus, we can reduce the above goal to supporting records, un-ordered sets, ordered lists, and nested hashtables.

As we are developing a Java-based tool, let us start with an interface containing a subset of the methods implemented by the Java Hashtable class:

```
public interface UniversalTable <KeyType, ElementType>{
  public Object put(KeyType key, ElementType value);
  public Object get(KeyType key);
  public Object remove(KeyType key);
  public Enumeration elements();
  public Enumeration keys();
}
```

This interface completely models a hashtable because it has methods to (a) associate an element with a key, (b) determine the element associated with a key, and (c) remove a key along with the associated element. The interface is parameterized by the types of the keys and elements. As the element types can themselves be tables, this interface also models nested hashtables of the kind supported by EZ. The last two methods in the interface seem to have been added by Java for purely convenience reasons – they make it possible to treat a hashtable as a pair of collections accessed

using CLU-like iterators [8]. However, as we show below, they also allow the interface to model records, ordered lists, and sets.

A record is simply a table with a fixed number of keys. Thus, a record implementation of this interface simply initializes the table with the fixed number of keys and does not let keys to be added or deleted. This is illustrated in the following class, which defines a subset of the contents of an email message-header:

```
//simulating a record whose fields are not ordered
public class AMessage implements UniversalTable<String, String> {
    Hashtable<String, String> contents = new Hashtable();
    public final static String SUBJECT = "Subject";
    public final static String SENDER = "Sender";
    public final static String DATE = "Date";
    public AMessage (String theSubject, String theSender, String theDate){
        put(SENDER, theSender);
        put(SUBJECT, theSubject);
        put(DATE, theDate);}
    public Enumeration keys() {return contents.keys();}
    public Enumeration elements() {return contents.elements();}
    public String get (String key) {return contents.get(key);}
    public Object put(String key, String val) {
        if (contents.get(key) != null) return contents.put(key, val);
        else return null; // record keys are fixed
    }
    public String remove (String key) {return null;}
}
```

The above class defines a record consisting of three fields named “Subject”, “Sender” and “Date”, and defines a constructor that initializes the value of these fields.

The two iterator-based methods can be used to model an ordered list. The return type, Enumeration, of these methods, is given below:

```
public interface Enumeration{
    public boolean hasMoreElements();
    public Object nextElement();
}
```

As we see above, this type defines an order on the elements to which it provides access. Thus, the `keys()` and `elements()` methods of our universal table can be used to define an order on the keys and elements, respectively, in the table. The class, `AMessageList`, given on the next page, illustrates this concept. Like the previous example, this class stores the mapping between keys and elements in an instance of the Java `Hashtable` class. However, unlike the previous class, it does not return these values in the order returned by the underlying `Hashtable`. Instead, it uses two vectors, one for keys and another for elements, to keep track of the order in which these values are added to the table, and returns them in this order. If a key is associated with a new element, then the new element takes the position of the old element associated with the key. When a key is removed, the key and the associated element are removed from the vectors storing them. As this code is somewhat complicated, we have incorporated it in a generic list class that is parameterized by the key and element type and implements `UniversalTable`. As a client may wish to insert rather than append components, we add another `put` method to the universal

table interface that takes the position of the key and element pair as an additional argument:

```
public Object put(KeyType key, ElementType value, int pos);
```

A set can be more simply modeled by overriding the put method to not replace the value associated with a key. Thus, we have been able to use a single interface to simulate four important structures: nested hashtables, records, ordered lists, and sets. Interestingly, we have done so by using a subset of the methods of an existing class – the Java Hashtable.

```
// simulating an ordered list
public class AMessageList
    implements UniversalTable<String, AMessage>{
    Hashtable<String, AMessage> contents = new Hashtable();
    Vector<String> orderedKeys = new Vector();
    Vector orderedElements = new Vector();

    public Enumeration keys() {
        return orderedKeys.elements();
    }
    public Enumeration elements() {
        return orderedElements.elements();
    }
    public AMessage get (String key) {
        return contents.get(key);
    }
    public AMessage put (String key, AMessage value) {
        AMessage oldElement = contents.get(key);
        AMessage retVal = contents.put(key, value);
        if (oldElement == null) {
            orderedKeys.addElement(key);
            orderedElements.addElement(value);
        } else {
            int keyIndex = orderedKeys.indexOf(key);
            orderedElements.setElementAt(value, keyIndex);
        }

        return retVal;
    }
    public AMessage remove (String key) {
        int keyIndex = orderedKeys.indexOf(key);
        if (keyIndex != - 1) {
            orderedKeys.remove(keyIndex);
            orderedElements.remove(keyIndex);
        }
        return contents.remove(key);
    }
}
```

Finally, to make our universal table a model that can notify editors/views and other observers, we add the following methods to UniversalTable:

```
public void addUniversalTableListener(UniversalTableListener l);
public void removeUniversalTableListener(UniversalListener l);
```

A listener of the table is informed about keys being put and removed:

```
public interface UniversalTableListener {
    public void keyPut(Object key, Object value);
    public void keyRemoved(Object key);
}
```

## 5 Binding Universal Model to Structured-Widgets

Let us now take the bottom-up approach of determining if instances of the universal table can serve as models of two existing Swing structured model-aware widgets: `JTree` and `JTable`?

Let us first consider `JTree`, which has several requirements:

1. Its model must be decomposable into a tree,
2. Both the internal and leaf nodes should have data items associated with them.
3. The node data items should be editable, that is, it should be possible to add and remove children of composite tree nodes, and modify the data items of all nodes.

To meet requirement 1, we must be able to decompose an instance of a universal table into component objects. The instance can be decomposed into its (a) key objects, (b) element objects, and (c) key and element objects (Figure 3).

We provide a special call that can be used by the programmer to make this choice for a specific application class, as shown below:

```
ObjectEditor.setChildren(AMessageList.class, ELEMENTS_ONLY);
ObjectEditor.setChildren(AMessageList.class, KEYS_ONLY);
ObjectEditor.setChildren(AMessageList.class, KEYS_AND_ELEMENTS);
```

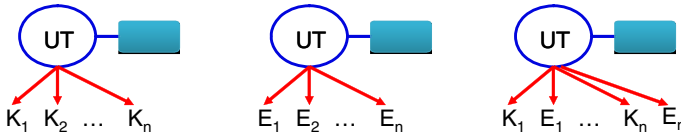


Fig. 3. Three alternative approaches to decomposing a universal table

These calls tell the toolkit to decompose instances of `AMessageList` into its elements, keys, or keys and elements. If a key or element is also a universal table, then it too can be decomposed in any of the three ways. In the case of `AMessageList`, each element is an instance of `AMessage`, which implements `UniversalTable`. Therefore, it too can be decomposed into sub-objects. Figure 4 shows the decompositions defined by the following calls:

```
ObjectEditor.setChildren(AMessageList.class, ELEMENTS_ONLY);
ObjectEditor.setChildren(AMessage.class, ELEMENTS_ONLY);
ObjectEditor.setChildren(AFolder.class, KEYS_ONLY);
```

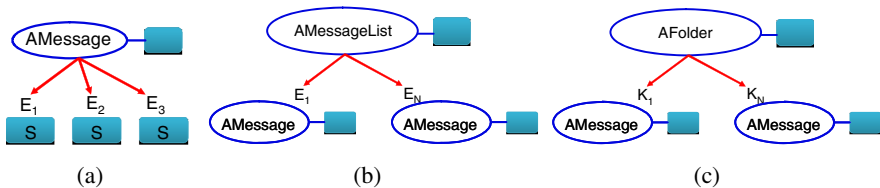


Fig. 4. Decomposing three example universal tables into components

Here, `AFolder` is a universal table with keys of type `AMessage` and elements of type `String`, mapping message-headers to the corresponding message texts:

```
public class Folder implements UniversalTable<AMessage, String>
```

Thus, `AFolder` and `AMessageList` are duals of each other in that the key type of one is the element type of the other. In Figure 4, an empty box is attached to an internal node to denote its data item, and a box with label `S` is used to denote a leaf node of type `String`.

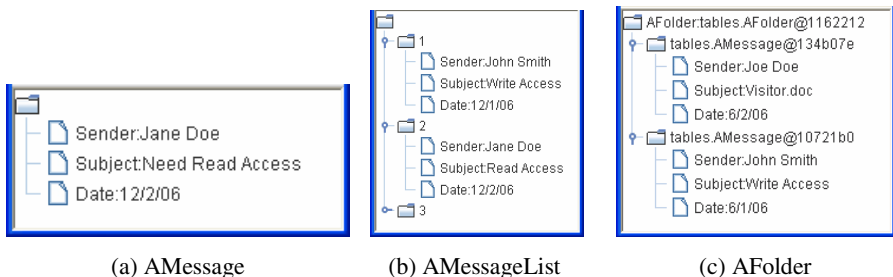
By default, a table is decomposed into its elements. A programmer can define the default decomposition for all universal tables by using the following call:

```
ObjectEditor.setDefaultHashtableChildren(KEYS_ONLY);
```

Let us now consider the second requirement of associating the tree nodes with data items. We could simply use the approach used by `JTree` of assuming that the `toString()` method of a tree node defines the value. However, to support form user-interfaces, we use a more complex approach described by the following routines:

```
Object getTreeDataItem(node) {
    if (getLabel() != "")
        if (node is leaf)
            return getLabel(node) + ":" + node.toString()
        else // node is element
            return getLabel(node)
    else // label = ""
        return node
}
String getLabel (node) {
    if node is labelled and label is defined
        return label
    else if (node is labelled and node is element) // label not defined for element
        return getTreeDataItem( key associated with element).toString()
    else // label not defined for key
        return ""
}
```

This algorithm is motivated and illustrated by the tree displays of `AMessage`, `AMessageList`, and `AFolder` shown in Figure 5.



**Fig. 5.** Associating model items with tree nodes



In Figure 5(c), none of the classes has overridden the `toString()` method, while in Figures 5(a) and 5(b), `AMessage` and `AMessageList` have overridden this method to return the null string. In all cases, the labeled attribute is true and the default label is the null string. In Figure 5(b), the data items associated with the `AMessage` elements are their keys: “1”, “2” and “3”. In Figure 5(c), the data items associated with the `AMessage` keys are the values returned by their `toString()` methods. `ObjectEditor` provides routines to set the values of the labeled and label attributes. For example, the following call says that, by default, the value of the labeled attribute is false:

```
ObjectEditor.setDefaultLabelled(false);
```

Similarly, the following call says that the value of the labeled attribute for instances of type `AMessage` is true:

```
ObjectEditor.setLabelled(AMessage, true);
```

The exact algorithm for determining the data item of a node can be expected to evolve – what is important here is that it depends on a programmer-specified label and takes into account whether the node is a key, element, leaf, or composite node.

Now consider the requirement of allowing nodes to be editable. Inspired by Java’s `MutableTreeNode` class, we add the following method to `UniversalTable` to allow its data items to be changed:

```
public void setUserObject(Object newValue);
```

The following code shows what happens when a node’s data item is changed:

```
Object edit(node, newValue) {
    if node is composite
        node.setUserObject(newValue)
    else if node is key // leaf key
        parent_of_node.put(newValue, parent_of_node.get(old key));
        parent_of_node.remove(oldKey);
    else // leaf element
        parent_of_node.put(key_of_node, newValue);
```

Editing the data item of a composite node results in the `setUserObject()` method to be called on the node with the new value. Editing the data item of a leaf element results in the key associated with the element to be bound to the new value. Thus, in Figure 5(a), changing “Jane Doe” to “Jane M. Doe” results in the “Sender” key to be associated with “Jane M. Doe”. Editing the data item of a leaf key results in the element associated with the old key to be associated with the new key. Thus, in Figure 5(b), changing the key “1” to “One” associates the first message with “One” instead of “1”.

The following code shows what happens when a new node is inserted into a composite node at position index:

```
insert(parent, child, index)
    if (keysOnly(parent))
        parent.put(child, node.defaultElement(child), index);
    else if (elementsOnly(parent))
        parent.put(node.defaultKey(child), child, index);
```

```

else if (keysAndElements(parent))
    if (isKey(parent, child, index) or index == size ) // inserting before key or at end
        parent.put (child, node.defaultElement(child), index/2);
    else // inserting before element
        parent.put (node.defaultKey(child), child, (index - 1)/2);

```

Based on the position of the inserted element and how the parent of the inserted element has been decomposed, the code determines if a key or element is to be inserted, and calls methods in the parent to determine the default key or element to serve as the new child. The `isKey()` method determines if the new node is a key based on the insertion position. The code assumes two new methods in the universal table interface:

```

public KeyType defaultKey(ElementType element);
public ElementType defaultValue(KeyType key);

```

These two methods are needed only because the universal table constrains the types of its key and elements. If it were to accept any object as a key or element, the toolkit could simply create a new object as a default key or object:

```

new Object()

```

The operation to remove a node is simpler.

```

remove (parent, child)
    if isKey (child) parent.remove (child) else parent.remove (key of child)

```

Finally, `ObjectEditor` provides a way to specify that a universal table should be displayed as a tree:

```

edit (UniversalTable model, JTree treeWidget);

```

This operation displays the model in `treeWidget`. Here, the programmer explicitly creates the tree widget, setting its parameters such as preferred size as desired. We also provide the operation:

```

treeEdit (UniversalTable model)

```

which creates the tree widget with default parameters. Sometimes a whole class of objects must be displayed using a particular kind of widget, so the following operation is also provided:

```

setWidget (Class universalTableClass, Class
widgetClass)

```

This call tells the toolkit to always display an instance of `universalTableClass` using an instance of `widgetClass`.

Thus, we have met all of the requirements imposed on us by the Swing tree widget. Let us consider now the Swing table widget. This widget needs the following information: (1) a two dimensional array of elements to be displayed; (2) the most specific class of the elements of each column; (3) the names of the columns; and (4) whether an element is editable.

The first requirement can be met by a non-nested or nested universal table. A one-level universal table (that is a universal table whose children are leaf elements) is considered a table with a single row or column based on whether its `alignment` is `horizontal` or `vertical`, respectively. A two-level universal table (that is a table

whose children are one-level tables) decomposed as keys only (elements only) is straightforwardly mapped to a table in which a row is created for each key (element) of the table consisting of the components of the key (element). A universal table decomposed as keys and elements whose keys are leaf values and elements are 1-level universal tables is decomposed into a table in which a row is created for each key of the object consisting of the key and children of the corresponding element. Currently, we do not map other universal tables to table widgets. The second requirement above is met by returning the class of the default element/key depending on how the table has been decomposed into children. As column names can sometimes be automatically derived from the semantics of the model, but should not be defined explicitly by the model, we use the following algorithm for determining them:

```
getColumnName(root, columnNum)
    if numRows(root) > 0 return firstRow(root).column(columnNum).getLabel();
    else return "";
```

If the matrix is not empty, it then uses the `getLabel()` operation defined earlier to return the label of a particular column in the first row. Recall that the operation returns a value based on the key of an element and the label attribute of the element. To meet the last requirement of `JTable`, we provide the following methods inspired by the Swing `JTableModel` class:

```
public boolean isEditableKey(KeyType key);
public boolean isEditableElement(ElementType element);
public boolean isEditableUserObject();
```

Figure 6(a) illustrates our schemes for meeting the requirements above using an instance of a `AMessageList`. Here, `AMessageList` is decomposed into keys and elements, `AMessage` is decomposed into elements, the keys of `AMessageList` are not labeled, and the elements of `AMessage` are labeled but have no explicit label set by the programmer. As a result, each row consists of the atomic String key, and the atomic elements of `AMessage`; and the keys of the elements of `AMessage` are used as column names but not displayed in each row. As in the tree widget case, we provide routines to bind a table widget to a model.

	Sender	Subject	Date
1	John Smith	Write Access	12/1/06
2	Jane Doe	Read Access	12/2/06
3	Mary Smith	R/W Access	12/2/06

<b>Sender</b>	<input type="text" value="Jane Doe"/>
<b>Subject</b>	<input type="text" value="Need Read Access"/>
<b>Date</b>	<input type="text" value="12/2/06"/>

Fig. 6. Table and Form Displays

The fact that a universal table models a record implies that we can also support forms, as these have been previously created automatically from database records [9]. However, database records (tuples) are flat. As universal tables are nested, we can create hierarchical forms. In fact, we can embed tables and trees in forms. Figure 7

shows a table embedded in a form. Here, we assume `AFolder` is decomposed into its keys, and `AMessage` is decomposed into keys. The algorithm for creating a form is:

```
displayForm (node) {
  panel = new Panel
  setLabel (panel, getLabel(node)) // can put label in the border, add a label widget, ....
  for each child of node
    childPanel = display (child)
    add (panel, childPanel)
  return panel
```

The operation `display(node)` returns a component based on the widget associated with the type of `node`. For a universal table, the widget is a form, tree, tabbed pane, or table. For an atomic type, it is an atomic widget such as a slider, combo-box, text-box or checkbox. The algorithm leaves the layout of children in a parent panel as implementation defined. In [10], we define a parameterized scheme for arranging form items in a grid.

Tabbed panes are similarly implemented:

```
displayTabbedPane (node) {
  tabbedPane = new tabbed pane;
  for each child of node
    childPanel = display (child)
    add(tabbedPane, childPanel, getLabel(child))
  return panel
```

Figure 7(b) shows the tabbed display for folder displayed in 7(a).

Universal tables are ideal for creating browsers, which are common-place, but have not been automatically supported by any user-interface tool. To create a browser, the `ObjectEditor` provides the following call:

```
edit (UniversalTable model, Container[] containers);
```

If the array, `containers`, is of size  $n$ , this call creates an  $n$ -level browser. A browser always decomposes a universal table into its keys. The top-level model is displayed in `container[0]`. When a key is selected in `container[i]`, it displays the associated element in `container[i+1]`, where  $0 \leq i < n$ . Figure 8 illustrates this scheme. Here, a three-level browser has been requested, and the top-level model is an instance of the class `AnAccount`, whose keys are strings and elements are of type `AFolder`:

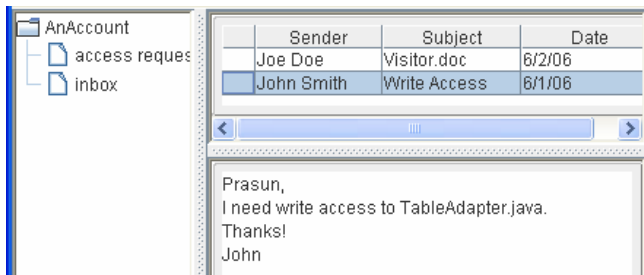
```
public class AnAccount implements UniversalTable <String, AFolder>
```

`AnAccount` has been bound to a tree widget, and `AFolder` to a table widget. The container array passed to the `edit` routine above consists of the left, top-right, and bottom-right windows, in that order. The toolkit shows the two `String` keys of the top-level model in the first container. Selecting the first `String` key in this container results in the associated folder element being displayed in the second container. Selecting one of the `AMessage` keys of this folder results in the associated `String` element to be displayed in the third container.



**Fig. 7.** Nested Form and Tabbed Panes

Like tables and trees, tabs, forms and browsers are structured model-aware widgets in that they are composed of components that are bound to children of the model. However, in the former, the nature of the automatically generated child components is fixed by the designer of the widget, while this is not the case in the latter. For example, a browser pane can consist of a table, tree, form, textbox or any other component to which a model is bound. The algorithms we have given above are independent of the exact widget bound to a model child. Support for such heterogeneous model-aware widget-structures is a fundamentally new direction for toolkits, but is consistent with the notion of supporting model-aware widgets. Some existing structured-widgets such as `JTable` do allow programmer-defined widgets to be embedded in a widget-structure, but the embedded widgets are not themselves model-aware widgets automatically supported by the toolkit. For example, a `JTable` or `JTree` cannot be automatically embedded in a `JTable`.



**Fig. 8.** A Three-Level Browser

Thus, we have described an approach that allows a single model to be bound to both existing and new user-interface components. There are many ways of implementing it. From a practical point of view, it should be possible to layer it on top of an existing toolkit without requiring re-implementation of existing model-aware widgets. This, in turn, requires adapters between the universal table models and the existing toolkit models. We could require a separate adapter for each existing toolkit model. For example, we could define separate adapters for tree and table models. However, we take a more complicated and perhaps less modular approach in which we define a universal adapter that can support both existing and new widgets. This adapter understands the

universal table interface, and implements the interfaces of the models of the Swing tree and table widget. This approach allows us to create a single adapter tree that can be dynamically bound to multiple widgets concurrently (Figure 9). The following algorithm describes the nature of the model structure, and how it is created:

```

UniversalAdapter createUniversalAdapter (Object model)
if (model is UniversalTable)
    UniversalAdapter modelAdapter = new StructureAdapter(model);
    for each key, element of model
        UniversalAdapter keyAdapter = createUniversalAdapter(key)
        UniversalAdapter elementAdapter = createUniversalAdapter(element)
        keyAdapter.setParent(modelAdapter);
        elementAdapter.setParent(modelAdapter);
        modelAdapter.setKeyElement(keyAdapter, elementAdapter);
else return new LeafAdapter(model);
    
```

Unlike the model structure, the adapter structure includes back links from children to parents, which are required by the model of the Swing tree widget. These links also allow us to find the key associated with an element, which is needed to label the latter. Programmers can determine the universal adapter bound to a model, and retrieve information kept by it such as the parent adapter, children, and currently bound widget. Thus, they don't have to manually keep such book-keeping information.

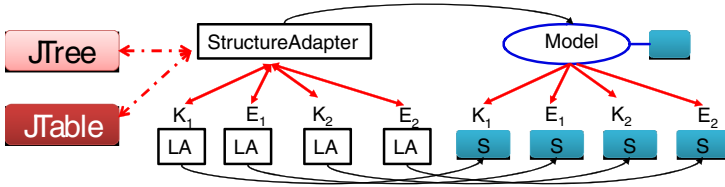


Fig. 9. Implementation architecture (LA = LeafAdapter)

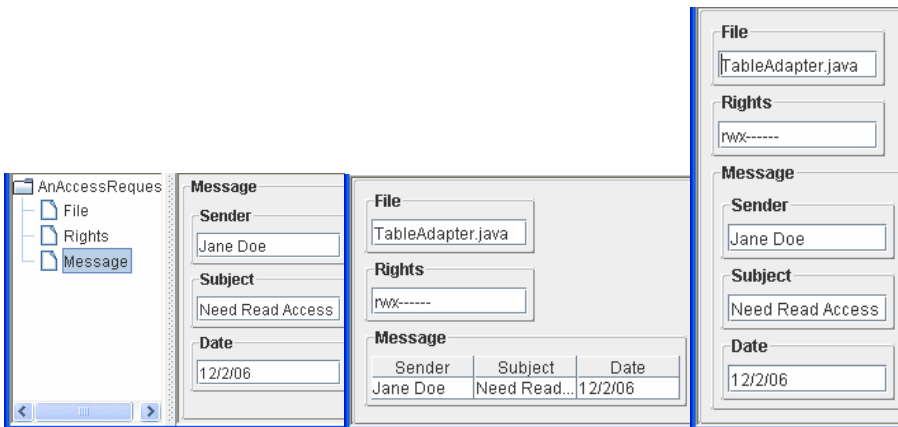


Fig. 10. Simultaneously displaying a nested record using all structured widgets

Figure 10 illustrates the use of universal adapters to simultaneously display a model using all structured-widgets supported by the toolkit. The model is an instance of `AnAccessRequest` with three fixed `String` keys, “File,” “Rights,” and “Message”, which are associated with elements of type `String`, `String`, and `AMessage`, respectively.

## 6 Discussion

We have described above the interface of a model object, and techniques for automatically binding it to both existing and new model-aware structured widgets. Thus, in comparison to existing user-interface toolkits, we simultaneously support a reduced model set and expanded model-aware widget set. Determining if we meet the other two requirements presented in Section 3 requires more analysis.

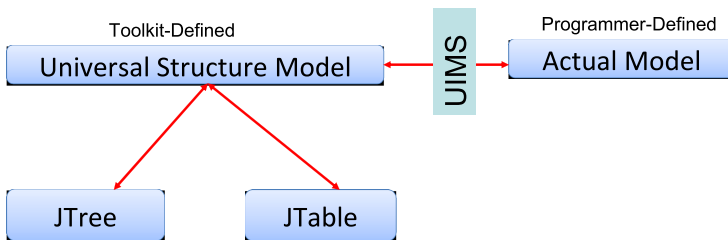
We went through (a) first a top-down phase in which we derived the interface of the universal table from well-established display-agnostic semantic structures, and (b) then a bottom-up phase in which we added additional methods to the interface needed by existing widgets. These methods do not increase the functionality of the model – their main purpose is to provide information the user-interface needs. For example, the user-interface needs to know the default key or element that should be added when the user executes the insert command. Similarly, it needs to know which keys and elements should be editable so that it can prevent the user from editing its visual representation.

Did the second phase compromise model purity? The answer, we argue, is no. The MVC architecture requires that the model be unaware of details of specific user-interfaces, so that these details can be changed without modifying the model. It is aware, however, that it will have one or more user interfaces – it allows views to be attached to it and sends notifications to them. The methods we have added play a similar role. The code in them also serves the same purpose as assertions. Assertions describe the behavior of an object to programmers, and prevent many mistakes. The additional methods we added in the bottom-up phase describe the behavior of an object to other objects – in particular the user-interface objects – and prevent mistakes. Consider the `isEditable()` methods. If a key or element is not editable, the model will not change it in the `put` method. However, an external object such as an editor would have to try to indirectly learn this behavior from repeated calls to the method. The `isEditable()` methods make this behavior explicit. Similarly, the methods returning the default key/element make the most specific class of the key/element apparent, and prevent additions of components of the wrong type. Just as notifications are now also used by non user-interface objects, we can expect these additional methods to have more general uses in the future.

Consider now programming effort. Mostly, our model does not require programmers to expose any information that is not also required by models of Swing. One exception is the information about editability of table data and components. While the Swing table model requires this information, the tree model does not. As this information not only increases the user-interface functionality but, in the long term, can be expected to prevent mistakes, we can say it does not significantly increase the programming cost. On the other hand, Swing requires tree nodes to keep track of their parent, and indicate if they are leaf nodes. If programmers are not careful, a forward (child) link can easily become inconsistent with a back (parent)

link, leading to significant debugging effort. Such links are kept by our implementation but not the models. In addition, our approach uses keys as default labels of elements, which works in several user-interfaces such as the ones shown here. Thus, in some respects, our approach reduces the programming effort required to create models of even existing model-aware widgets. In summary, our approach meets the programming effort requirement.

This is not to say that our design has created the best user-interface tool today. There is limited abstraction flexibility in that all models of a widget must implement the same toolkit-defined interface. In addition, programmers must manually determine the widget to be bound to a model, and set label and other user-interface attributes of these widgets. These are also limitations of existing toolkits. However, certain user-interface management systems (UIMS) such as [10-13] provide higher abstraction and automation. For these tools, our approach provides a method for increasing portability and reducing programming cost. We described above a simple approach for converting between the universal tables and existing models. If such code is added for each toolkit, then by layering on top of the universal table, a UIMS becomes portable and does not have to worry about implementing the new model-aware user-interface components supported by the universal table. We are planning to use this approach in a UIMS we are implementing as part of the ObjectEditor software[10]. For example, the properties of an object defined through getters and setters will be mapped to record fields, and then, using the scheme described above, to keys and elements of a universal table, which acts a proxy between the object and the widget. The interface of such an object would be programmer-defined and, hence, not constrained to a universal table. Thus, this approach assumes that a structured widget is linked in a chain to two models: a toolkit-defined proxy-model and a client-defined real-model. A UIMS can automatically translate between the events and operations of the two models, making the programmer oblivious of the toolkit-defined model. It is also possible to use this proxy-based approach in a manually-created user-interface – but the programmer would have to be responsible for translating between the two models. By reducing the number of toolkit-defined models, our approach reduces the number of translators that have to be written in the proxy-based approach.



**Fig. 11.** Interfacing with a UIMS to Support Programmer-Defined Types

To conclude, at the most abstract level, our message is that a toolkit should support both model and editor substitutability. At the next-level are the requirements of reduced model set, same or increased model-aware widget set, same or decreased programming effort, and model purity. The universal table interface and methods for



mapping it to sequences, sets, records and nested tables and binding it to tables, trees, forms, tabbed panes, and browsers provide one approach to meeting these requirements. More work is required to extend and refine the requirements and approach, use and evaluate the approach, and incorporate it in higher-level tools.

**Acknowledgments.** This research was funded in part by IBM, Microsoft and NSF grants ANI 0229998, EIA 03-03590, and IIS 0312328. The insightful comments of the reviewers and conference attendees helped improve the presentation.

## References

1. Krasner, G.E., Pope, S.T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk 1980. *Journal of Object-Oriented Programming* 1(3), 26–49 (1988)
2. Myers, B.A.: Separating Application Code from Toolkits: Eliminating the Spaghetti of Call-Backs. In: *ACM UIST 1991*, November 11-13 (1991)
3. Linton, M.A., Vlissides, J.M., Calder, P.R.: Composing User Interfaces with InterViews. *IEEE Computer* (February 1989)
4. Dewan, P.: A Tour of the Suite User Interface Software. In: *Proceedings of the 3rd ACM UIST 1990* (October 1990)
5. Olsen, D.R.: *User Interface Management Systems: Models and Algorithms*. Morgan Kaufmann, San Mateo (1992)
6. Codd, E.: A Relational Model for Large Shared Data Banks. *Comm. ACM* 13(6) (1970)
7. Fraser, C.W., Hanson, D.R.: A High-Level Programming and Command Language. In: *Sigplan Notices: Proc. of the Sigplan 1983 Symp. on Prog. Lang. Issues in Software Systems*, vol. 18(6), pp. 212–219 (June 1983)
8. Liskov, B.: Abstraction Mechanisms in CLU. *CACM* 20(8), 564–576 (1977)
9. Rowe, L.A., Shoens, K.A.: A Form Application Development System. In: *Proceedings of the ACM-SIGMOD International Conference on the Management of Data* (1982)
10. Omojokun, O., Dewan, P.: Automatic Generation of Device User Interfaces? In: *IEEE Conference on Pervasive Computing and Communication (PerCom)* (2007)
11. Nichols, J., Myers, B.A., Rothrock, B.: UNIFORM: Automatically Generating Consistent Remote Control User Interfaces. In: *Proceedings of CHI 2006* (2006)
12. Paterno, F., Manicini, C., Meniconi, S.: ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In: *INTERACT 1997* (1997)
13. Gajos, K., Weld, D.S.: SUPPLE: Automatically Generating User Interfaces. In: *IUI 2004* (2004)

## Questions

*Yves Vandriessche*

*Question: How do you finally handle the atomic objects?*

Answer: We don't, there are a lot of ways to handle this and they keep being reinvented every day.

**Remi Bastide:**

*Question: Most modern dynamic languages, e.g. Javascript, use the dictionary as the basic data structure and programmers tend to have their API towards using dictionaries. This conflicts your arguments.*

Answer: Most of these string-based languages actually come from SNOBOL.

**Morten Harning:**

*Question: Would it not be obvious to handle interface to user defined Java classes by treating objects not implementing Universal Table interface as Universal Tables by interpreting setters and getters as keys in a Universal Table.*

Answer: Absolutely. This is actually what we started doing, by only relying on Java naming conventions ended up being too messy.

# Exploring Human Factors in Formal Diagram Usage

Andrew Fish<sup>1,\*</sup>, Babak Khazaei<sup>2</sup>, and Chris Roast<sup>2</sup>

<sup>1</sup> University of Brighton, School of Computing,  
Mathematical and Information Sciences, UK  
Andrew.fish@brighton.ac.uk

<sup>2</sup> Sheffield Hallam University,  
Culture, Communication and Computing Research Institute, UK  
B.Khazaei@shu.ac.uk, c.r.roast@shu.ac.uk

**Abstract.** Formal diagrammatic notations have been developed as alternatives to symbolic specification notations. Ostensibly to aid users in performing comprehension and reasoning tasks, restrictions called wellformedness conditions may be imposed. However, imposing too many of these conditions can have adverse effects on the utility of the notation (e.g. reducing the expressiveness). Understanding the human factors involved in the use of a notation, such as how user-preference and comprehension relate to the imposition of wellformedness conditions, will enable the notation designers to make more informed design decisions. Euler diagrams are a simple visualization of set-theoretic relationships which are the basis of more expressive constraint languages. We have performed exploratory studies with Euler diagrams which indicated that novice user preferences strongly conform to the imposition of all wellformedness conditions, but that even a limited exposure diminishes this preference.

## 1 Introduction

Formal notations have been advocated as important within a variety of software development contexts, since they can offer clarity and precision; the provision of sophisticated tool support can strengthen confidence in the development processes and the quality of the end product. However, the role that such a notation plays is that of a representation that has to be composed, comprehended and updated as part of the development process. Hence, although the formality is a valued facet, there are other significant factors that affect their value. For example, notation appropriateness [1,2] can influence the quality of solutions that a user may entertain. Additional factors can come into play when we consider the role of environments which can affect the particular form in which information is expressed. Users working in different environments may have added difficulties when sharing specifications if their environments enforce the use of different forms of expression for example. The role of a representation is also important, since any specification is likely to be influenced by its primary use (to communicate to others or to record information for instance).

Software developers and design teams who have to work with formal notations are end-users. When designing a formal notation, features that support its intended use

---

\* Work partially funded by EPSRC grant number EP/E011160.

and uptake are often provided, but features that may limit its effective use can often be accidentally included. For example, the provision of “macros” or “libraries” for a base notation are features that are useful in supporting the user, whereas a lack of symbol discriminability and limited spatial layout can increase the difficulty in using notations [3]. Difficulties in encouraging user-uptake of symbolic formal specification notations, such as Z, is one of the reasons for the development of diagrammatic specification notations, such as constraint diagrams [4,5]. Also, the Unified Modeling Language is now commonly used in the software development cycle and since the only non-diagrammatic component is the Object Constraint Language (which can be used to express system invariants and pre/post-condition contracts for operations for example), a suitable diagrammatic alternative would fit in with the diagrammatic paradigm. It could also potentially widen the scope of usage of constraints by making them more accessible than their symbolic counterparts, and this may improve readers' understanding of formal specification documentation for instance.

When defining a visual specification notation, the presentation features are important for effective use: they can assist a user in easily identifying syntactic structures and facilitate the interpretation of semantic characteristics. Our perceptual competence at recognizing bounded areas and arcs in diagrams suggests that often less mental effort needs to be devoted to identifying syntactic structures in a diagrammatic representation than in a symbolic one. For a diagrammatic notation to be effective it is important that the relationships in the representation are well-matched with the domain characteristics [6], then the spatial relationships of the representation can lead to free rides [7], which are inferences gained for free due to the well-matching. Despite this potential, nothing limits visual specification notations from being used in a manner that does not exploit these benefits. Hence, it is of interest to examine approaches to ensuring the valuable use of visual notations and the subjective assessments that may also play a role in effective visual notation use.

Restrictions of the presentation of information (such as not allowing three lines to meet at a single point) are called wellformedness conditions. Such conditions are usually imposed with the intention of making the diagrams easier to comprehend and reason with. However, often there has been little or no user testing to determine the actual effects on users of these restrictions. Also, there can be many possible choices of condition and enforcing them can have side-effects such as reducing the expressiveness of the system, or of making it more difficult to present certain statements. Thus a balance needs to be struck between the imposition of some of these conditions and the utility of the system (e.g. being able to visualize as much as possible). Discovering the effects of wellformedness conditions on user preference and performance will help notation designers to determine the correct balance of conditions to be relaxed or enforced for different user groups.

Euler diagrams are a diagrammatic method for representing information about the relationships between sets. They have been used in various forms since Euler [8] first introduced them, and they generalize Venn diagrams [9] which represent all set intersections. Euler diagrams, and notations based on them, are currently being used in many application areas for the presentation of information, including: to represent non-hierarchical directories [10,11]; to visualize complex genetic set relations [12]; to represent ontologies in semantic web applications [13]; to enable the visualization of statistical data [14]. However, the main focus of usage of such diagrams is to model

object oriented software systems [5, 15,16] and to develop formal, diagrammatic logical reasoning systems and automatic theorem provers [4, 17,18-22]. Recently, the consequences of enforcing certain wellformedness conditions on Euler diagrams in the areas of drawability, semantic interpretation and reasoning have been investigated [23], and we wish to acquire further related information about user-preference.

In this paper, we describe exploratory studies which primarily set out to explore the relationship between the use of wellformedness conditions for Euler diagrams and user-preference. In addition, we examine how user comprehension is linked to these wellformedness conditions. The examination of preference is aimed at helping us develop an understanding of whether wellformedness is seen as desirable by users of diagram as well as by those designers who espouse the wellformedness conditions. Related long term goals include understanding how a user's preference may change with experience, how the context of use affects preference and how preference is linked to comprehension. The current exploration provides evidence about the importance of the choice of wellformedness conditions for potential users.

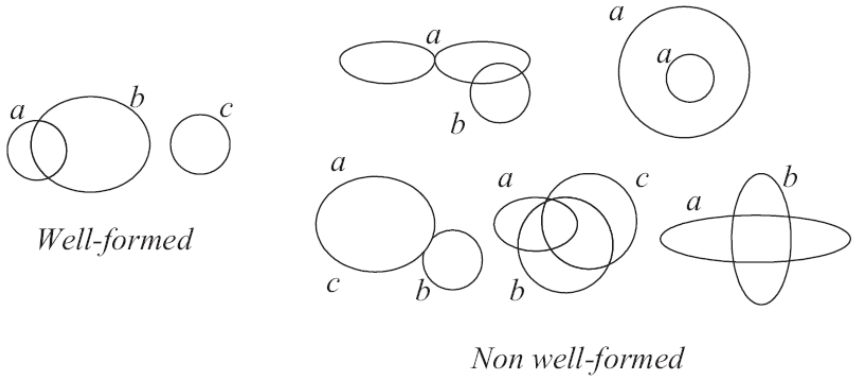
## 2 Euler Diagrams

We give an informal definition of Euler diagrams and their wellformedness conditions (see [24] for more details). An *Euler diagram* is a finite set of labelled closed curves (called *contours*) in the plane. A *zone* (or *minimal region*) is a connected component of the complement of the contour set, and a *region* is a union of zones. An Euler diagram is *well-formed* if it satisfies a given set of wellformedness conditions. A typical set of *wellformedness conditions* are:

1. **Simple contours:** The contours are simple closed curves.
2. **Unique contour labels:** Each contour has a unique label.
3. **No concurrency:** Contours are not concurrent (that is, they meet at a finite, discrete set of points).
4. **No tangential intersections:** Contours do not touch, but can cross each other transversely wherever they meet.
5. **No multiple points:** No more than two contours meet at any single point.
6. **Unique zone labels:** Each zone can be uniquely identified by the set of contours containing it and the set of contours excluding it.

For example, a well-formed Euler diagram is shown on the left of Figure 1. It has three labels  $a, b, c$ , three corresponding contours, and five zones determined by the label sets  $\{\}$ ,  $\{a\}$ ,  $\{a, b\}$ ,  $\{b\}$  and  $\{c\}$  – corresponding to the regions outside all contours, inside just  $a$ , inside both  $a$  and  $b$ , inside just  $b$ , and inside just  $c$ , respectively.

Figure 1 also shows five non well-formed diagrams. The top left one shows a “figure of eight” curve (which is a non-simple curve) for the contour labelled by  $a$  and so this fails condition 1. The top right diagram depicts two contours both of which are labelled by  $a$  and so this fails condition 2. The bottom left diagram fails condition 3 because it has two contours ( $a$  and  $c$ ) which are completely concurrent, that is, one is overlaid on top of the other; it also has two contours ( $a$  and  $b$ ) which meet tangentially and so it fails condition 4. The bottom middle diagram has a triple point (which is a multiple point) – all three contours pass through a single point – and so it fails



**Fig. 1.** Wellformedness of Euler diagrams

condition 5. The bottom right diagram has zones which are not uniquely identifiable using the contour labels – the region which is inside *a* but outside *b* is disconnected – and so it fails condition 6.

Conditions 1, 2 and 4 are almost always enforced, with simplicity only previously being relaxed to make reasoning easier [25]. Conditions 3 and 5 are sometimes enforced [20] and sometimes not [26]. Condition 6 is usually enforced, and has often been referred to as “no split /disconnected zones” [27].

**2.1 Semantics**

The semantics of Euler diagrams that we adopt are:

1. the interior of each contour represents the set denoted by its label, and each region of the diagram represents the corresponding set intersection determined by the labels.
2. a shaded or missing region of the diagrams represents an empty set, whilst a non-shaded region that is present in the diagram represents a non-empty set.

Figure 2 shows two semantically equivalent diagrams, each depicting three non-empty sets (*A*, *B* and *C*) such that  $A \cap C = \emptyset$  (that is, *A* and *C* are disjoint),  $B \cap C = \emptyset$  and  $A \cap B \neq \emptyset$ . The second utilizes shaded zones to indicate emptiness, whereas the first used the absence of zones for this. The introduction of shading into the system enables more varied forms of expression, and whilst it has the benefit of explicitly depicting emptiness (which can only be depicted by omission without the use of shading) it also brings with it greater diagrammatic complexity arising from more overlapping of the contours, as well as the need to understand more syntax.

There are many slight variations in diagram semantics in the literature (as well as extensions). The most notable variation here is that often dots or crosses (called spiders, constant sequences or *x*-sequences for example) are required to be placed in regions to represent non-emptiness [20 – 22], but we wished to burden the subjects with

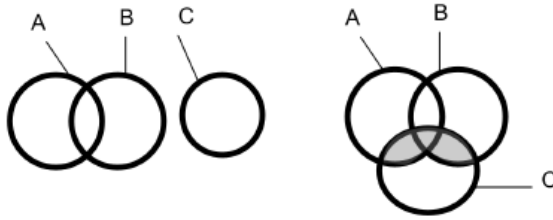


Fig. 2. Semantically equivalent diagrams

as little additional syntax as possible. An avenue that warrants further study is the testing of different choices of semantics for Euler diagrams – how these affect user preference and understanding, especially in the presence of various wellformedness conditions.

## 2.2 Roles of Euler Diagrams

Euler diagrams are thought to be an effective representation since the set-theoretic relationships that they represent are well-matched by the spatial relationships that they use. For example, the proper subset relationship is well-matched to the proper inclusion of curves in the plane (both are transitive, but not reflexive or symmetric). We can obtain the inference “ $A \subset C$ ” from “ $A \subset B$ ” and “ $A \subset C$ ” for free from the corresponding Euler diagram with three concentric circles shown in Figure 3, for instance.

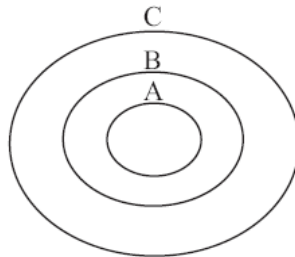


Fig. 3. Well-matching and free rides

An example demonstrating the importance of the role of notation and environment for Euler diagrams occurs in interactive Euler diagram theorem proving environments. Information is stored and reasoned with at an abstract level, which is useful for computations, but is not so appropriate for presentation to a user (to enhance user faith in an automated proof, or as an aid to understanding proof techniques). Imposing too many wellformedness conditions can prevent certain set theoretic statements from being represented diagrammatically, and a change of representation here is likely to be undesirable for a user. Furthermore, one may wish to tailor the presentation of a proof to individual user preference. For example, an environment could offer a choice between a short proof containing non wellformed diagrams and a longer proof using wellformed diagrams. In general there is a balance to be found between the number of diagrams in a proof to be displayed and both the size of the deduction steps used,

and the complexity of the diagrams used. An experienced user might prefer to view a non wellformed diagram which is compact but contains a lot of information over a collection of wellformed diagrams which are individually easier to comprehend but one also has the added cognitive load of having information spread across more diagrams.

It is also important to remember that diagrams may be authored, read and edited by different people (possibly in different groups, in different countries, using different environments). Therefore, being too rigid in the enforcement of wellformedness conditions may have detrimental effects on communication.

### 3 Preliminary Study

Before conducting a fully fledged study examining users perceptions of, and competence with, Euler diagrams a preliminary study was conducted to get some feedback on the likely outcomes of the main study. The pilot study was conducted with five subjects who were all half-way through a second year degree option on human-computer interaction.

Introducing the concept of Euler diagrams to subjects who may be unfamiliar with discrete maths concepts presented the problem that they may easily view diagram comprehension tasks as being assessments of their ability. Because of this we wished to focus upon non-abstract examples of Euler diagrams (for example, not using alphabetical labelling of the contours), while also ensuring that any examples would not encourage subjects to guess at answers based upon personal knowledge or expectation. To this end we developed a contextual setting in which Euler diagrams were proposed as a form of graphical output to an internet search facility - termed "Oigle"! Within this setting, subjects could be easily encouraged to focus upon judging the value and utility of the diagrams presented as output. The labels used for the concrete examples were motivated by lists of popular internet search terms. As far as possible, labels which were considered to have strongly related meanings were not used together in the same diagram. We also adopted a slightly different labelling convention than usual (compare Figures 1 and 2), with the aim of reducing potential ambiguity caused by the placement of a label.

The preliminary study took around 40 minutes, with time equally divided between: familiarization and training, and comprehension questions. The familiarization and training involved a short introduction to the "Oigle" concept and some basic examples of its output. Subjects were given four well-formed diagrams involving no more than four contours, asked to briefly describe them, and given the chance to compare their answers with model descriptions.

After the familiarization phase, subjects answered twenty "yes/no" comprehension questions. The questions were related to a sheet of nine diagrams, four of which were not well-formed; each of these diagrams involved no more than four contours. The results showed an average score of 16/20 indicating a good level of comprehension. The greatest variability in the answers was found for those questions concerning non well-formed diagrams, indicating that wellformedness within the diagrams used was influential. Additional feedback from the subjects indicated a preference for "avoiding unnecessary area divisions", "providing a neater layout of diagrams with more symmetry" and "clearer separation where regions were separate". Within the confines of



the study, no clarifications of the descriptions were offered. Subsequent follow-up studies will involve more focused interviews with the subjects who volunteered these explanations.

This initial study provided validation for the experimental setting, timing, and the potential to explore the influence of wellformedness conditions on comprehending the Euler diagram notation, especially when working with novice subjects. From this study we concluded that the questions could be more difficult, and we could include more complex diagrams. Therefore, a slightly more complex Euler diagram convention was chosen for the main study: we chose to employ the concept of a shaded region to indicate an empty set, as illustrated in Figure 2. Introducing shaded regions into the notation allows the same information to be represented by a greater variety of diagrams, and this provides a useful way of adding to the complexity of the experimental materials.

The preliminary study highlighted the variety of concrete layouts that exist for Euler diagrams, whether well-formed or not. For example, two overlapping contours can be drawn varying the relative sizes of the two contours and their relative position, and of course, their individual shapes. For our study we wish to limit the unwarranted impact that this variety may have, and focus specifically on the wellformedness conditions. For this reason we selected some “scoping” heuristics designed to ensure a level of conformance in the style and layout of diagrams, thereby enabling wellformedness conditions to be assessed more accurately:

1. Keep regions of a similar consistent size (except in purposefully ambiguous cases). Hence, non-trivial overlaps should be shown clearly as such, but if a tangential intersection is to be displayed then the presence (or absence) of an overlap need not be clearly shown.
2. Do not stretch contours unnecessarily. Hence contours do not become distorted unnecessarily.
3. The bounding rectangle of a diagram (where this means a rectangle containing all of the contours) should be close to square.
4. All labels should be outside the bounding rectangle of the diagram, whilst being closest to the contours that they label; they should appear alphabetically in a clockwise order.

These heuristics were proposed as “good practice” that should be followed where possible.

## 4 Main Study

The main study was directed towards establishing an understanding of how users react or respond to the visual language, and how this relates to issues such as comprehension, wellformedness and less precise, though still significant, concepts such as visual appeal. Although user preferences for specific diagrams may be highly subjective, it is valuable to know how closely their preference follows the notion of wellformedness and also how influential it might be upon comprehension and thus, utility. It is quite possible, especially with novice users, that preferences can influence comprehension, both in terms of accuracy and also willingness to engage with diagram related problems.

## 4.1 Experimental Design

Based on previous studies requiring subjective responses [1,28], there is evidence that experience can be an influential factor, and so we chose to gather user preference data both before and after using Euler diagrams. A set of comprehension questions about Euler diagrams similar to those of the preliminary study served as a (limited) Euler diagram experience. We employed a form of subjective preferences reporting that allows preferences to be easily identified. It is not uncommon in some experimental settings for subjects to proffer responses that they believe to be those desired by the study. This effect can be limited by providing subjects with a comparative judgment task. In this case we asked subjects to indicate their most preferred and least preferred diagrams within given sets.

Although motivated by specific concerns about wellformedness, the study was primarily exploratory, focused upon revealing factors that may be relevant for further studies. If we were to posit hypotheses driving the study these would be:

1. Wellformedness conditions concur with user comprehension and user preferences.
2. Experience with Euler diagrams influences user preferences.

The study consisted of three phases, an *a priori* preferences assessment, a comprehension phase and a *post priori* preference assessment:

1. Subjects were presented with four questions showing groups of three similar diagrams and were required to indicate which they preferred the most and which they preferred the least. Each question had at least one well-formed diagram, and prior to the study an expert assessment of the quality of the diagrams was also recorded. Figure 4 shows an example of one of these questions.
2. Two relatively complex Euler diagrams were provided and ten “yes/no” comprehension questions given. The questions were balanced both between positive and negative answers, and between the two diagrams. The responses to two of the questions were contingent upon whether tangential intersections created a non-trivial region, so that the set intersection was non-empty (a *liberal* reading), or they did not create a non-trivial region, so that the set intersection was empty (a *conservative* reading). Figure 5 shows the two diagrams together with some of the questions used.
3. Subjects were presented with four further preference questions in the style of the first phase. Figure 6 shows an example of one of these questions.

In order to gather information on subjects' preferences, in both phases 1 and 3 subjects were given no indication of criteria by which to judge preference other than their own — they were simply asked to indicate which diagram they thought was “best” and which they thought was “worst”. Subjects were given the opportunity to report back on any reasoning or rationale that they used for each phase. The diagrams employed for each question in phases 1 and 3 were similar in complexity, although not all of the diagrams within each question were semantically equivalent since this could imply a received interpretation of the inherently ambiguous non-wellformed cases. One of the primary purposes of Phase 2 was to provide the end users with some experience of comprehension, and so it was felt that natural language questions would suffice (as opposed to more formal questions posed in symbolic logic for instance).

The experimental preparation involved an initial phase of familiarization and training taking 25 minutes. The students were then asked to participate in a half an hour experiment involving the above phases. It was explained that this was a comprehension exercise indirectly relevant to their course and that the purpose was not to assess them but to help us to understand both their preferences and their difficulties in comprehending some diagrams. Twenty five second year B.Sc. Software Engineering students took part in the study. The students were covering elements of system analysis and design based on the Object Constraint Language (OCL) and the Unified Modelling language (UML). The students had some mathematical background - the minimum being the pass level in GCSE Maths. The nature of the course and the focus of the experiment had some similarities as the students were covering the range of diagrammatic notations within the UML. All of the students had successfully completed one and a half academic years of programming and software engineering study.

## 4.2 Results

All of the participants completed the study without any expressed difficulty. About half of the subjects provided feedback on the reason for their choices, and a few subjects did not identify both “best” and “worst” preferences in some questions. The results for each phase are explained below and are summarized in Tables 1, 2 and 3. There is a strong conformance between subjects, suggesting that despite the subjective nature of phases 1 and 3, there is a considerable amount of agreement between subjects. For phases 1 and 3, Table 1 shows the total number of responses (over all subjects and all questions in the phase) of “best” (and “worst”) actually being a well-formed diagram. For example, in phase 1 there were 79 user choices of “best” which were well-formed, but only 21 user choices of “best” which were not well-formed. Both the a priori and post priori responses show a strong correlation with the proposition that novice subjects’ preferences match well-formed Euler diagrams. Comparing results for phases 1 and 3 in Table 1, we also see that experience with Euler diagrams appears to lessen the conformance. The significance of this relationship was assessed by comparing the average score for each subject (where the score for a subject is the number of choices of “best” that are wellformed) with the probable score for no effect (see Table 2). The probable score for each question was given by the proportion of well-formed diagrams available in the question. The results for both the a priori case and the post priori case are highly significant when compared with their predicted averages ( $p < 0.001$  with Wilcoxon matched-pairs signed-ranks test,  $N = 25$ ).

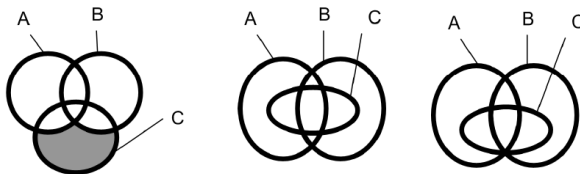
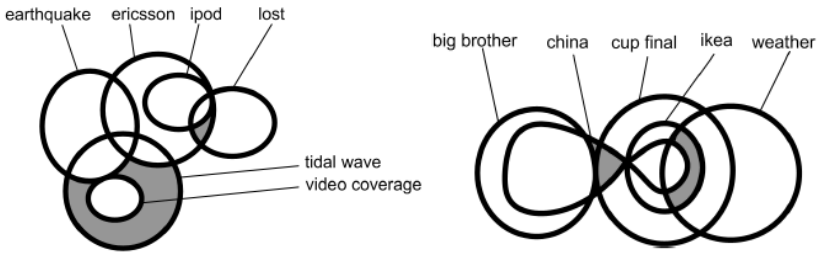


Fig. 4. Phase 1: choose the best and worst



There are some sites with results about [YES/NO] all three of “ipod”, “ericsson” and “lost”.
There are some sites matching “earth- [YES/NO] quake” and “tidal wave”.
Every site about “ipod” is also about ei- [YES/NO] ther “ericsson” or “tidal wave”.
Pages about “cup final” and “weather” [YES/NO] include some “ikea” pages.
There are no pages about “big brother” [YES/NO] and “weather”.

Fig. 5. Phase 2: comprehension

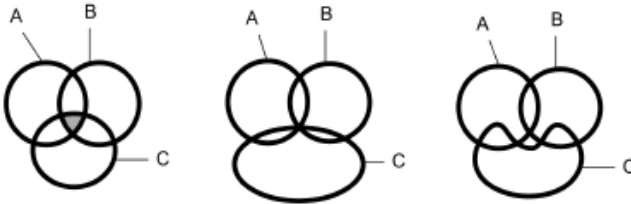


Fig. 6. Phase 3: choose the best and worst

Table 1. Results: matching preference to wellformedness in phases 1 and 3

well- formed	Phase 1		Phase 3	
	“best”	“worst”	“best”	“worst”
yes	79	12	68	35
no	21	81	31	57

Table 2. Results: comparing average score with expected average in phases 1 and 3

	Phase 1	Phase 3
average score (N = 25)	3.20	2.75
probable score	1.62	2.00

Subjects were also asked to provide feedback on the reasons for their preferences. From these the overarching theme was a preference for “clarity” and “readability”. Some subjects explicitly stated that their choice was based on visual appeal, occasionally with an explicit criterion such as having “symmetry”. Others alluded to more semantic concepts such as “set theory” and “equivalence”, with one subject explicitly referring to the problematic nature of tangential intersections. Some interesting cases focused more on comparative judgments, such as the degree of unnecessary complexity and the number of shaded regions; one subject even suggested that the textual equivalent of a diagram would be simpler to understand. Generally, we found a rich mixture of factors being employed ranging from visual and aesthetic concerns through to semantic clarity. This mix of informal comments combined with the high correlation of results suggests that, for the cases examined, wellformedness promotes visual representations that users can interpret as being both visually clear and also easy to interpret logically.

The comprehension task (phase 2) illustrates that the subjects, on the whole, have a reasonable grasp of Euler diagram semantics, especially as the diagrams used were considerably more complex than those of the preliminary study (see Figure 5). Table 3 shows the results of this task, with the “average” column indicating the average number of correct answers. An interesting feature is that diagram 2 (the bottom diagram in Figure 5) has a significantly lower level of accurate comprehension than diagram 1 (the top diagram in Figure 5). Diagram 2 involves more instances of non wellformedness than diagram 1. The difference between the diagrams was also apparent in the subjects' feedback, with several subjects referring to the complexity of the second diagram. Phase 2 also showed that in a minority of cases (2/7) tangential intersections were interpreted “liberally” (i.e. as being a non-empty intersection). Perhaps the most interesting observation here is that five of subjects (20%) were not consistent in their interpretation of the two diagrams. Hence, these subjects altered their interpretation based on the diagram encountered - an effect that could be attributed to the complexity of diagram 2 or the informal interpretation of the labels used in the diagram.

**Table 3.** Results from phase 2 (averages and interpretation of tangential intersections)

	Average	<i>liberal</i> reading
Diagram 1	3.92 (78%)	2
Diagram 2	2.64 (53%)	5
Total	6.56 (66%)	7

Our study was also interested in the influence of experience with Euler diagrams in phase 2 on subjects' preferences. Although phase 1 and phase 3 were not formally balanced in the experimental design, the conformance of subjects preferences is reduced for the post priori case (phase 3). The difference between the a priori and post priori can be confirmed statistically: the proportionate score in the two cases were compared using Wilcoxon and showed a significance of  $p < 0.05$ . The potential influences behind this effect are the subjects' experience with phase 2, and possible differences in complexity between phase 1 and phase 3. In order to exclude the second of

these factors, the variances of individual questions within phase 1 and phase 3, as well as the likely complexity of the questions was examined. In addition, the question complexity (or, more precisely, the complexity of the diagrams used in the questions) was assessed using a variety of metrics including: the range of wellformedness conditions contravened; the number of labels; the number of regions; the number of shaded regions; the clutter metric of [29]. From this inspection, no obvious candidate factors for differentiating between the question complexity in phase 1 and phase 3 were found.

To clarify whether the limited experience introduced by phase 2 was influential on user-preference, a supplementary study was conducted in which the questions from phases 1 and 3 were combined. The study was with 15 Masters level students, the high educational level of these subjects appeared to be reflected in the subjects' comprehension performance which was on average 73% (which is 7% higher than the average for the main study). Responses to the preference questions of phases 1 and 3 were less conformant than those in the main study, with an overall average score of 4.8 (see table 4 for a direct comparison of the results). Responses to questions from phase 3 were more conformant than those from phase 1 (though not significantly so). Hence, the effect observed in the main study was reversed in the supplementary when phase 2 was excluded. These additional results confirm that the differing question complexity of the phases 1 and 3 is not significant, and thus is unlikely to be influential in the main study. This strengthens the observation that the conformance is weaker by virtue of the experience of comprehension questions in phase 2.

**Table 4.** Conformance results for preference questions in the studies

well-formed	Main Study		Supplementary Study	
	“best”	“worst”	“best”	“worst”
yes	74%	25%	66%	36%
no	26%	75%	34%	64%

Given the conformance between subject preferences and well-formed diagrams, the impact of the different wellformedness conditions was also examined. The influence of individual wellformedness conditions on subject preferences were found to be ordered by relevance as follows: *multiple points*, *tangential intersection*, *non-simple contours*, *unique zone labels*, and *concurrency condition*. Hence, the presence of multiple points is the most influential condition upon user responses, and concurrency the least. However, this influence is not strong, and the order varies dramatically when examined for phase 1 and phase 3 separately. In addition, it is hard to draw firm links between this order and the feedback from users, as the users were not introduced to the specific conditions being examined. One area for further study is to encourage subjects to articulate their preference rationales in terms that can be easily related to the proposed wellformedness conditions. Also, the systematic testing of individual sets of conditions is a next step, using the feedback from this study to prioritize which sets of conditions are worth investigating first (given that there are many possible sets of conditions that can be imposed).

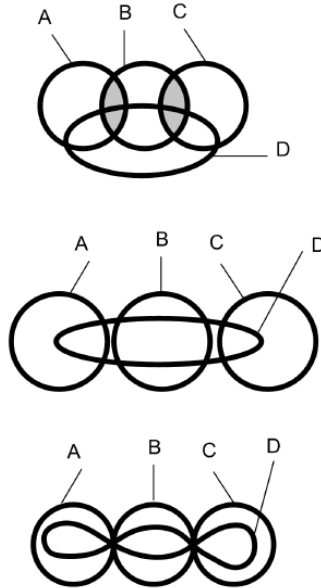
## 5 Discussion and Conclusions

In our exploratory studies with Euler diagrams, we tried to keep the diagrams as simple as possible, whilst still allowing enough variety to present collections of semantically similar diagrams for the subjects to choose from. One reason for this was to ensure that novices to the subject could easily engage with the tasks; the results of the subjects in the comprehension task indicate that the study was set at a level which was sufficiently cognitively demanding. Despite having only just been introduced to the concept of Euler diagrams, we found that subject preferences strongly conformed with the imposition of the wellformedness conditions (a significant result confirming our motivating hypothesis), but that experience with Euler diagrams influences user preference. The link between the wellformedness conditions and user comprehension is only very weakly evidenced by the results from phase 2 of the study (which shows a slight drop in comprehension for questions relating to the diagram with more instances of non-wellformedness). The authors are currently preparing to examine the link between user-preference and comprehension in the presence of the wellformedness conditions in more detail. We believe that comprehension needs to be examined in both of the settings of interpretation and construction of diagrams.

The “good practice” heuristics that were teased out in our preliminary study (section 3) enabled a greater degree of consistency and a better style of diagrams (which is something that often comes from user-experience). Thus, such scoping heuristics could improve the effectiveness, and help ease the uptake, of the notation. Future work will involve user testing to refine and justify the choice of such heuristics.

One of the long term aims of examining user preferences of wellformedness is to identify which conditions are of greatest use or value to users, and in what circumstances. Any resulting prioritization of wellformedness conditions can be useful in identifying which conditions can be relaxed with minimal disruption for users. Furthermore, it could inform the prioritization of theoretical work on the automatic generation of diagrams (to display output in a diagrammatic theorem proving environment for example). Although a prioritization of the conditions was identified for the study reported, it was not significant or stable and so the examination of specific conditions and their relevance for users requires further study. One feature of considering prioritized conditions is that they may vary between users, in which case a more fundamental question might be whether or not the wellformedness conditions represent coherent presentation constraints for users. For instance, in phase 2 of the study some subjects' interpretations of tangential intersections were not consistent, in that they varied between diagrams. The examination of the effect of working on the comprehension task indicates that this weakened the initial conformance of preference and wellformedness. This suggests that the experience of working with the diagrams enables subjects to tease out and discriminate between aesthetic factors and cognitively demanding factors when interpreting diagrams. Expert users may also have different preferences to novices as they may wish to represent or see information presented more compactly or laid out in a certain manner according to their task. For example, for the question shown in Figure 7, a diagram expert who completed the study preferred the non wellformed middle diagram (which has split zones), and not the wellformed first diagram (which had shaded regions). Therefore, we believe that wellformedness conditions for diagrammatic notations should be treated as presentation

aids, enforced and dismissed as the user wishes in order to aid understanding of the diagrams. Adopting the use of wellformedness conditions is likely to be especially useful for communication with novice users, but their a-priori imposition can restrict the notational utility and have an adverse effect on user perception of the notation.



**Fig. 7.** Phase 3: choose the best and worst

Subjects' informal feedback showed a broad range of concerns ranging from aesthetic, through diagrammatic clarity to more cognitively demanding issues such as diagram semantics. As well as demonstrating the variety of possibly conflicting concerns, this indicates other possible conditions which may influence preference and comprehension (such as symmetry and area proportionality). After suitable testing, notation designers could use such information either to introduce new, or refine existing, wellformedness conditions, or as possible improvements to the scoping heuristics. Ideally this would lead to an improvement in user communication when using these diagrams.

Constraint diagrams [17,4,5] are an extension of Euler diagrams, with added facilities to explicitly express quantification and navigation expressions. They were designed for use as a formal specification and reasoning system in an object oriented setting. We imagine that as systems become more complex (and expressive) by adding extra syntax, the difficulties in understanding by a user will increase and that this may affect user preference for wellformedness. Future studies will be performed to test these more complex notations, but an incremental testing approach was deemed necessary in order to build up a realistic understanding of user's preferences (otherwise there are too many complex interactions to be able to easily isolate any properties to test).



One of the many benefits of this exploratory research is the number of areas and questions that have been identified as needing further testing. In the long term we intend to provide a general framework for testing user preference and comprehension which will enable developers and users to gain insights into the human factors of their favourite notations. Our empirical study points to the need for the thorough investigation of any conditions imposed on a specification notation as a potential source for usability problems, and to consider possible improvements of the choice of conditions imposed based on user preferences. We advocate that a good design of an environment for authoring, viewing and editing diagrammatic specifications would in fact allow a user-based choice of which wellformedness conditions are imposed (which could be different for different users).

## References

1. Khazaei, B., Roast, C.: The Influence of Formal Representation on Solution Specification. *Requirements Engineering* (8), 69–77 (2003)
2. Roast, C.R., Siddiqi, J.I.: Contrasting Models for Visualisation (Seeing the wood through the trees). In: Duce, D., Puerta, A. (eds.) *Design, Specification and Verification of Interactive Systems 1999*, EuroGraphics. Springer, Wein (1999)
3. Britton, C., Jones, S.: The Untrained Eye: How languages for software specification support understanding in untrained users. *Human-computer Interaction* 14, 191–244 (1999)
4. Fish, A., Flower, J., Howse, J.: The Semantics of Augmented Constraint Diagrams. *Journal of Visual Languages and Computing* 16, 541–573 (2005)
5. Kent, S.: Constraint Diagrams: Visualizing Invariants in Object Oriented Modelling. In: *Proceedings of OOPSLA 1997*, pp. 327–341. ACM Press, New York (1997)
6. Gurr, C.: Effective Diagrammatic Communication: Syntactic, Semantic and Pragmatic Issues. *Visual Languages and Computing* 10(4) (1999)
7. Shimojima, A.: Operational constraints in diagrammatic reasoning. In: Allwein, G., Barwise, J. (eds.) *Logical Reasoning with Diagrams*, pp. 27–48. Oxford University Press, Oxford (1996)
8. Euler, L.: *Lettres a une Princesse d'Allemagne sur divers sujets de physique et de philosophie*. Letters Berne, Socit. Typographique 2, 102–108 (1775)
9. Venn, J.: On the diagrammatic and mechanical representation of propositions and reasoning. *Phil. Mag.* (1880)
10. Chiara, R.D., Erra, U., Scarano, V.: Vennfs: A venn diagram file manager. In: *Proceedings of Information Visualisation*, pp. 120–126. IEEE Computer Society, Los Alamitos (2003)
11. Chiara, R.D., Erra, U., Scarano, V.: A system for virtual directories using euler diagrams. In: *Proceedings of Euler Diagrams 2004*. *Electronic Notes in Theoretical Computer Science*, vol. 134, pp. 33–53 (2005)
12. Kestler, H., Muller, A., Gress, T., Buchholz, M.: Generalized venn diagrams: a new method of visualizing complex genetic set relations. *Journal of Bioinformatics* 21(8), 1592–1595 (2005)
13. Hayes, P., Eskridge, T., Saavedra, R., Reichherzer, T., Bobrovnikoff, D.: Collaborative knowledge capture ontologies. In: *Proceedings of K-CAP 2005* (2005)
14. Chow, S., Ruskey, F.: Drawing area-proportional venn and euler diagrams. In: Liotta, G. (ed.) *GD 2003*. LNCS, vol. 2912, pp. 466–477. Springer, Heidelberg (2004)
15. Harel, D.: On visual formalisms. In: Glasgow, J., Narayan, N.H., Chandrasekaran, B. (eds.) *Diagrammatic Reasoning*, pp. 235–271. MIT Press, Cambridge (1998)

16. Howse, J., Schuman, S.: Precise visual modelling. *Journal of Software and Systems Modeling* 4, 310–325 (2005)
17. Fish, A., Flower, J.: Investigating reasoning with constraint diagrams. In: *Visual Language and Formal Methods 2004*, Rome, Italy. ENTCS, vol. 127, pp. 53–69. Elsevier, Amsterdam (2004)
18. Flower, J., Masthoff, J., Stapleton, G.: Generating readable proofs: A heuristic approach to theorem proving with spider diagrams. In: *Proceedings of Diagrams 2004*, Cambridge, UK, pp. 166–181. Springer, Heidelberg (2004)
19. Hammer, E.: *Logic and Visual Information*. CSLI Publications (1995)
20. Howse, J., Stapleton, G., Taylor, J.: Spider diagrams. *LMS J. Computation and Mathematics* 8, 145–194 (2005)
21. Shin, S.J.: *The logical Status of Diagrams*. Cambridge University Press, Cambridge (1994)
22. Swoboda, N., Allwein, G.: Using DAG transformations to verify Euler/Venn homogeneous and Euler/Venn heterogeneous rules of inference. *Journal of Software and Systems Modeling* 3(2), 136–149 (2004)
23. Fish, A., Stapleton, G.: Formal issues in languages based on closed curves. In: *Proceedings of VLC 2006, Visual Languages and Computing*, Grand Canyon, USA, Knowledge Systems Institute, pp. 161–167 (2006)
24. Flower, J., Howse, J.: Generating Euler diagrams. In: *Proceedings of Diagrams 2002*, Callaway Gardens Georgia, USA, pp. 61–75. Springer, Heidelberg (2002)
25. Swoboda, N., Allwein, G.: Heterogeneous reasoning with Euler/Venn diagrams containing named constants and FOL. In: *Proceedings of Euler Diagrams 2004*. ENTCS, vol. 134. Elsevier Science, Amsterdam (2005)
26. Ruskey, F.: A survey of Venn diagrams. *Electronic Journal of Combinatorics* (1997), <http://www.combinatorics.org/Surveys/ds5/VennEJC.html>
27. Howse, J., Molina, F., Shin, S.J., Taylor, J.: Type-syntax and token-syntax in diagrammatic systems. In: *Proceedings FOIS 2001: 2nd International Conference on Formal Ontology in Information Systems*, Maine, USA, pp. 174–185. ACM Press, New York (2001)
28. Roast, C.R., Steele, R.A.: Using interfaces and liking interaction. In: Sharp, H., Le Peuple, J., Chalk, P., Rosbottom, J. (eds.) *Proceedings of Human-Computer Interaction 2002*. BCS, vol. 2, pp. 46–49 (2002) ISBN:1-902505-48-4
29. John, C., Fish, A., Howse, J., Taylor, J.: Exploring the notion of Clutter in euler diagrams. In: Barker-Plummer, D., Cox, R., Swoboda, N. (eds.) *Diagrams 2006*. LNCS, vol. 4045, pp. 267–282. Springer, Heidelberg (2006)

# ‘Aware of What?’ A Formal Model of Awareness Systems That Extends the Focus-Nimbus Model

Georgios Metaxas and Panos Markopoulos

Industrial Design, Technical University of Eindhoven,  
The Netherlands  
{g.metaxas,p.markopoulos}@tue.nl

**Abstract.** We present a formal-model of awareness-systems founded upon the focus and nimbus model of Benford et al [2] and of Rodden [19]. The model aims to provide a conceptual tool for reasoning about this class of systems. Our model introduces the notions of *aspects*, *attributes* and *resources* in order to expose the communicational aspects of awareness-systems. We show how the model enables reasoning about issues such as deception and plausible deniability, which arguably are crucial for enabling users to protect their privacy and to manage how they present themselves to their social network.

**Keywords:** CSCW, formal models, awareness systems, focus-nimbus, Z.

## 1 Introduction

*Awareness* systems are communication systems whose purpose is to help connected individuals or groups to maintain *awareness* of the activities and the situation of each other. In the domain of group-work where awareness systems were first studied, awareness has been defined as “*an understanding of activities of others that provides a context for your own activities*” [8].

In a more social context, interpersonal awareness can be considered as an understanding of the activities and status of one’s social relations, derived from social interactions and communications with them. Casablanca [12] was an early influential project that explored the design space of awareness technology for the domestic environment. Astra [17] studied intentional communication for the extended family and demonstrated that such communication can enhance feelings of connectedness and can prompt rather than replace direct communications. CareNet [6] focused on “Assisted living” by informing professional care-givers as to medication, nutrition, falls, etc., of elderly patients living alone. The Digital-Family-Portrait (DFP) [20] was designed to provide peace of mind to adult children regarding a lone parent living at a distance.

The works cited represent just a tiny fraction of the growing literature on the topic of awareness systems, which expands to an ever increasing variety of physical and social contexts addressing an equally diverse range of user needs. We discern two trends regarding this proliferation of research on awareness:

- The great majority of awareness systems concepts proposed in related literature cluster around some basic themes; some of the most common themes are,

communicating to someone that you think about him/her, conveying simple presence information at a particular location, sustained audio video links between places, serendipitous discovery of information about others, supporting flexibility and the conjoint creation of meaning between participants, etc.

- Theoretical discussions motivating the design of such systems gravitate towards the phenomena surrounding the social aspects of using awareness. For example, T.Erickson [9] has introduced the concept of social translucence that encapsulates issues of inter-subjectivity between users of awareness systems. Other issues relate to privacy of people and ways in which they might manage their accessibility to others, (e.g., [13], [3], [14]).

These two trends point to the need for a clear conceptualization of awareness systems that lends some clarity to the description of relevant phenomena. More specifically, such a conceptualization should abstract away from detailed aspects of form and application context, to describe the communication aspects of awareness systems in terms relevant for discussing social interactions between users.

Schmidt [21] discussed the endemic lack of conceptual clarity for the research domain we sketched out above. Noting the contradictory uses of the term awareness, he argued that dichotomies between attention and peripheral awareness, active and passive awareness, explicit and tacit, etc., are misleading. Rather he argued that awareness should be described in reference of activities, practices or phenomena or object that a person is made aware of. In line with this argument, the remainder of the paper presents an abstract model of awareness systems that incorporates related concepts and supports reasoning regarding social aspects of using awareness systems.

## 1.1 Related Work

There have been several attempts to create mathematical abstractions of awareness. Inspired from biology, Bandini et al. [22] proposed the reaction-diffusion metaphor that aimed to make “*awareness mechanisms fully visible and accessible to the involved actors for the purpose of adaptability*”. The model is based on the notions of *space*, and *fields*. *Space* is populated by entities, and it is used to evaluate when entities come in contact and to express how fields propagate in the space. *Fields* are the means by which awareness information is brought in and propagated in the space, and influences the entities able to perceive it. Mechanisms governing the *emission* and *reception* of fields provide the capability of modulating *awareness* on the side of the emitter as well as of the receiver.

Fuchs et al. [11] suggest an event distribution model for CSCW environments, that can be applied to support shared awareness in systems for the coordination of cooperative work. The model proposes the representation of the environment as a *semantic network*. Awareness about changes and synchronous activities in the system is supported by the generation and distribution of events in the semantic network.

Benford et al. [2] introduced the notions of Nimbus and Focus in a spatial model of group interaction, in order to address mutual levels of awareness within a virtual environment.

- *Focus* represents a sub-space within which a person focuses their attention. The more an object is within your focus the more aware you are of it.

- *Nimbus* on the other hand represents a sub-space across which a person makes their activity available to others. The more an object is within your nimbus, the more aware it is of you.

Based on these notions Benford et al. define a “measure of awareness” as a functional composition of *Focus* and *Nimbus* quantifiers; this measure provides the answer to the question: “*In a given room, how aware is entity i of entity j via medium k?*”.

Rodden [19] expanded the focus/nimbus model for a wide range of cooperative applications, beyond the boundaries of spatial applications, by using set notation to describe focus, nimbus, and awareness and the operations that can be performed on them.

The focus-nimbus model of Rodden has had several applications since it was introduced. Recently, Cohen et al [10] constructed a first-order logic representation of focus and nimbus enabling the definition of higher level operations for controlling multi-media streams between communicators using higher level operations such as mute, hide, etc. SOGAM (Service Oriented Group Awareness Model) [15], is a recent implementation oriented model, focusing on web services that can support group-awareness. These renditions of Rodden’s model are application specific and are not appropriate for supporting a general model of awareness systems and for reasoning for user relevant aspects such as, privacy, translucence, etc.

Privacy and awareness represent flip sides of the same coin. Noting the duality of these needs Boyle and Greenberg [4] applied the concepts of *attention*, *fidelity*, and *identity* in order to define privacy needs in the ubicomp domain. They proposed the following characterizations for privacy needs:

- *Solitude*: control over one’s interpersonal interactions, specifically one’s *attention* for interaction.
- *Confidentiality*: control over other’s access to information about oneself, specifically the *fidelity* of such accesses.
- *Autonomy*: control over the observable manifestations of the self, such as action, appearance, impression, and *identity*.

Boyle and Greenberg go on to project their tripartite conception of privacy on Rodden’s focus/nimbus model for awareness. *Foci* correspond roughly to attention so solitude can be thought of as focus regulation. *Nimbi* correspond to embodiments and socially constructed personas and to one’s relationships with information and artifacts in the environment. Nimbus regulation therefore roughly corresponds to confidentiality and autonomy. *Awareness*, which is defined as a functional composition of focus and nimbus, is analogous to the dialectic negotiation of privacy boundaries.

This paper continues where Boyle and Greenberg left this discussion, trying to give formal semantics to such a conception of privacy and awareness. The model we introduce in this paper is based on Rodden’s abstract version of the focus-nimbus model. We show how this model can provide a sound basis for describing mathematically the design space of awareness systems, in terms of the content exchanged, elementary user behaviors pertaining to sharing information about themselves or perceiving information about others. The sections that follow shall introduce the model and demonstrate how some principles for the protection of user privacy can be expressed succinctly, lending clarity and conciseness to the discussion of awareness systems and their design.

## 2 Model Overview

Where the original focus/nimbus model describes *how much* aware two entities are about each-other in a particular *space*, our model describes *what* are the entities aware of regarding each-other in a particular *situation*. The model we propose is an extension of the focus/nimbus model, populated with the notions of *entities*, *aspects*, *attributes*, *resources* and *observable items*. These notions are introduced below with the help of the following scenario:

*“John and Anna are seniors living alone; sometimes they invite each other for a walk. They like to do this easily and without social pressure on each other so they recently, installed a system that helps them convey their wish for a walk. When they feel like walking, they can flick a switch installed in their living room; the system indicates their intentions to the other side by lighting a small lamp in a visible position in the living room.”*

**Entities** are representations of actors, communities, and other agents (possibly artificial) within an awareness-system. The actors of the above scenario (i.e. *John* and *Anna*) are represented in an awareness system with the corresponding entities.

**Aspects** are any characteristics that refer to an entity’s state. An aspect is actually the complement to the incomplete-statement “*I want to be aware of your ...*”. In our scenario “*John wants to be aware of Anna’s wish for a walk*”; thus the phrase “*wish for a walk*” is an aspect, i.e. a characteristic of *Anna*’s state that may be shared with *John*. The notion of aspect is broad and loose enough encompassing more detailed terms like “location”, “activity”, “aspirations”, or even “focus”, and “nimbus”.

**Attributes** are the place-holders in our model for the information exchanged between *Entities*. An attribute can be thought of as a potential answer to the request “*Tell me something about your ‘X aspect’*”. In our scenario an answer to *John*’s request “*Anna tell me something about your ‘wish for walk’*” could be “*My ‘wish for walk’ is moderate*”; thus the answer “*My ‘wish for walk’ is moderate*” is an attribute, binding the value “*moderate*” to the aspect “*wish for walk*”.

In any situation an entity makes its state available to other entities using one or more attributes. Awareness though is dynamic. One’s **nimbus** is populated with *attribute-providers*; i.e. functions that return those attributes that one makes available to other entities in a specific situation.

A **resource** is a binding of an *aspect* with a way of displaying one or more attributes about this aspect. In any situation an entity might employ one or more resources to express its interest about certain aspects of other entities. Roughly speaking a *resource* is a statement such as “*I shall display the attributes you provide to me about your ... by ...*”. In our example, “*John plans to display the attributes that Anna provides to him about her wish for walk by turning the lamp either on or off*”.

One’s **focus** is populated with *resource-providers*; i.e. functions that return one’s resources that are engaged to display information about other entities in a specific situation.

An **observable item** is the result of displaying some attributes about an aspect using a *resource*. Roughly speaking an *observable item* contains the answer to the question “*How are these attributes displayed to you?*”. In our scenario a possible answer to the question “*How is ‘moderate wish for walk’ displayed to you?*” could be “*by dimming the light on my desk*”.

The negotiation of the reciprocal *foci* and *nimbi* of two entities in a given situation (i.e. the corresponding ‘produced’ *attributes* and *resources*) is a function which returns the *observable-items* that are displayed to the two entities about each other’s states, effectively characterizing their reciprocal awareness.

In the above scenario, *John* indicates his wish to go for a walk to *Anna* using the *walk-switch*. We can consider that *John’s* *Nimbus* contains an *Attribute-Provider* that returns (in any situation) an attribute about *John’s* wish for walk based on the state of the *walk-switch*. On the other hand, *Anna* can check *John’s* wish-for-walk by watching the corresponding *lamp*. System-wise we can consider that *Anna’s Focus* is expressed via a resource that switches the lamp on/off depending on *John’s* wish for walk. Needless to say that neither the *walk-switch* nor the *lamp* imply necessarily that *John* does actually wish to walk (he may forget to push the switch) or that *Anna* does notice the lamp (their respective actual/inherent nimbus and focus). However, we can imagine that *Anna* can unplug the lamp or even “assign” it to another person. So, *Anna* becomes aware of *John’s* mood for walk, by manipulating her *focus*. Similarly, we can imagine that *John* could choose not to let *Anna* know about the state of the *walk-switch*, thus *John* lets *Anna* become aware of his mood for walk by manipulating his *nimbus*.

### 3 Observable Items and Awareness

“*John is sitting on his sofa reading a magazine. Behind him, on his desk the walk-lamp illuminates indicating that his friend Anna feels like going for a walk.*”

In this situation the illuminating-lamp is an *Observable Item* that indicates to *John* whether *Anna* wants to go for a walk. It should be stressed here that by the term *observable* we do not imply that *John* is seeing the lamp or even whether *John* perceives it as an indication for *Anna’s* wish to go for a walk. We only stipulate that the lamp is available for observation, and that it is possible (in principle) for *John* to perceive. *John’s* lamp may be switched-on whether he is looking at it or not. We should also stress that the term *observable* does not imply a modality; information may be presented in any perceivable manner (auditory, visual, tactile, etc.).

Taking the above example as a basis, we can assert that in any situation there is a set of observable *items* that a given entity can observe. In the context of an awareness system we can consider that an entity *i* becomes aware about the state of entity *j* through an awareness-characteristic function  $a_{ij}$  which under a given situation *r* returns the set of observable by entity *i* items that present information regarding entity *j*:

$$\forall i,j:Entity; a_{ij}:RealSituation \rightarrow \mathbb{F} ObservableItem;$$

In this section and elsewhere *RealSituation* is an abstraction that we use to encapsulate the dynamic nature of the universe to which awareness refers. The model itself is neutral regarding the notion of reality; the model and the user-related properties in the following sections do not make any assumptions about what is “real”.

The exact semantics of  $a_{ij}$  will be shaped out, as we advance in the notions of focus, and nimbus. For convenience, we use  $a_{ij}^r$  to denote  $a_{ij}(r)$ .

As an example of an *ObservableItem* we can consider a function that returns an *ObservableItem* (light illumination):

$$lightIllumination: Lamp \times Voltage \rightarrow ObservableItem;$$

We will not define the function *lightIllumination* in detail but we can imagine that this function returns the effect of applying the specified voltage on a lamp source. For example *lightIllumination(lamp1,240V)* represents an observable item that originates from applying 240Volts on *lamp1*.

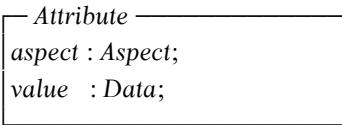
In the aforementioned scenario we can state that

$$a^r_{John, Anna} = \{ lightIllumination(lamp1,240V) \}$$

i.e. the *awareness* of *John* about *Anna* in a situation(*r*) is a set that includes one observable item that indicates *Anna*'s wish to walk by illuminating *lamp1*. Note that it would be more appropriate to say "potential awareness", since we have no information about *John*'s physical (inherent) focus. For brevity, we use instead the term "awareness" and we imply a corresponding interpretation for statements such as "*John is aware of Anna's wish for a walk*".

### 4 Attributes, Attribute Providers and Nimbus

Nimbus represents a sub-space across which an entity makes its state available to others. We can consider that in any real situation an entity's state(as it is presented to other entities) holds information about a wide range of aspects; we use the scheme "Attribute" to describe a piece of information("value") about an aspect("aspect").



For convenience, we use the idiom (*a:v*) to denote the attribute

$$\langle aspect \rightsquigarrow a, value \rightsquigarrow v \rangle, \text{ i.e. the attribute about aspect } a \text{ with value } v$$

There may be more than one attributes about the same aspect for a single entity; for example one's state may include an attribute about "location" with value "home"(location:home), and an other attribute also about "location" with the value "kitchen" (location: kitchen). Notice that the model does not preclude that one's state may include contradictory attributes (allowing for imperfect technology or intentional misinformation by the user).

One's attributes and the entities that they are available to may change over time. We define a function-type *AttributeProvider*, that when applied to a real situation returns an attribute and the set of entities that this attribute is made available to. Hence, an attribute provider may return different attributes available to different entities depending on the situation:

$$AttributeProvider ::= RealSituation \rightarrow (Attribute \times \mathbb{F} Entity)$$

For an instance of *AttributeProvider* *p* we use *p<sup>r</sup>* to denote *first p(r)* and *p<sup>r</sup>.e* to denote *second p(r)*; i.e. *p<sup>r</sup>* denotes the attribute that *p* returns at situation *r*, and *p<sup>r</sup>.e* denotes the set of entities that *p<sup>r</sup>* is made available to.



For each entity  $i$  we assume that  $nimbus_i$  includes all the entity's  $i$  attribute providers:

$$\forall i:Entity; nimbus_i : \mathbb{F} AttributeProvider$$

Given  $nimbus_i$ , we can define a function  $n_{ij}$  such that when applied to a real situation it returns the attributes of  $i$  that are available to  $j$ :

$$\forall r:RealSituation; i,j:Entity; n_{ij} : RealSituation \rightarrow \mathbb{F} Attribute \mid n_{ij}(r)=\{a: Attribute \mid (\exists p:AttributeProvider; p \in nimbus_i \bullet (a=p^r) \wedge (j \in p^r.e))\}$$

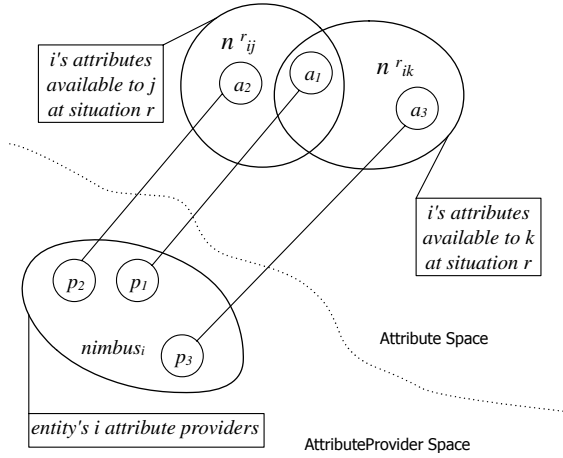


Fig. 1. The nimbus of entity  $i$  to entities  $j$  and  $k$

Figure 1 shows three attribute providers of entity  $i$  ( $p_1, p_2, p_3$ ), and their corresponding attributes in a situation  $r$  (i.e.  $a_1, a_2, a_3$ ). Attribute provider  $p_2$ , makes attribute  $a_2$  available to entity  $j$ ;  $p_1$  makes  $a_1$  available to entities  $j$ , and  $k$ ;  $p_3$  makes  $a_3$  available to entity  $k$ . Consequently the nimbus of entity  $i$  to  $j$  at this situation is  $n_{ij}^r=\{a_1,a_2\}$  and the nimbus of entity  $i$  to  $k$  at this situation is  $n_{ik}^r=\{a_1,a_3\}$ .

Previously it was noted that the model does not preclude that one's state may include contradictory attributes. For example an attribute about *location* with value *home* (*location: home*), contradicts the attribute (*location: away*). We populate the attribute space with a relationship that denotes contradicting attributes:

$$\begin{aligned} \_contradicting\_ : Attribute &\leftrightarrow Attribute; \\ \forall a,b: Attribute; a \text{ contradicting } b &\iff (a,b) \in \_contradicting\_; \end{aligned}$$

It was also noted that there may be more than one attributes about the same aspect for a single entity. Furthermore, one may agree that an attribute( $a_1$ ) about aspect "activity" with a value "sleeping" implies an attribute( $a_2$ ) about aspect "location" with a value "bed", and the latter may imply an attribute( $a_3$ ) about "location" with value "home" and so on. The exact ontological relationships and whether an ontology can be global, or application specific, or entity specific, or moreover situation-specific is out of the context of this paper. However given an ontological relationship between attributes:

$$\_implies\_ : Attribute \times Attribute$$

We can define a function that returns all possible attributes that are implied from a single attribute:

$$\begin{aligned} & \text{impliedAttributes} : \text{Attribute} \rightarrow \mathbb{F} \text{Attribute} ; \\ \forall a : \text{Attribute}; & \text{impliedAttributes}(a) = \{u : \text{Attribute} \mid (a, u) \in \_ \text{implies} \_ \} \\ & \text{where } \_ \text{implies} \_ \text{ is the reflexive transitive closure of } \_ \text{implies} \_ \end{aligned}$$

More generally we can take into account implications from attribute tuples, triads, quads, or from any set of attributes; we assume that the “*impliedAttributes*” function is extended to return all attributes implied from a set of attributes:

$$\text{impliedAttributes} : \mathbb{F} \text{Attribute} \rightarrow \mathbb{F} \text{Attribute} ;$$

The exact definition of this extensive function is out of scope; given its existence however, we can define  $n^*_{xy}$  to return all implied attributes of  $n^r_{xy}$ .

$$\forall r : \text{RealSituation}; n^*_{ij} = \{a : \text{Attribute} \mid a \in \text{impliedAttributes}(n^r_{ij})\}$$

#### 4.1 Nimbus Example

We can reflect on the nimbi of *John* and *Anna* in the scenario introduced earlier; *John* lets *Anna* know if he feels like walking by turning the switch on/off. In terms of the system *John* makes available to *Anna* in any situation  $r$ , an attribute  $a$  ( $a \in n^r_{\text{John}, \text{Anna}}$ ) about his “*wishforWalk*”. *John*’s nimbus contains an attribute provider that in any real situation returns the aforesaid attribute, and adjusts the attribute’s value according to the state of the switch:

$$\begin{aligned} & \text{sw1} : \text{AttributeProvider}; \text{sw1} \in \text{nimbus}_{\text{John}} \mid \forall r : \text{RealSituation}; \\ & (\text{sw1}^r.\text{aspect} = \text{wishforWalk}) \wedge \\ & (\text{sw1}^r.\text{value} = \text{if } \text{switchclosed}(\text{switch1}, r) \text{ then } \text{true} \text{ else } \text{false}) \wedge (\text{sw1}^r.e = \{\text{Anna}\}) \end{aligned}$$

Thus, *sw1* is an attribute provider in *John*’s nimbus, which when applied in a situation  $r$  it returns an attribute ( $\text{sw1}^r.\text{aspect} : \text{sw1}^r.\text{value}$ ) and an entity set ( $\text{sw1}^r.e$ ) that includes *Anna*. The attribute’s aspect is *wishforWalk* and its value is either *true* or *false* (depending on the state of *switch1*).

Now we can wrap up *John*’s nimbus ( $\text{nimbus}_{\text{John}}$ )

$$\text{nimbus}_{\text{John}} = \{\text{sw1}\}$$

Using the definition of  $n_{ij}$  we can verify that:

$$\forall r : \text{RealSituation}; n^r_{\text{John}, \text{John}} = \emptyset; n^r_{\text{John}, \text{Anna}} = \{\text{sw1}^r\};$$

Similarly for *Anna* and her installation:

$$\begin{aligned} & \text{sw2} : \text{AttributeProvider}; \text{sw2} \in \text{nimbus}_{\text{Anna}} \mid \forall r : \text{RealSituation}; \\ & (\text{sw2}^r.\text{aspect} = \text{wishforWalk}) \wedge \\ & (\text{sw2}^r.\text{value} = \text{if } \text{switchclosed}(\text{switch2}, r) \text{ then } \text{true} \text{ else } \text{false}) \wedge \\ & (\text{sw2}^r.e = \{\text{John}, \text{Anna}\}) \end{aligned}$$

*Anna*’s nimbus will be

$$\text{nimbus}_{\text{Anna}} = \{\text{sw2}\}$$

Using the definition of  $n_{ij}$  we can verify that:

$$\forall r : \text{RealSituation}; n^r_{\text{Anna}, \text{John}} = \{\text{sw2}^r\}; n^r_{\text{Anna}, \text{Anna}} = \{\text{sw2}^r\};$$

Note that *Anna*’s “*wishForWalk*” is available both to *John* and to herself, in contrast with *John* who makes available his “*wishForWalk*” only to *Anna*. This may sound awkward, however it points-out the fact that an entity is-not/can-not-be de facto aware of the information that is collected about it and made available to others (it might not be aware, e.g., in case of covert surveillance). Further this observation points out that *nimbus* does not imply a physical location or ownership of the underlying attribute providers.

## 5 Resources, Resource-Providers and Focus

Focus represents a sub-space within which an entity focuses its attention. System-wise we assume that an entity has a limited set of resources to represent the provided attributes regarding aspects of other entities. The scheme *Resource* describes an aspect of interest and a function that transforms the corresponding attributes to an observable item.

<p style="margin: 0;"><i>Resource</i></p> <p style="margin: 0;"><i>aspect</i> : <i>Aspect</i>;</p> <p style="margin: 0;"><i>render</i> : <math>\mathbb{F}</math> <i>Attribute</i> <math>\rightarrow</math> <i>ObservableItem</i>;</p>
---

Note that an entity may assign more than one resources that render the same aspect(s) of another entity. E.g., *John* can render *Anna*’s *wishForWalk* both by a lamp at home and an icon on his mobile phone.

One’s resources may change depending on the situation; to incorporate this in the model we define a function-type *ResourceProvider*, that when applied to a real situation returns a resource and an entity that it is assigned to. Hence, a single resource provider may return different resources assigned to different entities depending on the situation:

$$\text{ResourceProvider} ::= \text{RealSituation} \rightarrow (\text{Resource} \times \text{Entity})$$

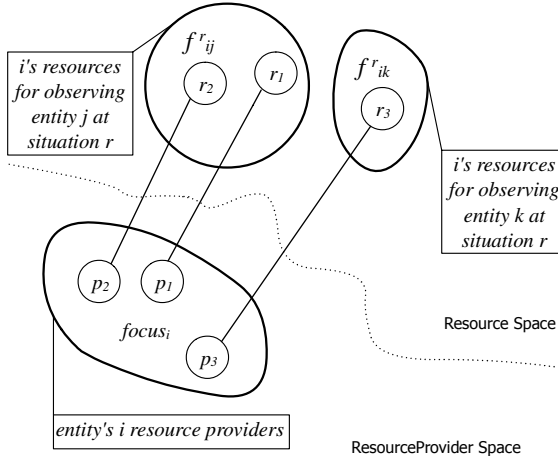
For a *ResourceProvider* instance  $p$  we use  $p^r$  to denote *first*  $p(r)$  and  $p^r.e$  to denote *second*  $p(r)$ . Hence  $p^r$  denotes the resource that  $p$  returns at the situation  $r$ , and  $p^r.e$  denotes the entity that  $p^r$  is assigned to. For each entity we assume that  $\text{focus}_i$  includes the set of entity’s  $i$  resource providers.

$$\forall i:\text{Entity}; \text{focus}_i : \mathbb{F} \text{ResourceProvider}$$

Given  $\text{focus}_i$  we define  $f_{ij}$  to return only those resources of  $i$  that focus on entity  $j$ :

$$\forall r:\text{RealSituation}; \forall i,j:\text{Entity}; f_{ij} : \text{RealSituation} \rightarrow \mathbb{F} \text{Resource} \mid f_{ij}(r) = \{c : \text{Resource} \mid (\exists p:\text{ResourceProvider}; p \in \text{focus}_i \bullet (c = p^r) \wedge (j = p^r.e))\}$$

In figure 2 we can notice on the bottom left three resource providers of entity’s  $i$  focus (i.e.  $p1$   $p2$   $p3$ ), and their corresponding resources in a situation  $r$  (i.e.  $r1, r2, r3$ ). The resource provider  $p1$ , assigns the resource  $r1$  to display information from entity  $j$ ;  $p2$  assigns  $r2$  to  $j$ ;  $p3$  assigns  $r3$  to  $k$ . Consequently the focus of entity  $i$  on  $j$  at this situation is  $f^r_{ij} = \{r1, r2\}$  and the focus of entity  $i$  on  $k$  at this situation is  $f^r_{ik} = \{r3\}$ .



**Fig. 2.** Focus of entity *i* upon entities *j* and *k*

### 5.1 Focus Example

Continuing our example, imagine that “*John uses a lamp to display Anna’s wish for a walk and vice versa*”. A lamp (resource) is assigned to display Anna’s *wishForWalk*. System wise, John’s focus on Anna contains a resource  $r$  ( $r \in f^{r_{John,Anna}}$ ) that renders attributes about the aspect “*wishforWalk*”. John’s focus ( $focus_{John}$ ) contains a resource provider, that returns this resource and adjusts the resource’s rendering (*illumination*) according to the attributes that the system provides:

$$\begin{aligned}
 wr1: & \text{ResourceProvider}; wr1 \in focus_{John} \mid \forall r:RealSituation; \\
 & (wr1^r.aspect = wishForWalk) \wedge \\
 & (\forall s:F \text{Attribute}; wr1^r.render(s) = \\
 \text{if } & (\exists p:Attribute; p \in s \mid p.aspect = wishForWalk \wedge p.value = true) \text{ then} \\
 & \text{lightIllumination}(lamp1, 240V) \text{ else } \text{lightIllumination}(lamp1, 0V)) \wedge \\
 & (wr1^r.e = Anna)
 \end{aligned}$$

Thus  $wr1$  is a *ResourceProvider* that returns a resource which renders attributes about *wishforWalk* either by turning on *lamp1* or by turning it off;  $wr1.e$  denotes that the returned resource should be assigned to *Anna*. Consequently,  $wr1$  is a resource provider in *John’s* focus, that when applied to a real situation  $r$ , it returns a resource that can render *Anna’s* *wishforWalk*.

We can wrap up John’s focus ( $focus_{John}$ ):

$$focus_{John} = \{ wr1 \}$$

We can apply the definition of  $f_{ij}$  to verify:

$$\forall r:RealSituation; f^{r_{John,John}} = \emptyset; f^{r_{John,Anna}} = \{ wr1^r \};$$

Similarly we can describe *Anna’s* focus on *John’s* wish for walk.

$$\begin{aligned}
 wr2: & \text{ResourceProvider}; wr2 \in focus_{Anna} \mid \forall r:RealSituation; \\
 & (wr2^r.aspect = wishForWalk) \wedge
 \end{aligned}$$

$(\forall s:\mathbb{F} \text{Attribute}; wr2^r.render(s) =$   
*if*  $(\exists p:\text{Attribute}; p \in s \mid p.aspect = wishForWalk \wedge p.value = true)$  *then*  
 $lightIllumination(lamp2,240V)$  *else*  $lightIllumination(lamp2,0V)$   $) \wedge$   
 $(wr2^r.e = John)$

Consequently *Anna's* focus will be:

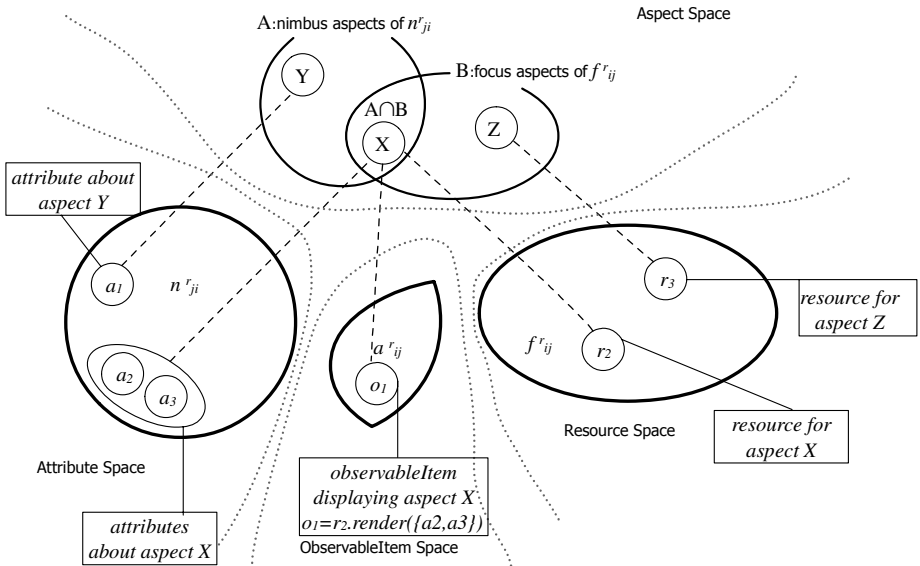
$$focus_{Anna} = \{ wr2 \}$$

We can apply the definition of  $f_{ij}$  to verify:

$$\forall r:\text{RealSituation}; f^r_{Anna, Anna} = \emptyset; f^r_{Anna, John} = \{ wr2^r \};$$

### 6 Focus/Nimbus Negotiation and Awareness-Systems

Figure 3 shows the attributes that an entity “*j*” makes available to an entity “*i*” at a situation “*r*” (i.e.  $a1, a2, a3$ ) through  $n^{r}_{ji}$ . On the top-left we see their projection (A) on the *Aspect Space* i.e. the aspects they refer to. For example the attribute  $a_1$  contains information about aspect *Y*, so its projection on the *aspect space* is *Y*. We notice also the resources that *i* assigns for observing *j* at *r* (i.e.  $r1, r2$ ) through  $f^{r}_{ij}$  and the resource projection (B) on the *Aspect Space*; i.e. the aspects that the resources claim to (i.e. are set to) render. For example, the resource  $r_2$  claims to render the aspect *X*, so its projection on the aspect space is *X*. The intersection  $A \cap B$ , represents the aspects that *i* wants to observe about *j*, and *j* is making available to *i* at the situation *r*. Consequently, the set of items that *i* can observe about *j* ( $a^{r}_{ij}$ ), are the result of rendering those attributes of  $n^{r}_{ji}$  that project on  $A \cap B$  (i.e.  $a_2$ , and  $a_3$ ), using those corresponding resources of  $f^{r}_{ij}$  that project on  $A \cap B$  (i.e.  $r1$ ); therefore (see bottom of figure 3)  $a^{r}_{ij}$  includes the observable item  $o1 = r2.render(\{a2, a3\})$ .



**Fig. 3.** Illustration of focus/nimbus negotiation and awareness that entity *i* has of entity *j*

We generalize this negotiation of the reciprocal foci, and nimbi between two entities as follows:

$$a_{ij} ::= \text{RealSituation} \rightarrow \mathbb{F} \text{ObservableItem};$$

$$\forall r: \text{RealSituation};$$

$$a_{ij}(r) = \{v : \text{ObservableItem} \mid (\forall c: \text{Resource}; c \in f_{ij}^r \bullet$$

$$v = c.\text{render}(\{u:\text{Attribute} \mid (u \in n_{ji}^r) \wedge (u.\text{aspect}=c.\text{aspect})\}))\}$$

Returning to our example, John's observable items about Anna's state is the result of rendering the value of Anna's *wishforWalk* as it is provided to John (i.e.  $sw2^r$ ) using the resource that John assigned for this purpose (i.e.  $wr1^r$ ). Conversely, Anna's observable items about John's state is the result of rendering the value of John's *wishforWalk* as it is provided to Anna (i.e.  $sw1^r$ ) using the resource that Anna assigned for this purpose (i.e.  $wr2^r$ ). On the other hand both  $a_{John,John}^r$ , and  $a_{Anna,Anna}^r$  are empty sets, since John's nimbus to himself is an empty set, and in the case of Anna, although her *wishforWalk* is available to her-self, there is no resource assigned to render it:

$$\forall r: \text{RealSituation};$$

$$a_{John,Anna}^r = \{wr1^r.\text{render}(\{sw2^r\})\}; a_{John,John}^r = \emptyset;$$

$$a_{Anna,John}^r = \{wr2^r.\text{render}(\{sw1^r\})\}; a_{Anna,Anna}^r = \emptyset;$$

At this point we can wrap together the definitions so far in a scheme that describes an awareness system. The scheme defines the set of entities in a system, their nimbi and foci, as well as their reciprocal awareness information using the definitions we have introduced so far:

AwarenessSystem	
<i>entities</i>	: $\mathbb{F} \text{Entity}$ ;
<i>nimbus</i>	: $\text{Entity} \rightarrow \mathbb{F} \text{AttributeProvider}$ ;
<i>focus</i>	: $\text{Entity} \rightarrow \mathbb{F} \text{ResourceProvider}$ ;
$n$	: $(\text{Entity} \times \text{Entity}) \rightarrow (\text{RealSituation} \rightarrow \mathbb{F} \text{Attribute})$ ;
$f$	: $(\text{Entity} \times \text{Entity}) \rightarrow (\text{RealSituation} \rightarrow \mathbb{F} \text{Resource})$ ;
$a$	: $(\text{Entity} \times \text{Entity}) \rightarrow (\text{RealSituation} \rightarrow \mathbb{F} \text{ObservableItem})$ ;
<hr/>	
$\text{dom } \textit{nimbus} = \textit{entities}; \text{ dom } \textit{focus} = \textit{entities};$	
$\forall r: \text{RealSituation}; i, j: \text{Entity}; i, j \in \textit{entities};$	
$\forall u: \text{Attribute}; c: \text{Resource}; v: \text{ObservableItem}$	
$r \mapsto u \in n_{ij}$	$\Leftrightarrow \exists p: \text{AttributeProvider}; p \in \textit{nimbus}_i \mid (u=p^r) \wedge (j \in u.\textit{access})$
$r \mapsto c \in f_{ij}$	$\Leftrightarrow \exists p: \text{ResourceProvider}; p \in \textit{focus}_i \mid c=p^r \wedge (j = c.\textit{entity})$
$r \mapsto v \in a_{ij}$	$\Leftrightarrow \exists p: \text{Resource}; p \in f_{ij}^r \mid$
$v = p.\textit{render}(\{u:\text{Attribute} \mid (u \in n_{ji}^r) \wedge (u.\textit{aspect}=c.\textit{aspect})\}))$	

We use the idioms  $\textit{nimbus}_i$  for  $\textit{nimbus}(i)$ ,  $\textit{focus}_i$  for  $\textit{focus}(i)$ ,  $n_{ij}$  for  $n(i,j)$ ,  $f_{ij}$  for  $f(i,j)$ ,  $a_{ij}$  for  $a(i,j)$ ,  $n_{ij}^r$  for  $n(i,j)(r)$ ,  $f_{ij}^r$  for  $f(i,j)(r)$ ,  $a_{ij}^r$  for  $a(i,j)(r)$ .

In the following sections we will demonstrate how the introduced model allows us to reason about some privacy related properties of awareness systems.

## 7 Plausible Deniability

The term plausible deniability has been often used (e.g., see [3],[1]) to describe how users of communication systems may rely on ambiguity in order to have a plausible excuse for avoiding communication or interaction with a third party.

Price et al. [3] explore the social need for plausible deniability in ubicomp systems and in relation with one’s location and identity. As they point out, many systems depart from social norms that are otherwise present in face-to-face interactions (where a person can easily see whether he/she is being observed by others). Price et al classify five types of user controlled “noise” to protect location privacy (*Anonymizing, Hashing, Cloaking, Blurring, and Lying*).

In a similar line, Lederer et al. [16] report that people decide to disclose information about their activities and location based on the identity of the requester and the situation in which it happens. Consolvo et al. [7] introduce several requirements for location-aware applications. Among these they mention the need to support denial (e.g. the ability not to disclose any information), and deception (e.g. the ability to deceive in the response). In their studies, blurring (i.e. the ability to disclose less specific information) was encountered less frequently. Summarizing, we can identify three basic deceptive patterns:

- *Deception/Lying*: intentionally false information
- *Denial/Cloaking*: no information disclosure
- *Blurring/Evasion*: revealing part of the information

These are discussed below in terms of the model of awareness introduced.

### 7.1 Deception / Lying

Lying can be thought as giving intentionally false information about an aspect. We consider that an entity is lying when it is giving to some other entity contradicting information compared to itself about an aspect.

For example, consider an entity “a” that makes available to itself an attribute (*location: home*) whereas it makes available to entity “b” an attribute (*location: away*). Given that (*location: home*) is contradictory to (*location: away*) we can state that “a” is lying to “b” about its *location*.

Bearing in mind a simple ontology like the one we described earlier, if entity “a” would make (*activity:sleeping*) available to it-self then the predicate “a is lying to b about its location” would still hold since in the context of the specific ontology, the attribute (*activity: sleeping*) implies (*location: home*) which contradicts to the attribute (*location: away*). Following the above we can formalize deception/lying:

$$\begin{aligned} \_isLyingTo\_About\_ : RealSituation &\rightarrow \mathbb{P} (Entity \times Entity \times Aspect) \\ &\forall r:RealSituation; x, y Entity; a:aspect; \\ x \text{ isLyingTo } y \text{ About } a (r) &\Leftrightarrow (x, y, a) \in \_isLyingTo\_About\_ (r) \Leftrightarrow \\ \exists u, v:Attribute \mid u \in n_{xy}^* &\wedge v \in n_{xx}^* \wedge u.aspect=a \wedge u \text{ contradicting } v \end{aligned}$$

i.e.,  $x$  is lying to  $y$  about an aspect  $a$ , when there is at least one attribute about  $a$  that is made available to  $y$  (explicitly or by implication), such that it contradicts with an attribute that  $x$  makes available to him/her-self (explicitly or by implication).

## 7.2 Denial / Cloaking

Price describes “Cloaking” as the ability to hide one’s location or identity. More generally, cloaking can concern any aspect of one’s nimbus. Hence we consider cloaking as the ability to conceal any attributes about an aspect of an entity from another entity.

For example, consider an entity “*a*” that makes no attributes available to entity “*b*” about its *location*, where as it makes available to an entity “*c*” an attribute (*location: home*). We can say in this example that *a* is hiding its *location* from *b*.

Taking in account a simple ontology like the one described earlier, we could say that even if only an attribute (*activity:sleeping*) would be available to entity “*c*” the predicate “*a* is hiding its *location* from *b*” would still hold since in the context of the specific ontology, (*activity: sleeping*) implies several attributes about *location* such as (*location: bedroom*) and (*location: home*). Therefore in the formal definition that follows we use  $n_{xy}^*$  which actually contains all the possible implied attributes of  $n_{xy}^r$ .

$$\begin{aligned} \_isHiding\_From\_ : RealSituation &\rightarrow \mathbb{P}(Entity \times Aspect \times Entity) \\ &\forall r:RealSituation; x, y Entity; a:aspect; \\ x \text{ isHiding } a \text{ from } y (r) &\Leftrightarrow (x, a, y) \in \_isHiding\_From\_ (r) \Leftrightarrow \\ &\exists z: Entity \mid (\exists u:Attribute ; u \in n_{xz}^* \wedge u.aspect=a) \wedge \\ &\quad \neg(\exists u:Attribute; u \in n_{xy}^* \wedge u.aspect=a) \end{aligned}$$

i.e., *x* is hiding an aspect *a* from *y*, when there are no attributes about *a* that are made available to *y* either explicitly or by implication, and at the same time there is at least one attribute about *a* that *x* makes available to an other entity *z*. Note that *z* can be any entity including *x* it-self.

## 7.3 Blurring / Evasion

In contrast with *Cloaking*, *Blurring* is not hiding an aspect, but rather it concerns withholding information. Price describes “*blurring*” as the ability to decrease the precision of one’s location. In a wider context we can replace “location” with any aspect of one’s nimbus. To account with the term “decrease” we define “blurring” in comparison to a reference entity. Hence we consider that an entity is *blurring* information about an aspect to another entity, when the first is revealing *less information about this aspect* to the latter than a reference entity.

Before proceeding to a formal definition let’s consider the phrase “*less information about an aspect*”. This phrase implies that we need to take in account the term “*information about an aspect*”. For that, we introduce a function *attributesAbout*, that when applied on a set of attributes and an aspect, it returns only those attributes that concern the specified aspect:

$$\begin{aligned} attributesAbout : \mathbb{F} Attribute \times Aspect &\rightarrow \mathbb{F} Attribute \\ \forall s: \mathbb{F} Attribute; a: Aspect; attributesAbout(s,a) &= \{u:Attribute; u \in s \mid u.aspect=a\} \end{aligned}$$

To evaluate the expression “*less information*” we consider that if an attribute-set *s* is a subset of an attribute-set *t*, then the set *s* contains less information than the set *t*. For example a set that includes an attribute about *location* with value *home* (*location: home*) contains less information than the set  $\{(location: home), (location: bedroom)\}$  since the first set a subset of the latter.



Taking in account a simple ontology like the one described earlier, we can tell that the set  $\{(location: home)\}$  contains less information than the set  $\{(location: bedroom)\}$ , since the latter implies the first. Moreover  $\{(location: home)\}$  contains less information than the set  $\{(activity: sleeping)\}$  since the latter implies both  $(location: bedroom)$  and  $(location: home)$ . Consequently in the formal definition that follows we use  $n^*_{xy}$  which actually contains all the possible implied attributes of  $n^r_{xy}$ .

$$\begin{aligned} \_isBlurring\_to\_ : RealSituation &\rightarrow \mathbb{P}(Entity \times Aspect \times Entity) \\ &\text{let } x, y: Entity; a: Aspect; r: RealSituation; \\ x \text{ isBlurring } a \text{ to } y (r) &\Leftrightarrow (x, a, y) \in \_isBlurring\_to\_ (r) \Leftrightarrow \\ \exists z: Entity \mid \text{attributesAbout}(n^*_{xy}, a) &\subset \text{attributesAbout}(n^*_{xz}, a) \end{aligned}$$

i.e.  $x$  is blurring information about an aspect  $a$  to  $y$ , when all the attributes about  $a$  that are made available to  $y$  (explicitly or by implication), are a subset of the attributes about  $a$  that are made available to an entity  $z$  (explicitly or by implication). Note that the reference entity  $z$  can be any entity including  $x$  itself.

## 8 Discussion on Physical/Inherent Awareness

So far we have considered observable items without taking into account whether physical entities (such as actors) are indeed physically (inherently) aware of them. This is a point where one can utilize the quantitative notion of modeling awareness with Rodden’s focus/nimbus model. We can actually consider that each *observableItem* has an inherent/physical nimbus, and each *entity* has an inherent focus. The composition of an entity’s inherent focus with an observable item’s inherent nimbus defines how aware an actor is of the observable item it self. If we assume that a system has sufficient resources/capabilities to apply Rodden’s focus-nimbus model in the *Entity-ObservableItem* relationship (i.e. we can define the focus/nimbus composition), then we can reason in detail about the information (observable-items) that one is aware of.

For that we may consider a function  $n^+$  that associates an *ObservableItem* with its inherent nimbus in any situation, a function  $f^+$  that associates an *Entity* with its inherent focus in any situation, and an awareness quantifier function  $a^+$  :

$$\begin{aligned} n^+ : RealSituation \times ObservableItem &\rightarrow InherentNimbus; \\ f^+ : RealSituation \times Entity &\rightarrow InherentFocus; \\ a^+ : InherentFocus \times InherentNimbus &\rightarrow InherentAwareness \end{aligned}$$

For an *entity*  $x$ , and an *observableItem*  $u$ ,  $a^+(f^+(r,x), n^+(r,u))$  quantifies the question “How aware is entity  $x$  of observable item  $u$  at situation  $r$ ”. Using a predefined threshold  $h$  we can state that  $x$  is aware of  $u$  at situation  $r$  when its inherent awareness  $a^+(f^+(r,x), n^+(r,u))$  is greater than the predefined threshold:

$$\begin{aligned} \_isPhysicallyAwareOf\_ : RealSituation &\rightarrow (Entity \times ObservableItem) \bullet \\ &\forall x: Entity; u: ObservableItem; r: RealSituation; \\ x \text{ isPhysicallyAwareOf } u (r) &\Leftrightarrow (x, u) \in \_isPhysicallyAwareOf\_ (r) \Leftrightarrow \\ &a^+(f^+(r,x), n^+(r,u)) > h \end{aligned}$$

Now we can define intentionally/unintentionally perceived awareness information; we can consider that entity  $x$  is intentionally aware of an observable item  $u$  when an  $x$  is aware of  $u$ , and  $u$  is one of the items that are generated through the system for that entity:

$$\begin{aligned} & \_isIntentionallyAwareOf \_ : RealSituation \rightarrow (Entity \times ObservableItem) \bullet \\ & \quad \forall r:RealSituation; x:Entity; u:ObservableItem \bullet \\ x \ isIntentionallyAwareOf \ u(r) & \Leftrightarrow (x,u) \in \_isIntentionallyAwareOf\_ (r) \Leftrightarrow \\ & \quad (x \ isPhysicallyAwareOf \ u(r)) \wedge (\exists y:Entity \mid u \in a_{xy}^r) \end{aligned}$$

Similarly we can consider that entity  $x$  is unintentionally aware of an observable item  $u$  when an  $x$  is aware of  $u$ , but  $u$  is not anyone of the items that are generated through the system for that entity:

$$\begin{aligned} & \_isUnintentionallyAwareOf \_ : RealSituation \rightarrow (Entity \times ObservableItem) \bullet \\ & \quad \forall r:RealSituation; x:Entity; u:ObservableItem \bullet \\ x \ isUnintentionallyAwareOf \ u(r) & \Leftrightarrow (x,u) \in \_isUnintentionallyAwareOf\_ (r) \Leftrightarrow \\ & \quad (x \ isPhysicallyAwareOf \ u(r)) \wedge \neg (\exists y:Entity \mid u \in a_{xy}^r) \end{aligned}$$

One may doubt the feasibility of computing functions  $n^+$ ,  $f^+$ , and  $a^+$  as they refer essentially to cognitive phenomena. Yet, one's focus may be approximated with varying degrees of success by knowing whether they are present in front of the computer, or even further, monitoring their head pose or even their eye-gaze. In other words, an entity's nimbus can approximate its inherent focus allowing reasonable approximations of  $n^+$ ,  $f^+$ , and  $a^+$ . In our scenario a weight-sensor on a chair facing the lamp could be included in John's nimbus for some reason (e.g. to notify Anna about John's presence). Whether John is aware of the lamp is more likely when he sits on the chair, although not certain (he might have his eyes closed or be day-dreaming).

Although we can define a relationship that relates observable-items with the attribute(s) that they present successfully, we can not assume that if an entity is physically aware of an observable item, that the entity is also physically aware of the presented attribute(s), since we do not model the cognitive processes of awareness (e.g., the lamp can display *Anna's* wish-for-walk, *John* can be physically aware of the lamp, but still John at the same time may be unaware of *Anna's* wish-for-walk). Modeling user perception is outside the scope of the model presented here; such issues have been addressed by cognitive models elsewhere such as the model of *unawareness* [18].

## 9 Conclusion

We have introduced a formal model of awareness systems, based on the focus/nimbus model of Benford [2] and Rodden [19]. Where the original focus and nimbus model describes *how much* aware is entity  $i$  of entity  $j$  in a particular *space* our model describes *what* is entity  $i$  aware of regarding entity  $j$ , in a particular *situation*.

We have demonstrated that the model allows the formal expression of abstract concepts such as focus, nimbus, awareness but also socially oriented behaviors such as blurring information about oneself, lying etc. The model presented here abstracts away from modeling the propagation of awareness information as in [22] and [11], or information flow modeling as in [5]. It advances the focus/nimbus model of [2],[19] in that it is explicit about the object of awareness: i.e. the relationship of the information an entity can potentially provide about itself to that actually observed by another entity. This is necessary for modeling the social aspects of awareness systems as shown above.

Currently we are extending this work to model related concepts such as social translucence, community awareness, intentionality and symmetry of awareness systems. In the next steps of our research, an end-user programming platform for

awareness systems will be created where users will be allowed to easily tailor the behavior of their system to effect blurring, anonymity, symmetry etc. The model presented can guide the design of this experimental platform and provides the conceptual foundations for defining an ontology by which awareness information can be described and reasoned about.

## References

1. Aoki, P., Woodruff, A.: Making Space for Stories: Ambiguity in the Design of Personal Communication Systems. In: Proc. CHI 2005, pp. 181–190 (2005)
2. Benford, S., Bullock, A., Cook, N., Harvey, P., Ingram, R., Lee, O.: From rooms to cyberspace: models of interaction in large virtual computer spaces. *Interacting with Computers* 5(2), 217–237 (1993)
3. Price, B.A., Adam, K., Nuseibeh, B.: Keeping ubiquitous computing to yourself: A practical model for user control of privacy. *International Journal of Human-Computer Studies* 63(1-2), 228–253 (2005)
4. Boyle, M., Greenberg, S.: The Language of Privacy: Learning from Video Media Space Analysis and Design. *ACM ToCHI* 12(2), 328–370 (2005)
5. Bryans, J.W., Fitzgerald, J.S., Jones, C.B., Mozolevsky, I.: Formal Modelling of Dynamic Coalitions, with an Application in Chemical Engineering. In: Proc. of the 2nd International Symposium on Leveraging Applications of Formal Models (to appear)
6. Consolvo, S., Roessler, P., Shelton, B.E.: The careNet display: Lessons learned from an in home evaluation of an ambient display. In: Davies, N., Mynatt, E.D., Siio, I. (eds.) *UbiComp 2004*. LNCS, vol. 3205, pp. 22–29. Springer, Heidelberg (2004)
7. Consolvo, S., Smith, I., Matthews, T., LaMarca, A., Tabert, J., Powledge, P.: Location Disclosure to Social Relations: Why, When, & What People Want to Share. In: Proc. of the Conference on Human Factors and Computing Systems: CHI 2005, pp. 81–90 (2005)
8. Dourish, P., Bellotti, V.: Awareness and Coordination in Shared Workspaces. In: *Proceedings, CHI 1992*, pp. 117–124. ACM Press, New York (1992)
9. Erickson, T., Halverson, C., Kellogg, W.A., Laff, M., Wolf, T.: Social translucence: designing social infrastructures that make collective activity visible. *Commun. ACM* 45(4), 40–44 (2002)
10. Fernando, O., Adach, K., Cohen, M.: Phantom Sources for Separation of Listening and Viewing Positions of Multipresent Avatars in Narrowcasting Collaborative Virtual Environments. In: *Proceedings, ICDCSW 2004* (2004)
11. Fuchs, L., Pankoke-Babatz, U., Prinz, W.: Supporting cooperative awareness with local event mechanisms: The GroupDesk system. In: *Proceedings of ECSCW 1995*, pp. 247–262 (1995)
12. Hindus, D., Mainwaring, S.D., Leduc, N., Hagström, A.E., Bayley, O.: Casablanca: Designing social communication devices for the home. In: *Proceedings CHI 2001*, pp. 325–332 (2001)
13. Hong, J.I., Landay, J.A.: An Architecture for Privacy- Sensitive Ubiquitous Computing. In: *Mobisys 2004*, Boston, MA, pp. 177–189 (2004)
14. Iachello, G., Smith, I., Consolvo, S., Chen, M., Abowd, G.: Developing Privacy Guidelines for Social Location Disclosure Applications and Services. In: *Proceedings of the Symposium on Usable Privacy and Security (SOUPS 2005)* (2005)
15. Gao-feng, J., Yong, T., Yun-cheng, J.: A service-oriented group awareness model and its implementation. In: Lang, J., Lin, F., Wang, J. (eds.) *KSEM 2006*. LNCS, vol. 4092, pp. 139–150. Springer, Heidelberg (2006)

16. Lederer, S., Mankoff, J., Dey, A.K.: Who Wants to Know What When? Privacy Preference Determinants in Ubiquitous Computing. In: Proceedings of Extended Abstracts of CHI 2003, ACM Conference on Human Factors in Computing Systems, Fort Lauderdale, FL, pp. 724–725 (2003)
17. Markopoulos, P., Romero, N., van Baren, J., IJsselsteijn, W., de Ruyter, B., Farshchian, B.: Keeping in touch with the family: home and away with the ASTRA awareness system. In: CHI Extended Abstracts 2004, pp. 1351–1354 (2004)
18. Modica, S., Rustichini, A.: Awareness and Partitional Information Structures. *Theory Dec.* 37, 107–124 (1994)
19. Rodden, T.: Populating the Application: A Model of Awareness for Cooperative Applications. In: Proc. ACM 1996 (CSCW 1996), pp. 87–96 (1996)
20. Rowan, J., Mynatt, E.D.: Digital family portrait field trial: Support for aging in place. In: Proc. CHI 2005, pp. 521–530 (2005)
21. Schmidt, K.: The Problem with Awareness Introductory Remarks on Awareness in CSCW. *Computer Supported Collaborative Work* 11(34), 285–298 (2002)
22. Simone, C., Bandini, S.: Integrating Awareness in Cooperative Applications through the Reaction-Diffusion Metaphor. *Computer Supported Cooperative Work: The Journal of Collaborative Computing* 11(3–4), 495–530 (2002)

## Questions

### **Michael Harrison:**

*Question: Had you thought about using non-standard logics such as knowledge logics to express the information you're trying to express? See for example: Fagin, R., Halpern, J. Y., Moses, Y. and Vardi, M Y. Reasoning about knowledge., MIT Press, Cambridge, Massachusetts, 1995.*

Answer: We don't try to express knowledge as cognition. We haven't looked in that direction. Thank you for the reference.

### **Fabio Paterno':**

*Question: For what applications is the modeling appropriate?*

Answer: For investigating how people can configure their awareness systems themselves. It helps to identify patterns within an awareness system. It allows people to configure awareness of their activities and supports lightweight communication systems.

### **Morten Borup Harning:**

*Question: Is the idea that your model can help modeling for privacy by making the awareness properties and interdependencies clearer?*

Answer: Privacy is a concern of the model. We can describe aspects of behavior that are relevant to privacy.

### **Anke Dittmar:**

*Question: I can imagine that people would change their behavior if they make information explicit through an awareness system based on formalized descriptions of lying etc. and their knowledge about the system. Did you consider this in your formalization?*

Answer: We have tried to keep as close as possible to what people actually do, but we haven't considered this particular issue so far.

# Service-Interaction Descriptions: Augmenting Services with User Interface Models

Jo Vermeulen, Yves Vandriessche, Tim Clerckx, Kris Luyten, and Karin Coninx

Hasselt University – transnationale Universiteit Limburg,  
Expertise Centre for Digital Media – IBBT,  
Wetenschapspark 2, B3590 Diepenbeek, Belgium  
{jo.vermeulen,yves.vandriessche,tim.clerckx,kris.luyten,  
karin.coninx}@uhasselt.be

**Abstract.** Semantic service descriptions have paved the way for flexible interaction with services in a mobile computing environment. Services can be automatically discovered, invoked and even composed. On the contrary, the user interfaces for interacting with these services are often still designed by hand. This approach poses a serious threat to the overall flexibility of the system. To make the user interface design process scale, it should be automated as much as possible. We propose to augment service descriptions with high-level user interface models to support automatic user interface adaptation. Our method builds upon OWL-S, an ontology for Semantic Web Services, by connecting a collection of OWL-S services to a hierarchical task structure and selected presentation information. This allows end-users to interact with services on a variety of platforms.

**Keywords:** Model-based user interface development, Semantic web services, Screen layout, Automatic generation of user interfaces, User interface design, Ubiquitous computing.

## 1 Introduction

In this paper, we introduce a framework to design *services* that automatically present a suitable user interface (UI) on a wide variety of *computing platforms*.

The main objective of our system is to allow mobile users to flexibly interact with services in a city environment. A city environment is often very volatile. Users come and go, carrying with them different devices and having different needs for the resulting user interface (e.g. a visually handicapped person might prefer speech interaction).

By *service*, we refer to an application that provides useful functions to end-users. Users interacting with these services use a variety of devices with different operating systems and user interface toolkits. A *computing platform* is the combination of a device, operating system and toolkit. The user interface for a service thus runs on a computing platform.

The city environment we described roughly corresponds to the vision of ubiquitous computing [26]. Its goal is for users to move through their environment, finding resources and services as they go, and to have those services provided in the context of

their physical environment. This vision is slowly becoming a reality with the increasing market penetration of ever more capable mobile devices, the availability of advanced sensors and cheaper network access.

Semantic service descriptions are more and more used to describe services in a ubiquitous computing environment. Discovering, invoking and even composing these services based on their semantics has already proven effective. Unfortunately, the resulting user interface was left out of the equation. Usually, the user interface for interacting with a service is still designed by hand. This seriously decreases the flexibility of the system. Designing each user interface by hand requires prior knowledge of the available services, their inner workings and possible service compositions, not to mention the computing platform where the user interface has to be deployed and the context-of-use.

People will use services as they become available. However, the designers of a service may have never anticipated the user's device as a target platform. It is not reasonable to require services to have a custom-made user interface available for each possible situation, neither is it reasonable the other way around, to require each target platform to support every possible service. A more general solution is needed.

Our approach uses existing metadata about semantic web services and custom, high-level annotations about the resulting user interface, to allow for advanced adaptation to any target platform. These custom annotations link user interface models with the logical components of the service. We call the resulting service description a *service-interaction description*.

We describe three contributions in this work:

- The combination of semantic service descriptions with a model-based user interface development approach. While annotating service descriptions with user interface information has been explored before, the use of model-based techniques results in a higher degree of abstraction, enabling adaptation to any target platform.
- The creation of an extensible semantic network<sup>1</sup> of presentation information which is used to model an extra layer of abstraction on top of the User Interface Markup Language (UIML). By providing the link between the abstract and concrete presentation information, we are able to perform an automatic mapping from the former to the latter.
- A hierarchical and reusable graphical layout model that describes layout on a concrete level while keeping the interface flexible. We obtain this through the use of spatial constraints and by connecting the layout to the abstract user interface. With this we attempt to comply to the plasticity requirements inherent in user interfaces for services that have to be deployed on a variety of platforms.

The remainder of the paper is organized as follows. The next section discusses related work. Then, we give an architectural overview of our approach. Subsequently, the details of service-interaction descriptions are discussed (Sect. 4). Sect. 5 gives an overview of how high-level user interface models are transformed into a concrete user interface. First, we describe the central model in our approach: an annotated task

---

<sup>1</sup> A semantic network is a form of knowledge representation, consisting of concepts and semantic relations between these concepts.

model which will be used to extract the dialog model (Sect. 5.1). Secondly, we introduce the semantic network built on top of UIML and explain how it can be used to perform automatic widget selection (Sect. 5.2). Next, we discuss layout templates which can be used to position the selected widgets for the graphical modality (Sect. 6). After presenting the main ideas, we provide a walkthrough of the design of a photo sharing service using our system (Sect. 7). Finally, we draw some conclusions while looking ahead for possibilities in future work.

## 2 Related Work

Much work has been done in combining service descriptions with user interface information. We do not aspire to give a complete overview of the existing work in this area. Yet, we have selected a couple of notable examples which we feel are most relevant for this paper. We believe our approach is unique in that the use of high-level user interface models results in a higher level of abstraction while still offering the possibility for manipulating the final presentations. In addition, by building on semantic web services it is possible to leverage the existing work in automated discovery, invocation, composition and monitoring of these services.

XWeb is inspired by the architecture of the World Wide Web. It allows a variety of interactive platforms to communicate with services by means of a uniform protocol [8]. Service providers specify XViews that define the interaction with the data of the service, in a device-independent manner. The clients themselves decide how to render these XViews. A drawback of XWeb is that each client must know when to request the correct XView. There is no information about the structure of the user interface, and in which way an end-user will interact with the service. We, on the other hand, do provide this information through the task and dialog model.

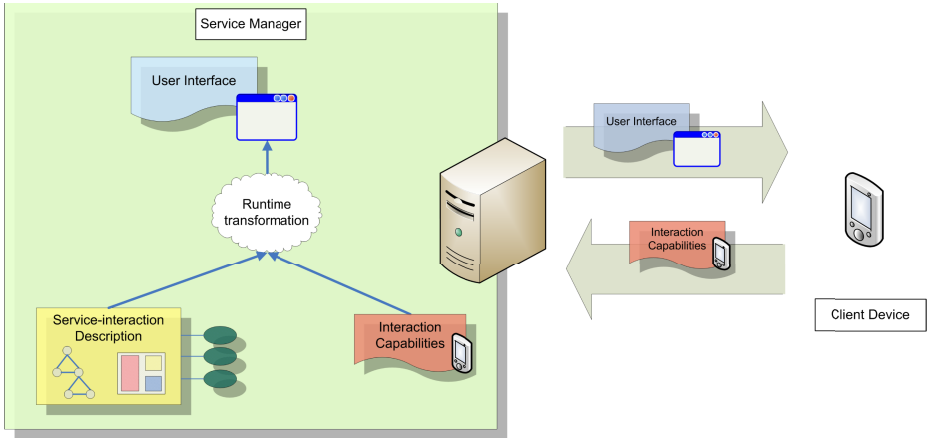
Khushraj et al. [14] also use OWL-S service descriptions and augment them with user interface information to generate personalized user interfaces. Their system is oriented towards automated form-filling based on context information, which means the user interface annotations are too concrete to be useful for the problems that are targeted in our approach.

ICrafter [23] is an architecture to select, generate and/or service user interfaces at runtime. The authors also state that they support aggregation of service UIs. User interface generators be written for patterns of services, which are services conforming to a common programmatic interface. In fact, this comes down to providing the same user interface for a collection of services with similar semantics, instead of merging two existing service UIs. Semantic web services already solve the problem of composing the functional descriptions of two services, but in a more generic way. A disadvantage of ICrafter is the fact that the appliance-specific UI generators have to be programmed by hand. This means that whenever a new target platform has to be supported, a corresponding UI generator needs to be created. The use of a concrete abstraction layer (UIML in our approach) solves this problem.

Manolescu et al. [21] describe a model-driven design and deployment process for integrating web services with web applications that have a predefined user interface. Another example of the combination of WSDL service descriptions and user interface models is the CATWALK framework [25]. This framework mainly concentrates on the creation of the actual web pages that interact with the services.

### 3 Architectural Overview

The work we present in this paper enables users to flexibly interact with services. Our approach is centered on the combination of semantic service descriptions and high-level user interface models [10]. Fig. 1 illustrates how the system can create a suitable user interface to allow an end-user to interact with a particular service.



**Fig. 1.** Architectural overview

The client device on the right wants to make use of a particular service. To do so it sends a *service-interaction request* to the *Service Manager*. This request consists of a description of the client platform's interactive capabilities, together with a reference the service it wants to address. First, the *Service Manager* looks up the correct service-interaction description which consists of both the functional description and the user interface information. Then, the service's high-level user interface information is combined with the knowledge of the client's interaction capabilities to form a concrete user interface for the client platform. This transformation is performed *at runtime*. Finally, the *Service Manager* sends a *service-interaction response* to the client, containing the user interface for the requested service.

The next section will discuss the creation of service-interaction descriptions. Afterwards, Sect 5 and 6 will explain in detail the transformation of high-level user interface information into a concrete user interface.

### 4 Service-Interaction Descriptions

The first step in our approach is to extend service descriptions with user interface information. First, we define the terms *web service* and *service description*. The World Wide Web Consortium (W3C) defines a *web service* as “a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks” [15]. A web service generally provides a *service description*, which includes a description of its interface among other information (e.g. the



URL<sup>2</sup> where the service can be reached). Most existing web services use the Web Service Description Language (WSDL)<sup>3</sup> for this purpose.

Although it is possible to augment WSDL with user interface information (as demonstrated by Kassof et al. [13]), WSDL's lack of semantics makes it very difficult to generate a suitable user interface. The Semantic Web is a vision of the next generation of the World Wide Web, characterized by formally described semantics for content and services [2]. These semantics are described by knowledge representation languages such as the Resource Description Framework (RDF)<sup>4</sup> and the Web Ontology Language (OWL)<sup>5</sup>. RDF and OWL, in turn, refer to *ontologies*, specifications of conceptualizations [11], which enable *reasoning* through the use of logic rules. *Semantic web services* originate from the augmentation of web service descriptions with formal semantics. The extra semantics facilitate the automation of discovery, invocation, composition and monitoring of these services. It is exactly this new "extension" that enables us to automatically generate suitable user interfaces for web services. First, the added semantics are useful for selecting an appropriate presentation (e.g. the meaning of inputs and outputs). Secondly, we can easily link the service with our own semantics, which is in this case the high-level user interface information. We chose to use OWL-S<sup>6</sup>, an OWL-based web service ontology. An OWL-S service can be mapped to a concrete realization of the service (such as a WSDL description). This means existing web services can be reused and extended with an OWL-S description.

We should note however that there is an important difference between an end-user's perception of a service and what is described in an OWL-S service description. For example, the Google search WSDL description<sup>7</sup>, defines three basic operations: `doGetCachedPage`, `doSpellingSuggestion`, and `doGoogleSearch`. If we convert this WSDL file to OWL-S, we end up with three different OWL-S services (one for each operation). It is not possible to describe these operations as a single OWL-S service since they each have different inputs and outputs. After all, an OWL-S service advertises itself by its functional description which includes the accepted inputs and outputs. Nevertheless, the end-user views the entire WSDL description (the combination of search, spelling suggestions and cached pages) as a single service provided by Google. To prevent confusion, our notion of a service should correspond to the one of the end-user. The high-level user interface information for this kind of service will thus often cover multiple OWL-S services. This means a number of OWL-S services will have to be coupled into a custom service description which is in turn linked to the abstract user interface. We define this as a *service-interaction description*, since it contains the necessary information to allow both machines and humans to easily interact with a particular service.

The abstract user interface of service-interaction descriptions is based on a hierarchical task model which describes the tasks that can be performed by users in order to reach a goal. We describe this task model with the ConcurTaskTrees (CTT) notation [22]. Tasks can be decomposed into subtasks, resulting in a hierarchical tree

---

<sup>2</sup> Uniform Resource Locator

<sup>3</sup> <http://www.w3.org/TR/wsdl>

<sup>4</sup> <http://www.w3.org/TR/rdf-concepts/>

<sup>5</sup> <http://www.w3.org/TR/owl-features/>

<sup>6</sup> <http://www.w3.org/Submission/OWL-S/>

<sup>7</sup> <http://api.google.com/GoogleSearch.wsdl>

structure. The deeper we go into the hierarchy, the more concrete the tasks are. The task model can be used to extract more concrete models, such as the dialog model and presentation model [19]. Elements from the dialog and presentation models are associated with leaf tasks<sup>8</sup>. The designer also has to link these leaf tasks to service components, which as a result provides the link between the user interface models and the service descriptions. The next section provides more details on how this allows the abstract user interface information to be translated to a concrete user interface.

Fig. 2 shows the different components of a service-interaction description. It combines a hierarchical task model with a layout model and a number of OWL-S services. These services can be grounded into a single WSDL description for easy invocation by the concrete user interface.

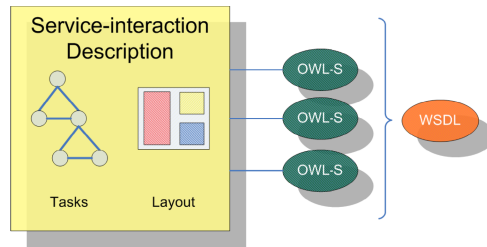


Fig. 2. An overview of the components of a service-interaction description

## 5 Producing the Concrete User Interface

The previous section described how semantic web services were augmented with high-level user interface models. These models provide enough abstraction to be applicable for every computing platform. However, to be actually useful, they have to be translated into a concrete user interface for a specific platform. This section will discuss how we perform this transformation.

First, we give an overview of the four levels of abstraction for multi-platform user interfaces, as defined by the CAMELEON Reference Framework [4] (sorted from the most concrete to the most abstract level): (1) the *Final User Interface* (FUI) is the operational UI; (2) the *Concrete User Interface* (CUI) expresses any FUI independently of any markup or programming language; (3) the *Abstract User Interface* (AUI) expresses any CUI independently of any interaction modality (e.g. graphical, vocal, tactile, ...) via the mechanism of Abstract Interaction Objects (AIOs) as opposed to Concrete Interaction Objects (CIOs) for the CUI and Final Interaction Objects (FIOs) for the FUI; and finally (4) the *Task and Concepts* level, which describes the various interactive tasks to be carried out by the end user and the domain objects that are manipulated by these tasks.

The service-interaction descriptions contain a hierarchical task model in the ConcurTaskTrees (CTT) notation [22], which corresponds to the Tasks and Concepts level. We assume that each client device knows how to transform a CUI to a FUI.

<sup>8</sup> Leaf tasks are the most concrete tasks: they cannot be decomposed further into subtasks.

This means the transformation process ranges only from the Task and Concepts level to the CUI. First, the task model should be transformed into an AUI, whereafter this AUI is transformed into a CUI. The next section discusses the first mapping, while Sect. 5.2 provides more details about mapping the AUI to a CUI, for which we use the UIML language.

### 5.1 Annotating the Task Model

In order to ease the transition from the task model to an AUI, we annotate leaf tasks with service components and AIOs. This requires the task model to be decomposed up to the level that each leaf task can be connected to a single AIO and service component. A *service component* can be an input or output of an OWL-S service or the service itself.

An important step in the transformation to the AUI is the extraction of a dialog model. The dialog model is a State Transition Network (STN), modeling the possible states of the user interface. In each state, a “dialog” is conducted between the user and the system. We use the annotated task model to generate a corresponding dialog model [19]. Each state in this model is an *Enabled Task Set (ETS)*. An ETS is a collection of tasks that are enabled during the same time period, which means they should be presented to the user simultaneously, i.e. in the same dialog [22].

In conclusion, our AUI consists of the annotated task model and the extracted dialog model. We now know of which states the user interface is comprised and which leaf tasks belong to these states. The fact that these tasks are annotated with AIOs and service components will prove useful in the next section.

### 5.2 Widget Selection through Enhanced UIML Metadata

The next step is to transform the AUI into a CUI. As described earlier, we assume that each client device knows how to present a CUI to the user. For the CUI level, the User Interface Markup Language (UIML) [1] is used.

UIML is an XML-based language to describe the structure, style, content, and behavior of a user interface. Unlike other user interface markup languages, UIML does not use metaphor-specific tags (such as `window` or `button`), but only generic tags (e.g. `part`, `property`, ...). These generic tags can be associated with a set of abstractions, defined in the `peers` section. The `peers` section specifies how these abstractions can be translated into a final presentation. Basically, the abstractions define a *vocabulary* of classes and names to be used with a UIML document. Since the vocabulary is specified separately, new devices and UI metaphors can be supported when they become available in the future. The CIOs will be defined by this vocabulary.

We use UIML solely for the CUI level, because its level of abstraction is not sufficient for covering different platforms with widely varying interaction mechanisms. The vocabulary can only provide a very thin layer of abstraction above the target platform since it uses a one-to-one mapping of an abstraction to a final widget. If we situate UIML in the CAMELEON framework [4], it only covers the concrete and final level. The vocabulary can thus be seen as a one-to-one mapping from concrete interactors (CIOs) to final interactors (FIOs). Although it is possible to describe abstract

interactors (AIOs) with UIML, we would then have to map them directly to FIOs. This is too big of a step to be feasible for every possible platform and interaction mechanism.

The remaining problem now is how to perform a smooth transition from the AUI to UIML. Most tools (e.g. DynaMo-AID [6]) often only define this mapping internally. In our opinion, it is better to specify this information externally in a machine-readable way.

An interesting approach to connect the different levels of abstraction is described by Demeure et al. [9]. They have exploited a semantic network of the concepts and relationships that are involved at each level of abstraction to pose interesting questions about a running user interface. For example, one could ask “What are the alternative CIOs for the CIO *ListBox*?” This would allow us to perform automatic widget remapping just by reasoning about the semantic network. Adaptation rules would not have to be hard-coded into the software or into the user interface design. Demeure's semantic network is defined in a custom format, which complicates interoperability with other software. With the advent of the Semantic Web [2] however, the Resource Description Framework (RDF) has been widely accepted as the standard format for representing knowledge.

**A semantic network built on top of UIML.** We adopt the approach presented in [9] by Demeure et al., and adjust it to our system. We will use RDF to describe the UIML *peers* section and link it with an external AIO classification, thereby building our own semantic network. An additional advantage of using RDF is the easy integration with service-interaction descriptions, which are also described with RDF. Since the UIML vocabulary covers the concrete and final levels, the first step is to express this information with RDF.

We defined a *peers* ontology<sup>9</sup> by performing a straightforward mapping from UIML tags to OWL classes. The four concepts (and therefore OWL classes) defined in this ontology are: *Presentation*, *DClass*, *DProperty*, and *DParam*. A simple tool was developed to convert an original UIML vocabulary to its RDF representation and vice-versa.

In order to connect the concrete and abstract levels, we extend the ontology with the concept of an *AIO* and the relationship *reifies*. The *reifies* relationship works on a *DClass* and an *AIO* instance, to indicate that the former is a concretization of the latter. Note that we do not explicitly define an ontology of AIOs. Our ontology only defines the *AIO* concept, and the relationship that links it with a *DClass*. This approach is necessary to provide the same level of flexibility for the abstract level as the UIML vocabulary provides for the concrete level. It allows *AIO* classifications to be specified separately in external ontologies. The only requirement for this is that the different AIOs are specializations of our *AIO* concept, so that they can be linked with a *DClass* instance.

Of course, in order to actually link CIOs with AIOs, we first need to define a set of AIOs that we can use. According to the definition from [4], AIOs should be modality-independent. We will use a very high-level, minimal set of AIOs that are differentiated according to the functionality they offer to the user: (1) *input* components allow users to enter or manipulate data; (2) *output* components provide data from the application to the user; (3) *action* components allow a user to trigger some functional-

<sup>9</sup> This ontology is available at <http://research.edm.uhasselt.be/~uiml/peers/elements/0.1>

ity; and finally (4) *group* components group other components into a hierarchical structure. We define these four AIOs (*Input*, *Output*, *Action* and *Group*) in an external ontology as the only instances of the AIO concept of our peers ontology.

**Adding data types.** A disadvantage of the generic, modality-independent AIO classification we just discussed is the fact that each AIO applies to a large number of CIOs. This means that extra information is required in order to select the correct CIO for a given AIO. The service description provides us with the associated data type, which allows us to narrow down the number of possible CIOs. Consider for example the AIO *Input*. This AIO can map on different CIOs such as a combo box, a spin box, a text entry, a check box, a radio button, or a calendar. However, if we add the constraint that the data type should be a *boolean*, our choice is automatically limited to the check box and radio button.

The concept *DataType* and the relationship *hasDataType* was added to our peers ontology, in order to relate DClass instances with a data type. Again, data types can be defined externally, to allow for maximum flexibility. We created a data type classification, based on XML Schema<sup>10</sup>. The ontology consists of the primitive types of XML Schema (e.g. *decimal*, *string*, *void*, etc.) in addition to a number of data types which are often used in user interfaces (e.g. *Image*, *Color*, etc.).

The leaf tasks that are annotated with an AIO and a service component provide the necessary information to be mapped on a concrete interactor. Sect. 5.1 defined a service component as an input or output of a service, or the service itself. Inputs and outputs have an associated type, while the service can be linked with the data type *Void*. However, inputs and outputs of a OWL-S service are often associated with *semantic types*, which are arbitrary concepts (e.g. *Price*). It would be unreasonable to require each OWL-S service to use our own data types. We therefore allow a service developer to link semantic types with their corresponding data type (e.g. *Price* could be linked with *Float*). This technique allows us to associate inputs and outputs of a service with elements from the data type ontology, while retaining the semantics of the existing OWL-S service. To do so, we extend the peers ontology with the relationship *associatedDataType* that can links arbitrary concepts with a *DataType* instance. When a leaf task is associated with an entire service (linked to the data type *Void*), it will also be coupled with the AIO *Action*. This is to indicate that a leaf task invokes a certain service. An example of a CIO that is associated with the AIO *Action* and the data type *Void* is a *Button*.

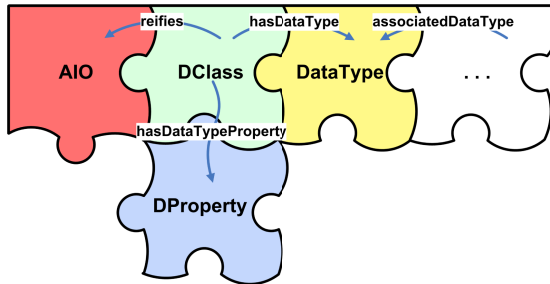
A final requirement to translate an AIO and data type to a CIO, is to indicate through which DProperty the DClass is associated with the data type. For example, the DClass *Label* can be associated with the AIO *Input* and the data type *String*, through the property *text*. We add the relationship *hasDataTypeProperty* to the peers ontology for this purpose. A UIML renderer should know how to translate each element from the data type classification to its platform-specific data type (e.g. *String* to `java.lang.String`).

**Conclusion.** The metadata we added on top of the UIML vocabulary defines a mapping from an AIO and data type tuple to a certain CIO. Fig. 3 gives an overview of

---

<sup>10</sup> <http://www.w3.org/TR/xmlschema11-2/>

the different concepts we introduced, and the relationships between them. Note that the puzzle piece on the far right represents an arbitrary concept that can be linked to a *DataType* instance.



**Fig. 3.** The different concepts in the semantic network and the relationships that connect them

The extended UIML vocabulary that was introduced here will represent a target platform's interactive capabilities. When a client device wishes to use a certain service, it sends this description to the Service Manager, as discussed in Sect. 3. In order to translate the AUI to a CUI, which is described with UIML, we use the following process. For each ETS in the dialog model, the enabled tasks are translated to corresponding CIOs, using the associated AIO and service component. To arrive at a concrete UIML user interface, a skeleton UIML description could be used, which would be filled in with the CIOs from the previous step. However, this is not an ideal solution for graphical user interfaces (GUIs). After all, static positions for the CIOs or even a standard layout will not scale between widely varying screen sizes and in addition could seriously affect the usability of the resulting user interface. The next section introduces a layout model, which we developed to overcome this problem.

## 6 Specifying the Layout

This section will present a layout model, which is an extension of our approach targeted to graphical user interfaces, as discussed in the previous section. Existing work has been done in specifying the layout on the abstract user interface level, but relations between AIOs on this level are hard to map onto a concrete layout. We will therefore focus in this work on the graphical modality. The use of a layout model is still justified because there is a need for a certain amount of flexibility which cannot be obtained by a static layout specified at design time.

### 6.1 Current Approaches

The most common approach to specify the layout in model-based user interface development is to group AIOs. An example of this is the hierarchically structured Logical Windows abstraction [10]. Combining AIOs under a *Group* AIO parent will guarantee that these components will stay logically grouped in the concrete user interface.

The way group AIOs are represented in the CUI will affect the eventual positions of their children. For example, group AIOs can be mapped onto a horizontal box container, which means their children are positioned on a horizontal line, from left to right. Group AIOs can be part of another group AIO, which allows a nested layout specification. However, the UI designer has only limited control over the final layout with this technique.

A pattern-based approach such as described in [24] and [18] defines layout patterns that aggregate interface elements into a specified graphical layout. In practice, these layout patterns represent simple layout containers (eg. a horizontal box). The corresponding layout model consists of layout patterns written beforehand in a template language. This technique works on a more concrete level, giving the designer a good idea of what the final UI layout will look like. However, from a modeling perspective it would be better if a designer could specify his own templates within the layout model instead of using a template language.

Another way of expressing a more concrete layout is by the use of spatial constraints on abstract UI elements [7]. This technique has been covered in many publications, such as [3] or [12]. Usually there are two approaches for obtaining these layout constraints: the designer can explicitly specify the required constraints (by means of a visual tool or by using a declarative constraint language) or constraints can be generated automatically. The latter uses either visual cues [17] or external ones such as data relationships.

Allen constraints express relationships between time and space intervals [16]. By specifying Allen relationships between AIOs we can express both spatial relationships for visual layout and temporal relationships for non-visual interfaces. Allen relationships have to be mapped onto a more concrete level, much like group AIOs. We wish to work on a more concrete level to avoid exposing the designer to this mapping problem inherent to the use of group AIOs and Allen relationships in layout design.

## 6.2 Layout Model

In this section we present a tentative approach for specifying a layout model that can be applied on the CUI level. By specifying layout on a concrete, 2D graphical level we avoid the AUI layout abstraction problem. We try to preserve the hierarchical structure introduced by group AIOs, enable reuse of patterns and allow concrete spatial constraint relations. However, we still need some of the abstractions provided by AIOs as we cannot predict the specific target platform. As seen in Fig. 4, the layout model consists of two parts, a set of *layout templates* and one of *layout instances*.

A layout template describes the structure of the layout, using hierarchical layout elements and layout relations representing spatial constraints between these elements. In Fig. 4, the root layout element of the template represented has three child elements, two leaf elements and one nested layout element which has three children of its own. The arrows between sibling layout elements represent the layout relations that exist between them. A layout template needs to be instantiated with AIOs and related with a certain state from the dialog model to be able to provide a concrete UI description. The resulting layout instance will describe the mapping between the abstract layout elements and AIOs for a single dialog. AIOs are connected to layout elements using layout instances to enable reuse of the layout templates.

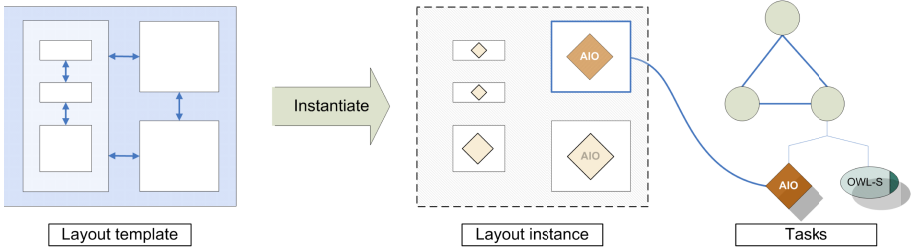


Fig. 4. Instantiating a layout template with AIOs

The structure of a layout template is described by *hierarchical layout elements*. These layout elements are equivalent to group AIOs; they provide a logical window and can be nested to create a hierarchy, as explained earlier in Sect. 6.1. A logical window in this context means that layout relations can only be defined between siblings and their parent element. Layout templates differ from group AIOs in that they use geometric relations between the elements they contain to describe the actual graphical layout.

We currently use a simple set of linear geometrical constraints as an example: *align-top*, *align-center*, *left-of*, *under*, *above*, etc. In addition we also add some more complex relations: *horizontal box* and *vertical box* containers. Layout relations are abstract enough to support other types of constraints. A layout template contains a reference to a single layout element and a collection of layout relations. The referenced layout element acts as the root node of a hierarchy of layout elements. The collection of layout relations contains geometric constraints expressed between the elements of that layout element hierarchy.

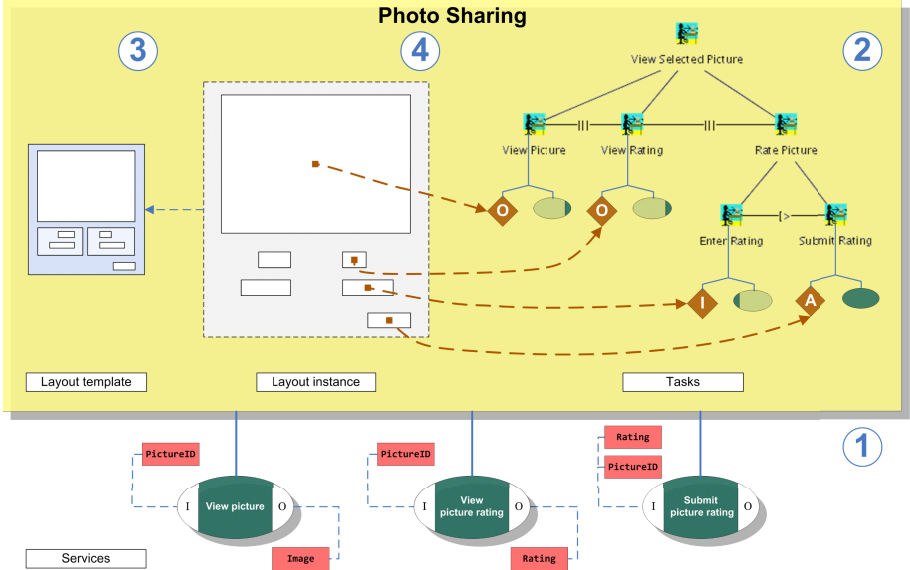
A layout element in a template is a placeholder on which layout relationships such as geometric constraints are defined. During the instantiation of the layout template we can fill these placeholders with AIO elements from the abstract user interface model. A layout instance describes the mapping between layout elements and AIOs. The layout instantiation process has two main requirements. AIOs used in an instantiation have to be coupled to tasks inside the same Enabled Task Set (ETS) [22] from the dialog model. By definition, tasks in different enabled task sets cannot be shown at the same time. The designer will thus create a layout for each ETS. As a second requirement, we prohibit layout templates to be instantiated with group AIOs. As mentioned earlier, group AIOs can be used to logically group AIOs on the AUI level. Since the layout inside group AIOs is unspecified, it is not possible to instantiate a layout element with a group AIO without using default layout rules. However, this would defeat the purpose of our layout specification. It is up to the designer to split the layout elements to allow a one-to-one mapping.

For this work, we use UIML to specify the concrete user interface. However, our layout model is generic enough to be mapped on other CUI representations. We generate a skeleton UIML description based on the layout instances. This skeleton contains the structure of the UI expressed as nested `part` elements. The instance's layout constraints will be mapped onto the UIML layout extension we developed in [20]. The specific `DClass` of the child parts in this skeleton will be filled in by the widget selection as explained earlier in Sect. 5.2. Our technique offers a certain amount of flexibility in the layout by the use of spatial layout constraints and a hierarchical layout specification.



## 7 Case Study

We clarify our approach by applying it to a mobile city service that allows people to share pictures with each other. Users can rate each picture of which an average rating is computed. The remainder of this section provides a walkthrough of the development of this service, which consists of four steps as shown in Fig. 5.



**Fig. 5.** The service-interaction description corresponding to a selected part of the photo sharing service

To integrate the photo sharing service within our system, we need to create a service-interaction description which consists of a collection of services and a task and layout model. We extended the existing DynaMo-AID tool [6] with support for developing service-interaction descriptions. For brevity's sake, we will focus only on the functionality and corresponding user interface to show the details of a single picture. This allows users to take a look at the picture, view its average rating, and add a rating of their own.

### 7.1 Collecting the Required Services

The first step is to import the necessary OWL-S services, which corresponds to step (1) of Fig. 5. The required services for viewing a picture's details are: (i) a service to retrieve a single picture; (ii) a service to get the average rating of a picture; and finally, (iii) a service to rate a certain picture. Fig. 5 shows these services and their semantic input and output types.

## 7.2 Creating the Task Model

After importing the OWL-S services, the next step is to create a hierarchical task model that specifies how users will interact with the photo sharing service. The task model should be decomposed up to the level where every leaf task can be annotated with a single AIO and service component. Afterwards, the task model is used to extract a corresponding dialog model (that is constituted of a number of Enabled Task Sets).

The part of the task model we will discuss is the interaction task *View Selected Picture* and its four subtasks, as shown in step (2) of Fig. 5. The task *View Picture* is annotated with the *Output* AIO and *Image* data type. *View Rating* and *Enter Rating* are both linked to the data type *Rating*, while the former has the AIO *Output* and the latter the AIO *Input*. Finally, the *Submit Rating* task is annotated with the *Action* AIO and *Void* data type.

At this point, we should also map the semantic types of the inputs and outputs to our data type classification, as described in Sect. 5.2. For example, *Rating* will be mapped to *StringEnum*.

## 7.3 Creating or Reusing a Layout Template

Before designing the layout we need the ETS containing the tasks we discussed in step (2) of Fig. 5. This gives us an overview of the tasks and attached AIOs that need to be presented in a single dialog. Although an existing template (or even some of its parts) could have been reused, we create a layout template from scratch here to illustrate our technique. The layout template in step (3) is constructed by drawing a couple of boxes which represent the layout elements. The shape and size of the boxes are irrelevant, but their relation to each other is. The nesting of these boxes represents the hierarchy of the corresponding layout elements.

After constructing the layout element hierarchy, the designer adds layout relations to the template. Layout relations are specified explicitly by selecting the target elements (for example the two middle boxes) and by applying a geometric constraint (e.g. *align right*).

## 7.4 Instantiating a Layout Template

Step (4) in Fig. 5 depicts the instantiation of the layout template that was just created. First, the designer selects an ETS. The AIOs linked to the tasks in this ETS can then be connected to leaf layout elements in the layout template. In our example, the set of AIOs provided by the ETS is insufficient to specify the desired user interface. To add the labels “Average Rating” and “Your Rating” the designer needs to create two additional AIOs using the existing presentation model functionality in the DynaMo-AID tool [6]. The data attached to these AIOs uses the same vocabulary as explained in Sect. 5.2 to enable widget selection. This instantiation process is repeated for each ETS in the dialog model. The service-interaction description is now complete.

## 7.5 The Resulting User Interface

After integrating the service-interaction description in our system, users can interact with the photo sharing service. To do so, their client sends a service-interaction

request along with its extended UIML vocabulary to the Service Manager, as discussed in Sect. 3. The Service Manager then replies with a platform-specific UIML description of the corresponding user interface. Finally, the client renders the UIML code, and presents it to the user. Fig. 6 shows two examples of the resulting user interface on different platforms: (a) a PDA with the Windows Mobile operating system and Windows Forms toolkit; and (b) a Smartphone with the Symbian operating system and UIQ toolkit. Note that the photo sharing service has no specific knowledge of either of these two platforms. It just uses the metadata added to the UIML vocabulary and the specified layout instances to map the abstract user interface to a concrete one.

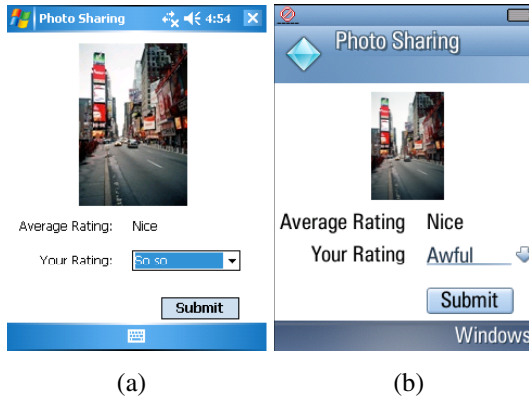


Fig. 6. The final user interface for the *View Selected Picture* task on two different platforms

## 8 Conclusions and Future Work

This paper presented *service-interaction* descriptions which combine OWL-S services with high-level user interface models in order to present a suitable user interface on any target platform. We proposed a semantic network built on top of UIML to ease the transition of the abstract to the concrete user interface. Our general approach was extended with a layout model to obtain a more visually consistent and usable UI for the graphical modality. Finally, we illustrated our approach by applying it to a photo sharing service.

We are exploring several directions for future work. First, we would like to verify the modality-independent design of the system by testing other modalities (e.g. speech). The layout model that was described in Sect. 4, would then have to be ignored since it is only useful for the graphical modality. Secondly, since it is possible to compose semantic web services, it would be interesting to investigate how the UI is influenced by this. For example, we could explore how the layout model can be modified to support this composition. In our own previous work [5] we have already taken a first step towards merging service UIs. We have shown a way to model service-aware user interfaces at the task level allowing the user interface of the main application and the one of the service to be merged into one consistent user interface. The assumption we made was that each service would have a corresponding abstract user

interface consisting of the same models as the main application. The work we presented here extends this technique at the presentation level of the user interface and explicitly links the service to the task specification.

A difficult problem concerns inconsistencies between service UIs, since the average user cannot master more than a few different user interfaces. The layout relations used in this work have been fairly straight-forward. Alternative ways of obtaining and expressing layout could be found to make the layout design process both easier and more expressive. Finally, it would be useful to extend the semantic network to allow for more advanced CIO matching. For example, CIOs could be annotated with their required size, allowing us to automatically switch to a smaller CIO when the available screen space decreases.

## Acknowledgments

Part of the research at EDM is funded by ERDF (European Regional Development Fund), the Flemish Government and the Flemish Interdisciplinary institute for Broad-Band Technology (IBBT). The CoDAMoS (Context-Driven Adaptation of Mobile Services) project IWT 030320 is directly funded by the IWT (Flemish subsidy organisation).

## References

1. Abrams, M., Phanouriou, C.: Uiml: An XML language for building device-independent user interfaces. In: XML 1999, Philadelphia, USA (1999)
2. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. *Scientific American* 284(5), 34–43 (2001)
3. Borning, A.H.: Thinglab—a constraint-oriented simulation laboratory. PhD thesis (1979)
4. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Souchon, N., Bouillon, L., Florins, M., Vanderdonckt, J.: Plasticity of user interfaces: A revised reference framework. In: Proceedings of 1st International Workshop on TAsk MOdels and DIAGrams for user interface design, pp. 127–134 (2002)
5. Clerckx, T., Van den Bergh, J., Coninx, K.: Modeling multi-level context influence on the user interface. In: PerCom Workshops, pp. 57–61 (2006)
6. Clerckx, T., Vandervelpen, C., Luyten, K., Coninx, K.: A Prototype-Driven Development Process for Context-Aware User Interfaces. In: Proceedings of the 5th International Workshop on TAsk MOdels and DIAGrams for user interface design, Diepenbeek, Belgium (October 2006)
7. Coninx, K., Luyten, K., Vandervelpen, C., Van den Bergh, J., Creemers, B.: Dygimes: Dynamically generating interfaces for mobile computing devices and embedded systems. In: Chittaro, L. (ed.) *Mobile HCI 2003*. LNCS, vol. 2795. Springer, Heidelberg (2003)
8. Olsen Jr., D.R., Jefferies, S., Nielsen, T., Moyes, W., Fredrickson, P.: Cross-modal interaction using xweb. In: UIST 2000: Proceedings of the 13th annual ACM symposium on User interface software and technology, pp. 191–200 (2000)
9. Demeure, A., Calvary, G., Coutaz, J., Vanderdonckt, J.: The comets inspector: Towards run time plasticity control based on a semantic network. In: Proceedings of the 5th International Workshop on TAsk MOdels and DIAGrams for user interface design, Diepenbeek, Belgium (October 2006)
10. Eisenstein, J., Vanderdonckt, J., Puerta, A.: Applying model-based techniques to the development of UIs for mobile computers. In: IUI 2001: Proceedings of the 6th international conference on Intelligent user interfaces, pp. 69–76. ACM Press, New York (2001)

11. Gruber, T.R.: A translation approach to portable ontology specifications. *Knowl. Acquis.* 5(2), 199–220 (1993)
12. Badros, G.J., Borning, A., Stuckey, P.J.: The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. On Computer-Human Interaction* 8(4), 267–306 (2001)
13. Kassoff, M., Kato, D., Mohsin, W.: Creating guis for web services. *IEEE Internet Computing* 7(5), 66–73 (2003)
14. Khushraj, D., Lassila, O.: Ontological approach to generating personalized user interfaces for web services. In: *International Semantic Web Conference*, pp. 916–927 (2005)
15. Lafon, Y.: *Web Services Activity Statement* (2002), <http://www.w3.org/2002/ws/Activity>
16. Limbourg, Q.: *Multi-Path Development of User Interfaces*, Ph. D. Thesis. PhD thesis (September 2004)
17. Lok, S., Feiner, S., Ngai, G.: Evaluation of visual balance for automated layout. In: *IUI 2004: Proceedings of the 9th international conference on Intelligent user interfaces*, pp. 101–108 (2004)
18. Lonczewski, F., Schreiber, S.: The fuse-system: an integrated user interface design environment. In: *Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces 1996*, pp. 37–56 (1996)
19. Luyten, K., Clerckx, T., Coninx, K., Vanderdonckt, J.: Derivation of a dialog model from a task model by activity chain extraction. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) *DSV-IS 2003. LNCS*, vol. 2844, pp. 203–217. Springer, Heidelberg (2003)
20. Luyten, K., Vermeulen, J., Coninx, K.: Constraint adaptability of multidevice user interfaces. In: *Workshop on The Many Faces of Consistency, CHI 2006 workshop* (April 2006)
21. Manolescu, I., Brambilla, M., Ceri, S., Comai, S., Fraternali, P.: Model-driven design and deployment of service-enabled web applications. *ACM Trans. Inter. Tech.* 5(3), 439–479 (2005)
22. Paterno, F.: *Model-Based Design and Evaluation of Interactive Applications*. Springer, London (1999)
23. Ponnekanti, S., Lee, B., Fox, A., Hanrahan, P., Winograd, T.: Icraft: A service framework for ubiquitous computing environments. In: Abowd, G.D., Brumitt, B., Shafer, S. (eds.) *UbiComp 2001. LNCS*, vol. 2201. Springer, Heidelberg (2001)
24. Sinnig, D., Gaffar, A., Reichart, D., Seffah, A., Forbrig, P.: Patterns in model-based engineering. In: *Proceedings of Fourth International Conference on Computer-Aided Design of User Interfaces*, pp. 195–208 (2004)
25. Lohmann, S., Kaltz, J.W., Ziegler, J.: Dynamic generation of context-adaptive web user interfaces through model interpretation. In: *Proceedings of Model Driven Design of Advanced User Interfaces 2006* (October 2006)
26. Weiser, M.: Some computer science issues in ubiquitous computing. *Commun. ACM* 36(7), 75–84 (1993)

## Questions

### *Laurence Nigay:*

*Question: Within a hierarchy of tasks the user will need several levels of service. How will you combine them: at the concrete or abstract level?*

*Answer: Indeed we need multiple services. It is an interesting question to combine the services – e.g. photo sharing with something else. It is interesting for future work.*

**Michael Harrison:**

*Question: Why didn't you use a workflow language such as BPEL to describe the orchestration of services and then describe the CIOs in the device invocation?*

Answer: In our opinion it was too low level. We use semantic web services as a more appropriate level. CTT provides a higher level hierarchical structuring of services.

**Fabio Paterno':**

*Question: To make this approach work, it means using this approach for all aspects. How do you anticipate integrating with multiple services?*

Answer: We use a model-based UI development approach ... meta data ... add user interface and some semantics.

**Prasun Dewan:**

*Question: Do you see any fundamental differences between creating UIs for mobile and desktop systems?*

Answer: On desktop you have more restricted capabilities. Screen space is greater on desktops and there is more need for multimodal input such as speech on mobile devices. We use the abstractions to define user interfaces so tasks would be the same but how they are mapped to the UI is different.

*Question: Multimodal could be useful in desktop computing too.*

Answer: Layout algorithms have to be tuned to mobile contexts, so we need to make the layout plastic.

**Morten Borup Harning:**

*Question: How do you envision that people go around discovering the possible UIs? Will you be installing the UIs before you need them or will you be able to discover them?*

Answer: Semantic web services allow us to discover new services. As yet there is just one UI which is defined at an abstract level.

# A Design-Oriented Information-Flow Refinement of the ASUR Interaction Model

Emmanuel Dubois<sup>1</sup> and Philip Gray<sup>2</sup>

<sup>1</sup>IRIT-LIHS, University of Toulouse, France

<sup>2</sup>GIST, Computing Science Department, University of Glasgow, UK  
pdg@dcs.gla.ac.uk, Emmanuel.Dubois@irit.fr

**Abstract.** The last few years have seen an explosion of interaction possibilities opened up by ubiquitous computing, mobile devices, and tangible interaction. Our methods of modelling interaction, however, have not kept up. As is to be expected with such a rich situation, there are many ways in which interaction might be modelled, focussing, for example, on user tasks, physical location(s) and mobility, data flows or software elements. In this paper, we present a model and modelling technique intended to capture key aspects of user's interaction of interest to interactive system designers, at the stage of requirements capture and early design. In particular, we characterise the interaction as a physically mediated information exchange, emphasizing the physical entities involved and their relationships with the user and with one another. We apply the model to two examples in order to illustrate its expressive power.

**Keywords:** Mixed Interactive Systems, User's Interaction Modelling, Requirements Capture, Information flow characterisation, Design Analysis, Interaction Path.

## 1 Introduction

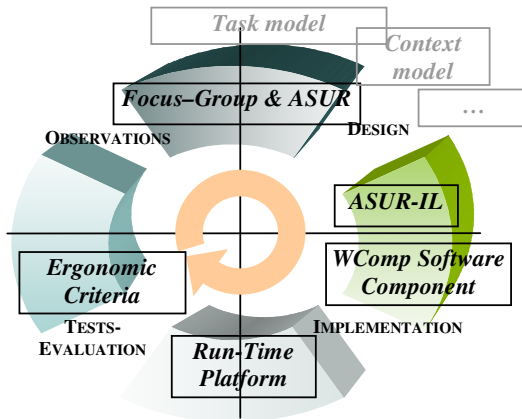
The last few years have seen an explosion of interaction possibilities opened up by ubiquitous computing, mobility, and tangible interaction. Techniques for modelling interaction in and with such systems, however, have not kept up. As is to be expected with such rich domains as these, there are many ways in which interaction might be modelled, focussing, for example, on user tasks, physical location(s) and mobility, data flows or software elements. The current situation with respect to such models presents designers with both feast and famine. On the one hand, there is a large and bewildering variety of descriptive models available to us, originating from the world of conventional interactive systems: task models, models of interaction objects, software models and existing spatial models. On the other hand, we have very few descriptive models developed for capturing augmented reality, mobile, tangible and ubiquitous applications (hereafter, we refer to these in this paper as 'mixed interactive systems'<sup>1</sup>). These range from ASUR [5, 7], the basis of the work presented in this

---

<sup>1</sup> In this paper our use of the term 'mixed interactive systems' is merely intended to informally group systems that fall under the commonly used terms, 'augmented reality', 'mobile systems', 'tangible systems' and 'ubiquitous systems'. By so doing we do not imply any common definition.

paper, that models interaction in its physical and digital aspects, through the Model of Mixed Interaction (MMI) [4] which focuses on interaction modality to MCPRD [8], a software architecture model for mixed reality.

As with software modelling, there is no single, monolithic model suitable for all software development purposes. Like UML, different models are useful for different purposes at different stages in the development process. However, unlike UML, designers of mixed interactive systems do not yet have a well-found set of models that are generally accepted, well-integrated with one another and that fit into a development process. Nevertheless, the first stages in this creating such a set are underway. Thus, ASUR, for example, now fits into a suite of models and into a development process. Figure 1 illustrates the approach.



**Fig. 1.** An integration of models & design method for Mixed Interactive Systems (i.e. ubiquitous, mixed and mobile applications)

In this paper, we present an interaction model, based on ASUR, that is intended to better express the user's experience of the physical environment in order to communicate information to and from a computer system. The goal is to capture aspects of that experience that are:

- relevant to requirements capture and to the early stages of design,
- for the assessment, comparison and discovery of designs;
- without overly complicating the analyst's or designer's task.

Although our approach is presented as an extension of ASUR, its fundamental features are independent of ASUR and could be used on their own or incorporated into other similar modelling frameworks.

Following a short overview of ASUR, we present the new interaction model, illustrated with a small example of its application. We then introduce the notion of interaction groups and use it to analyse the design options available for another example interaction technique. After briefly comparing our model to related approaches, we finish by drawing some general conclusions and considering future work.



## 2 ASUR Overview

ASUR is a notational-based model for describing user-system interaction in mixed interactive systems. ASUR is intended to help in reasoning about how to combine physical and digital “worlds” to achieve user-significant results. It is used in addition to a traditional user-system task description in order to identify objects involved in the interaction and at the boundaries between the two worlds. Adopting a user’s interaction point of view, the model is helpful in expressing the results of the requirements analysis and addressing the global design phase of a mixed interactive system. Indeed ASUR supports the description of the physical and digital entities that make up a mixed system, including adapters (Ain, Aout) bridging the gap between both digital and physical worlds, digital tools (Stool) or concepts (Sinfo, Sobj), user(s) (U) and real objects involved as tools (Rtool) or constituting the task focus (Robj). In addition, directed relationships (arrowed lines) express physical and/or digital information flows and associations among the components. To better specify these elements, viz., ASUR components and relationships, a number of characteristics have been identified, including such design-significant aspects as:

- For components: the location where the information carried by the component is perceivable or modifiable (top of table, half of the room, ...), the sense or action required so that the user perceive or act on it (hearing, sight, touch, physical action, ...) etc.
- For relationships: The dimensionality of communicated information (2D, 3D, stereoscopic, ...), the type of language used (text, graphic, image, ...), the point of view (ego/exo-centric, ...), etc.

The ASUR model in Figure 2 shows the interaction between a user and a 3D digital environment, using a “magic wand”. The user, User\_0, handles and moves a physical wand (Rtool) that is tracked by a camera (Ain). The camera sends the position of the wand to a digital Activator (Stool) that may act on other digital entities.

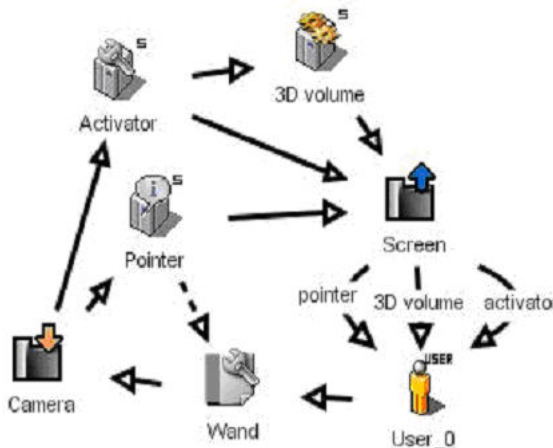


Fig. 2. An ASUR diagram example

It also sends the position to a pointer (Sinfo) object. The pointer is in fact a representation of the end point of the physical wand (dashed arrow); this representation is useful for providing interaction feedback. If the functionality is activated, data such as the rotation angle of the wand is transferred to a 3D volume object (Sobj). Finally the 3D volume, the activator and the pointer are displayed on a screen (Aout) to be perceived by the user (U). A more detailed description of this example, including all the modelled characteristics, is presented in [7].

An ASUR description of a mixed interactive system is thus useful in the early design phases to support the exploration and analysis of interaction designs. However it abstracts away features of software design and its implementation. Those two aspects are supported by a complementary model, ASUR-IL that stands for ASUR-Implementation Layer [6] and are out of scope of this paper.

Although ASUR captures the basic features of an interaction, it does not have the expressive power to say very much about the user's interactive activity or experience. It is this aspect that we have modified and which is presented in the remainder of this paper.

### 3 Modelling the Means of Interaction

One can use an application to communicate and/or receive information or to perform work (e.g., act on the world via the application). In this paper we use the term *interaction* to denote this kind of activity with an application and we characterise it as a sequence of information exchanges and/or actions between one or more users and one or more systems.

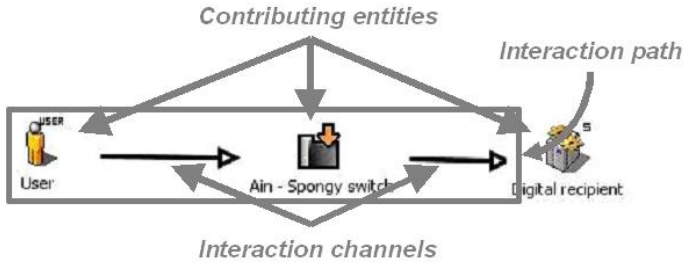
As described in section 2, any interaction originating from a user and that has at least one digital recipient (e.g., an application), or conversely, is mediated by an *adaptor*. Thus, to enter a name into an account record, one may need to use a keyboard. An interaction between two physical entities may be *mediated* by other physical entities. For example, one may use a stylus to interact with a PDA touch screen.

A sequence of such entities and their relationships used in an interaction forms an *interaction path*. The interaction exchange or action between elements in the path is conducted via one or more *interaction channels* along which information or action is communicated.

We begin this section with the description of a simple example. This is followed by a summary of the key elements in our ASUR extension which are then illustrated by applying them to the example. We then identify additional properties that can be expressed and explored on the basis of our model extension.

#### 3.1 A Running Example: A Spongy Switch

Let us consider a very simple, if somewhat unusual, example: a "spongy switch". An appropriately instrumented sponge might be used to communicate to some application one of two states: state 1 when the sponge is compressed and state 2 when the sponge is left uncompressed. At this stage in the analysis we do not yet specify how the compression is sensed, but merely that it can be. Figure 3 gives a simple ASUR diagram showing the entities and channels involved in this interaction path.



**Fig. 3.** ASUR diagram of a Spongy Switch

There are two contributing entities: a user and an instrumented sponge. The arrowed lines indicate that the user must act on the sponge in some way in order to change its state of compression and the resulting change of compression can be transmitted to some digital recipient.

### 3.2 Interaction Entities

The ASUR model of figure 3 includes three entities in the interaction path: the user, the spongy switch and the digital recipient of sponge state changes. In order to further describe interaction paths, we distinguish two types of entities: adaptors and mediating entities.

#### 3.2.1 Adaptors

By definition, an adaptor must perform a transform of the information on the incoming channel to that on the outgoing channel, one channel belonging to the physical environment and the other to the digital world.

This transformation can be simply an analogue to digital transformation but, in more abstract formulations of the adaptor, the transformation may perform other operations as well. In fact, the analogue to digital conversion is part of the definition of an adaptor. However, it's often useful to bundle this function with both sensing on the one hand and useful low-level transformations on the other hand. The level of abstraction is not fixed by our modelling technique but by the use to which the description is put.

For example, the adaptor used to localise the wand in figure 2 may either be in charge of grabbing a picture of the scene and detecting the presence of the wand (basic transformation from video capture to Boolean value) or grabbing a picture, detecting the presence of the wand and providing a digital recipient with a 4x4 matrix indicating the position and orientation of the wand. In this last situation, a converter is considered as part of the adaptor.

Presently, accelerometers, compasses, magnetometers, etc are supplied as special purpose devices and thus may be usefully modelled separately. However, we can anticipate that in the future these sensors will be fully integrated into mobile and pervasive devices such that they can be abstracted away as part of an interaction path. This is for example the case with Pan-Tilt-Zoom cameras that integrate automatic tracking of moving entities [1].

### 3.2.2 Mediating Entities

In figure 2, the physical wand manipulated by the user has no integrated mechanism supporting the encoding of its physical position into digital data. Therefore, it is not an adaptor. In this, as in most, cases, the physical entity constitutes an intermediate stage in the communication.

By definition, entities required to support intermediate stages in the communication and which are not themselves adaptors are mediating entities. We distinguish two different types of mediating entities: interaction carriers and contextual entities.

*3.2.2.1 Interaction Carriers.* Interaction carriers are mediating entities that are necessary for information communication. Carriers can

- provide a means of changing the user experience without changing the interaction functionality (e.g., the use of a stylus rather than a finger when interacting with a touch sensitive display),
- support “action at a distance” (e.g., a light pen) or
- act as a storage or feedback mechanism (e.g., handwriting on a piece of paper left as a trace by a digital pen).

The concept of interaction carriers can be further refined by identifying “active” and “passive” carriers:

- Active carriers are transmitters of non-persistent information along the interaction path. For example, a stylus transmits to a precise position on a touch screen a force generated by a user; the wand in figure 2 represents a position, etc.
- Passive carriers can carry, and store, part of the information communicated along the interaction path. For example, a tangible object left in a particular position on a table can serve as a physical storage device and the information might be picked up later via a camera.

*3.2.2.2 Contextual Entities.* In addition to interaction carriers there may be other physical entities involved in an interaction, such as the table on which the sponge may be placed. We call these contextual entities.

## 3.3 Characterising Interaction Paths

So far our description of the spongy switch in figure 3 doesn’t tell us very much of interest. For example, there is nothing yet to distinguish a user’s actual physical manipulation of this device from the manipulation of, say, a light switch. Furthermore, we cannot tell what sort of information the switch can communicate nor how the state of the sponge is sensed. We propose to anchor the required expressiveness in the description of the interaction paths.

Information paths are characterised by five basic properties. Two of the properties apply to the interaction channel itself, two others apply to the participating entities, one to the originating element and the other to the receiving element, and a final property applies to the entire path.

### 3.3.1 Channel Properties

*3.3.1.1 Medium.* When the interaction channel is physical (e.g. between a physical participating entity and an Ain, Aout or another physical participating entity), the medium is “the physical means by which the information is transmitted”, that is, a set of physical characteristics or properties, used to communicate information.

When the interaction channel is digital (e.g., from an Ain to a digital entity or between two digital entities) we may want to capture information about the nature of the connection, e.g., bandwidth, uptime, whether it is wireless (e.g., rf, infrared, 802.11), etc.

In figure 2, the medium of the interaction channel between the physical wand and the camera in charge of its localisation is visual; the tracking of the camera is a visual based detection.

*3.3.1.2 Representation.* This is a description of the coding scheme, or language, used to encode the information in the medium.

Note that there may be multiple levels of representation of the information. For example, a command to switch a light off or on might be represented as a sentence in a natural language, which is itself represented in auditory form for transmission to the input adaptor (modifying the medium; i.e., causing vibration in the intervening air). It is this auditory form which is used directly to modify the medium; the other representations (i.e., the natural language sentence and the operational command) may be formulated mentally by the user and subsequently may be extracted via an interpretation process by the input adaptor or other system components downstream from the adaptor.

In figure 2, the representation of the interaction channel between the physical wand and the camera in charge of its localisation is the position of the extremity of the wand in the physical space.

### 3.3.2 Properties of Participating Entities

*3.3.2.1 Method of modification.* This refers to the method of manipulating or otherwise affecting the medium. In the case of user-generated input, the user must act upon the medium to produce the state of the medium, or changes in its state, that are information encodings (i.e., that structure the medium according to the coding scheme). Similarly, an output adaptor must modify the medium of its channel. A speaker, for example, would use vibration of the speaker cone to set up vibrations in the air forming the medium of its channel to the user.

Mediating entities may also play a role here. In figure 2, when considering the interaction channel from the wand to the camera, the method of modification used by the wand onto the channel is the movement of the wand: movements of the wand affect the (visual) medium of the channel by changing the wand position (the representation).

In some cases the source of the information may perform no active modification of the medium. Consequently, the information is extracted from the channel via the active sensing process of an appropriately “stateful” sensor. For example, a camera (plus image processing) may be able to determine that some object in its field of view has not moved. This is perhaps the limiting or degenerate case of “affecting” the medium; i.e., the medium is “affected” by not being changed.

In the case of digital to digital channels, the method of modification is typically of no interest for purposes of interaction design. However, other related properties of the channel may be significant, e.g., push vs. pull; continuous vs. intermittent, average and peak load.

**3.3.2.2 Sensing Mechanism.** This depicts the device(s) and process(es) by which the state or changes in state of the medium are captured by the information recipient. In addition to a camera, as in the magic wand example, other typical sensing mechanisms include, among many others: pressure sensors, touch screens, microphones, cameras including integrated image processing, such as motion detection, accelerometers, graphical and tactile displays, speakers and earphones

If the communication has a user as the ultimate recipient, then sensing mechanisms include all the normal human perceptual channels.

### 3.3.3 Properties of the Overall Interaction Path

The properties presented so far express how an interaction path might communicate information or initiate action. The *intended user model* refers to what the user should know about the interaction in order to carry it out successfully. It may refer to one atomic interaction path (e.g., a channel plus its source and destination) or it may refer to more complex paths. We distinguish two parts of the intended user model: its core (or content) and its context.

**3.3.3.1 Intended User Model (Core).** This first dimension of the Intended User Model is the specification of the information that is intended to be communicated. This applies both to exchanges from and to the user. It is intended by the designer, ideally internalised and/or understood by the user and often indicated/represented in the system [9].

In the magic wand example, the core of the IUM of the interaction path between the user and the camera (through the physical wand) is the activation of the command that will affect the 3D volume. The user has to know that manipulating the wand is required to activate the command.

**3.3.3.2 Intended User Model (Context).** The contextual IUM refers to all other pieces of user knowledge necessary to carry out the interaction successfully. This might include being aware of associations (“clicking on this button will cause that object to disappear”) or understanding the mechanism by which the interaction is realised (“my face is being captured by that camera”).

In the magic wand example, the contextual IUM must include the boundaries of the physical space in which the wand is localised and outside of which the wand is no longer visible and can no longer be used to activate a command.

## 3.4 Path Properties of the Spongy Switch

### 3.4.1 Applying the Model to the Spongy Switch

Each of the relationships, between user and sponge and between sponge and sensor, can be characterised using our path properties. The model illustrated in figure 3 has an interaction path consisting of one originating entity (the user), one channel and one

recipient entity (the instrumented sponge). We have, however, also shown, an additional channel, linking the sponge to some receiving digital recipient (e.g., a concept or component in the application). To keep this example simple, our description of the path properties will apply to the user-sponge path only.

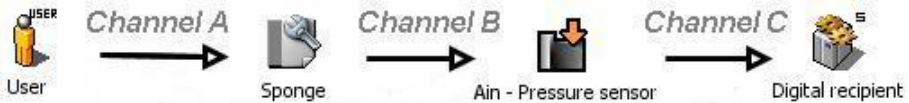
We shall use a simple table to present the path properties:

**Table 1.** Path properties of the Spongy Switch

<b>Medium Representation</b>	The state of compression of the sponge A set of discriminable compression values. For our example, we will choose two, compressed and uncompressed.
<b>Modification Method</b>	The sponge’s compression state is modified by the user squeezing or releasing the sponge.
<b>Sensing IUM</b>	Sensing is via a pressure sensor embedded in the sponge. At this stage, without yet having contextualised the interaction technique, the intended user model can only be described as communicating one of two discrete states, otherwise uninterpreted. In the context of a digital whiteboard, squeezing the spongy switch might correspond to selecting the eraser.
<b>Other properties</b>	

**3.4.2 Refining the Spongy Switch Model**

*3.4.2.1 Decomposition.* The spongy switch description so far does not separate out the sponge from its pressure sensor; they are treated as a single integrated entity. It’s often useful to treat a complex interaction device or mediating entity in this way, abstracting over its internal composition. However, it can also be useful at times to refine the description, revealing details of its internal structure as illustrated in Figure 4. In this case we have two channels, one from the user to the sponge and one from the sponge to the adaptor (pressure sensor). We leave channel C unspecified here for purposes of simplicity.



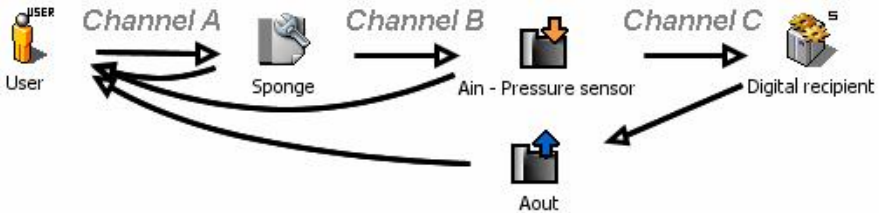
**Fig. 4.** A Refined Diagram showing Spongy Switch Internal Structure

Channel A has sponge compression as its medium and channel B has sponge internal pressure as its medium. Notice that now there is a transformation from the channel A representation to that of channel B (i.e., from states of compression to pressure states). Indeed, it may well be that the user is capable of placing the sponge in a number of different degrees of compression (i.e., the channel A representation has more than 2 states) but that the sensor can only recognize two different levels of pressure. Additionally, the subsequent channel, from sensor to digital recipient, may itself have a different resolution, with the sensor reducing the number of discriminable states communicated on that channel.

**Table 2.** Path properties of the refined description of the Spongy Switch

	<b>Channel A</b>	<b>Channel B</b>
<b>Medium</b>	The state of compression of the sponge	Internal pressure of the sponge.
<b>Representation</b>	A set of discriminable compression values. For our example, we will choose two, compressed and uncompressed.	A set of discriminable pressure values.
<b>Modification Method</b>	The sponge’s compression state is modified by the user squeezing or releasing the sponge.	None
<b>Sensing</b>	None	Sensing is via a pressure sensor.
<b>IUM</b>	There is one intended user model, which is the same as the unrefined path (see section x).	
<b>Other properties</b>		

3.4.2.2 *Feedback.* So far, we have only shown an input path. Clearly, feedback paths are necessary. Figure 5 illustrates a possible design, identifying three paths, one at the physical level, one that indicates the interpretation of the sponge manipulation and a final one that presents the results of application significant operations. We revisit this topic in section 4.2.1. on a more concrete example to analyse these feedback paths.



**Fig. 5.** A second Refined Diagram showing Spongy Switch Feedback

### 3.4.3 Exploring a Design Space

Our spongy switch description, although still very simple, already enables us to begin exploring an interaction design space. We can find alternative entities that will function similarly within an interaction path unchanged with respect to its information communication properties. For example, if the medium of channel B becomes visual, while its representation and the associated method of manipulation remain unchanged, the sensing mechanism might be changed to image capture. In order to leave channel C unchanged, the Ain also has to be able to derive a level of compression value. To satisfy these new design options, the pressure sensor might be replaced by a camera, positioned to capture the shape of the sponge and that encapsulates a “sponge shape to compression level mapping system”. Since channels A and C remain unchanged, this replacement can safely be made without either changing the user experience or the



interaction functionality, as modelled<sup>2</sup>. Such a replacement, leaving the user experience (including the distinctive aspects of squeezing a sponge) the same and the system functionality unchanged could be useful in order to reduce implementation costs (no need to construct a special sponge) or to increase mobility (no need for wire dangling from the sponge).

### 3.5 Refining the Properties of the Interaction Path

So far we have not addressed the question of the level of abstraction of the descriptions in a characterisation of an interaction path. The level in the spongy switch example is perhaps sufficient to communicate a reasonably concrete design solution. However, there are likely to be features of the interaction which need further refinement, either to make the specification sufficiently precise to be implemented or to identify key features affecting its usability. For example, one will need to know how many discriminable compression states are necessary for the application and are achievable with a particular sponge/sensor combination. Additionally, the weight of a participating entity may be significant; the lightness of a sponge might make it a good candidate for elderly users who have weak muscles.

We believe that this refinement will normally occur as part of an iterative process. In the early stages of a design, we may simply identify the need to output an image of a digital object. However, this is not sufficient for an implementation, for which additional details of the form of rendering will be needed.

## 4 Studying Interaction Groups

So far we have introduced the principles and characteristics of the interaction path concept and illustrated them via a simple example. In this section we use a different example to introduce the notion of *interaction groups*, built on top of interaction paths, and we explain how that notion can be used to capture and analyse design alternatives.

### 4.1 The PDA Balloon Case Study

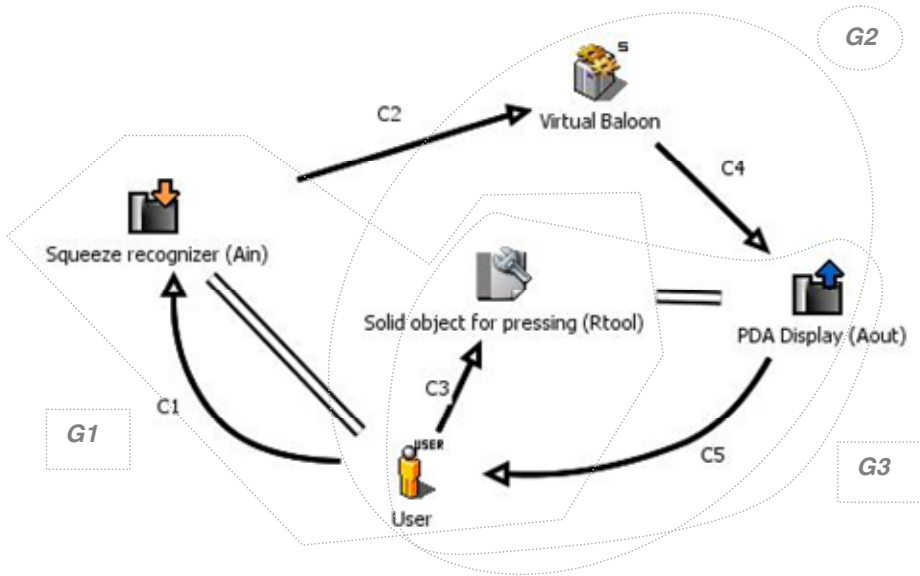
#### 4.1.1 Overview

Our second example is based on an interaction technique developed to demonstrate the use of sensors as captors [11]. This interaction technique involves a user; an adaptor for input that is able to capture, analyse and identify squeezing actions of a user; a PDA that the user holds in the hand; and the PDA's display, used to present representations of digital entities.

Based on this interaction technique, an application has been designed to enable a user to interact with a virtual balloon. Figure 6 presents the basic ASUR model of this application. The virtual balloon is presented via an image on a PDA display. The user can change the balloon size by "inflating" it; this is carried out by squeezing on the PDA case (denoted as "solid object for pressing"). Each squeeze will increase the size of the balloon by one level.

---

<sup>2</sup> Of course, there may be other properties of the interaction which would impinge on the two end channels. For example, the video might be slower, use more bandwidth and might be a disturbing presence in the user's environment.



**Fig. 6.** Basic ASUR diagram for the PDA balloon and interaction groups (cf. section 4.2)

The most interesting feature of this application is the indirect relationship between the user’s squeeze and the system’s sensing of that event. There is no sensor on the PDA; rather an accelerometer is placed on the user’s forearm<sup>3</sup>. This accelerometer detects muscle tremor. The squeeze action increases this tremor and the Ain uses an algorithm for recognising the distinctive tremor pattern associated with a squeeze that is sufficiently strong.

The next section details the characteristics of the interaction channels that will be referenced when illustrating the subsequent interaction group analysis.

#### 4.1.2 Channel Descriptions

Channel C1 represents the link between the user and the “Squeeze recognizer” (Ain). In the particular use of this interaction technique, inflating a digital balloon, there is no intended user model since the sensor is intended to be completely invisible for the user. The digital channel C2 is required to transmit the captured information to the digital resource that manages the digital balloon. Channel represents an interaction the purpose of which is simply to motivate the user to generate muscle tremor. However, in order to produce an effective design the method of generating this tremor must be appropriately linked semantically to the notion of balloon inflation; we shall return to this issue in section 4.2 below.

<sup>3</sup> In the original technique developed by Strachan et al. [11], the tremor sensor is actually mounted onto the PDA-case. In order to better illustrate the grouping mechanism presented here, we use a slightly different design technique, in which the tremor sensor is indeed fixed on the user’s arm.

C4 is another digital channel required to present properties of the digital balloon through an adaptor for output. In our case, the property of interest is the size of the digital balloon. Finally, channel C5 transfers data that will be perceived by the user; the information carried by the channel must represent the size of the balloon. Table 3 summarises all the characteristics of these interaction channels.

**4.2 Interaction Groups**

We use the term ‘interaction group’ to refer to a set of entities and channels that together have properties that are relevant to a particular design issue. As will become evident, there are typically many such interaction groups that can be identified for a particular interaction design. Some of these groups will be universal (applicable to any design) while others will depend on the task and context or on the requirements of an analysis performed by a specialist (ergonomist, ethnographer, device designer, software engineer). For example, entities or channels that represent or transfer information about a single common concept or that share the same type of constraints may form an interaction group. The set of all interaction groups for a given design forms a potentially complex graph of associations, with different views for different purposes. Via the PDA balloon example, we present several different groupings that exemplify the sorts of groups that are likely to be of interest for many interaction designs.

**Table 3.** Characteristics of the interaction channels of the PDA balloon

<b>Channels</b>	<b>C1</b>	<b>C2</b>	<b>C3</b>	<b>C4</b>	<b>C5</b>
<b>Medium</b>	Muscle tension	Digital	Pressure on the Rtool	Digital	Light
<b>Representation</b>	Recognizable tremor pattern	ONE discrete command	2 values: squeezing / grasping	Set of balloon sizes	Image of balloon
<b>Modification Method</b>	Tremor	Not relevant	Hand squeezing	Not relevant	Light modulation
<b>Sensing</b>	Accelerometer	Specific API	Null	Specific API	Visual
<b>IUM</b>	None: the user is not supposed to be aware of it	N/A	Single hard squeeze inflates by one level.	Not applicable	Size of a balloon
<b>Other interesting properties</b>	Granularity of the squeezing detection	Wired or Wireless connection		Property of interest: balloon size	Attributes of the chosen representation

### 4.2.1 Grouping for Feedback

Grouping for feedback aims at identifying entities and channels involved in an interaction flow linking the response of the system to actions of the user. To promote this group and ensure that the feedback will be effectively perceived, it is important to consider the definition of the characteristics of these channels as a whole: for example if using audio as modification method of C5, it is probably not adapted to adopt the same modification method for C1 (loop) since both channels will be used almost simultaneously each time. In addition, it is also important to clearly differentiate the characteristics of these channels from those of other channels involved in the interaction but not in the feedback group.

In the case of the PDA balloon, one feedback group includes all the channels: C1 and C3 in parallel, C2, C4 and C5. As a consequence, C2 and C4 must persist throughout the interaction and must not be interrupted due to, say, a poor WiFi connection. It is also important that the different values of the representation carried by C4 are correspondingly represented via C5.

Channels C3 and C5 alone also form a separate articulatory feedback group because acting on the Rtool through C3 automatically triggers effects perceivable via C5. However, feedback is only one of the features of this grouping; we will discuss it again in section 4.2.4.

### 4.2.2 Grouping Based on Coherence among Properties

Some groups join together elements with related properties in order to generate a coherent effect, such as visual continuity. For example, a grouping might associate a set of channels and assert that they must all use the same medium (e.g. Visual) or indeed must use different media (e.g., visual and sound) in order to provide perceptual continuity.

A first example of this sort of grouping for coherence is the group called G1 on the PDA balloon diagram (see figure 6). This group, identified at the design level, consists of the User, the PDA case (Rtool), the accelerometer (Ain), and channels C1 and C3. It is based on coherence between the modification method of channels C1 and C3. Therefore, a change to one of these modification methods must ensure that this coherence property is maintained: that is, the modification method of C1 (i.e., tremor) must be an indicator, effect or co-occurrence, of the modification method of C3 (squeezing with hand). In other words, by grouping the two channels, we are saying that they work together as a single mechanism and it clearly expresses that these elements used simultaneously enable the inflation activity.

An alternate implementation solution might consist of changing the adaptor to a camera, thus changing C1 so that it captures visual properties of the user's modification method in C3. Clearly, for this to be acceptable, the modification method properties of C1 (e.g., visual deformation of the muscles in the forearm or characteristic distortion of the Rtool) must correspond to the squeeze manipulation of C3.

A second example is the group G2 that consists of the digital balloon, the PDA display and case, the user and channels C3, C4 and C5. The group identified is based on coherence among the Intended User Model of the involved channels. G2 captures the notion that the user interface elements (i.e., PDA display and PDA case) together form a representation of the presented concept (the virtual balloon). Ensuring a coherent

IUM among channels of this group may also be reinforced by applying constraints or associations to other properties of the channels, such as:

- the PDA display must show a visual representation of the virtual balloon, i.e., something that looks like a balloon (constraint) and its displayed size must correspond to the virtual balloon size property (association).
- the method of manipulation of C3 should correspond to the method of manipulation of a real balloon, to reinforce the association of the action with the intended type of virtual object expressed via the “other properties” part of C3 (association)

More generally groups of interaction paths based on coherence among properties could refer to any single common property or set of properties shared over several interaction paths or channels (e.g., all of the paths that use video sensing mechanism, all those that use grasping, all those that participate in the same IUM, etc.). The potential force of analysis based on these groups is that it allows the specification and refinement of the different forms and levels of articulatory, perceptual and cognitive continuity that may be considered when evaluating an interactive system.

### 4.2.3 Action and Effect Association

This expresses a semantic association that links user interface elements to certain application concepts. The goal is generally to help the user to cognitively unify elements of the groups. Such grouping can lead to requirements on several properties of the elements in the group.

G3 is such a group in the PDA balloon example. The group consists of User, PDA case, PDA display plus C3 and C5. This group is not only a feedback group. Indeed, the purpose is to unify the actions on the PDA case with the resulting effects presented in the PDA display to help the user associate the squeeze on the case as the *cause* of the inflation. There are three aspects of this grouping that serve to reinforce the cognitive association of the action and the effect:

- the physical closeness of PDA case and PDA display (represented by the physical proximity relationship on the ASUR diagram,
- the feedback loop of C3 followed by C5 and
- the fact that the PDA case and the PDA display are both in the user’s visual field at the same time (this property is not directly expressible in the diagram; however this could be added to “other properties” of C3 and C5).

### 4.2.4 Other Groupings

While we have examined several interaction groups arising from an initial analysis of our simple example, the value of the interaction grouping concept is potentially much greater. Part of our future work is to further explore the sorts of purposes to which interaction groupings can be put. Among potential groups of interest are sets of inputs that must be combined to perform some task, e.g., a speech input with a gesture input (“put that there”). This would correspond to a form of grouping for multimodal coordination.

The correspondences expressed in G1 motivate a sub-grouping of the Rtool and the Ain entities to create an “abstract instrument” with a single perceived input channel, C1, and a single output channel, C2. This concept of “abstract instrument” need further investigations but constitutes another form of grouping and establishes a clear parallel with the notion of instrument in the instrumental interaction [2].

A grouping for distribution / communications might be used to assert that a set of services/concepts must reside on the same machine or indeed be distributed or use a common form of communication.

In the case of collaborative systems, groupings of paths may show information flows among or between users. Additionally, agronomists may want to group physical devices and their locations.

## 5 Relationship to Other Models

Card et al's input modelling language is perhaps the closest to our model in its attention to the physical and concrete aspects of an interaction [3]. However, we are interested in embedding this aspect into a larger descriptive framework that includes both the physical context, feedback loops and its role in information exchanges with an application.

Our information-exchange model could be deemed a variant of instrumental interaction [2]. We have pointed out, for example, how our interaction group mechanism can be used to specify abstract interaction instruments. However, our model highlights and refines the informational and physical aspects of the interaction. Consequently, our model can be considered complementary, and a possible addition to or refinement of, the instrumental interaction model.

Coutrix & Nigay [4] offer a recent approach that, like our model, combines both the physical and digital dimensions of the interaction. Their interest is primarily in the transformations of information through mediating software components that together express interaction modalities. Our approach, however, includes a richer description of the interaction from the point of view of a user's manipulative and perceptual actions and their relationship to a user's intentions. Again we believe that these are complementary descriptions that could benefit from being used together.

Smith [10] applies a flownet model to the description and analysis of design-significant features of a system involving haptic interaction. This model, like ours, is designed to enable low-cost exploration of concrete interaction design issues such as the continuity of physical actions and the coherence and adequacy of feedback. Smith's approach, unlike ours, can deal with the dynamics of interaction and, indeed, it would be interesting to add flownet semantics to our model to augment this aspect. However, Smith's model does not include an intended user model nor our feature of (potentially extensible) interaction groups. Additionally, Smith's model stops at the point of a user's generation of input and/or consumption of output and thus does not capture the role of mediating objects. As we have suggested above, differences between sufficiently similar models, such as those just noted, offer opportunities for cross-fertilisation.

## 6 Conclusions

The model we have presented in this paper takes seriously the fact that interaction is both a concrete phenomenon, embedded in a physical context, and also a complex combination of information exchanges that support activity in a mixed physical/digital world. It picks out aspects that are potentially design significant and organises them in a way that is intended to facilitate design reasoning (e.g., making comparisons between choices of device, identifying new solutions, finding problems).

We have developed our model in order to refine and enrich an existing model, ASUR. We have found this approach to be fruitful and the new elements introduced here seem to fit comfortably with the original model. However, it remains to be determined if this association relies on some fundamental connection between the original ASUR notion (i.e., component-based composition, interaction-centred viewpoint) and these new concepts related to physically realised interaction channels. In other words, it may be possible to take ideas from our approach and use them to augment other models, such as those referred to in section 5.

The ASUR interaction model is intended to be very high level. It is not intended to capture the way in which the information communication is structured or realised via actual interactors or dialogue sequences in particular languages. It is designed to focus on key aspects of the interaction from the point of view of features that need to be identified early in the design process. Further work is needed to link descriptions of interaction using our model into a development process leading to effective implementations.

**Acknowledgments.** The authors wish to acknowledge the award by the University Paul Sabatier of Toulouse of a visiting professorship to Philip Gray during the period March- September 2006. This award helped make possible the collaboration resulting in this paper.

## References

1. Auto Tracking VB-C50i Network Pan/Tilt/Zoom Camera by Canon, <http://www.nuspectra.com/detail.aspx?ID=1078>
2. Beaudouin-Lafon, M.: Instrumental Interaction: an Interaction Model for Designing Post-WIMP User Interfaces. In: Proc. CHI 2000, pp. 446–453. ACM Press, New York (2000)
3. Card, S.K., Mackinlay, J.D., Robertson, G.G.: A morphological analysis of the design space of input devices. *ACM Trans. Inf. Syst.* 9(2), 99–122 (1991)
4. Coutrix, C., Nigay, L.: Mixed Reality: A Model of Mixed Interaction. In: Proceedings of AVI 2006, Venezia, Italy, 23-26 May 2006, pp. 43–50. ACM Press, New York (2006)
5. Dubois, E., Gray, P.D., Nigay, L.: ASUR++: a Design Notation for Mobile Mixed Systems. In: Paterno, F. (ed.) *Interacting With Computers*, vol. 15, pp. 497–520 (2003)
6. Dubois, E., Gauffre, G., Bach, C., Salembier, P.: Participatory Design Meets Mixed Reality Design Models - Implementation based on a Formal Instrumentation of an Informal Design Approach. In: Calvary, G., Pribeanu, C., Santucci, G., Vanderdonckt, J. (eds.) *Computer-Aided Design of User Interfaces V. Information Systems Series*, pp. 71–84. Springer, Heidelberg (2006)
7. Dupuy-Chessa, S., Dubois, E.: Requirements and Impacts of Model Driven Engineering on Mixed Systems Design. In: Gérard, S., Favre, J.-M., Muller, P.-A., Blanc, X. (eds.) *Proceedings of IDM 2005*, Paris, France, pp. 43–54 (2005)
8. Ishii, H., Ullmer, B.: Emerging Frameworks for Tangible User Interfaces. *IBM Systems Journal* 39, 915–931 (2000)
9. Norman, D.: *Cognitive Engineering*. In: *User Centered System Design: New Perspectives on Human-Computer Interaction*, pp. 31–61. Lawrence Erlbaum Associates, Mahwah (1986)
10. Smith, S.P.: Exploring the Specification of Haptic Interaction. In: Doherty, G., Blandford, A. (eds.) *Proc. DSV-IS 2006*, Dublin, pp. 146–159 (2006)
11. Strachan, S., Murray-Smith, R.: Muscle Tremor as an Input Mechanism. In: *Proc. UIST 2004*, vol. 2 (2004)

## Questions

**Prasun Dewan:**

*Question: Are you expecting to build tools for verifying that the modelled properties are actually implemented?*

Answer: This is not currently planned: this is meant to be a lightweight means of reasoning. Verification isn't on our current to-do list.

*Question: What is the practical use of it?*

Answer: As a means of communicating the characteristics of the application.

**Laurence Nigay:**

*Question: In your example, the key problem is that the user cannot observe the input mechanism. Did you think about honesty or observability?*

Answer: I don't think the lack of awareness of the channel is a problem.

**Panos Markopoulos:**

*Question: It would be interesting to see if the reasoning power that this approach delivers actually helps designers?*

Answer: That's proposed in the validation stream.



# On the Process of Software Design: Sources of Complexity and Reasons for Muddling through

Morten Hertzum

Computer Science, Roskilde University  
Roskilde, Denmark  
mhz@ruc.dk

**Abstract.** Software design is a complex undertaking. This study delineates and analyses three major constituents of this complexity: the formative element entailed in articulating and reaching closure on a design, the progress imperative entailed in making estimates and tracking status, and the collaboration challenge entailed in learning within and across projects. Empirical data from two small to medium-size projects illustrate how practicing software designers struggle with the complexity induced by these constituents and suggest implications for user-centred design. These implications concern collaborative grounding, long-loop learning, and the need for a more managed design process while acknowledging that methods are not an alternative to the project knowledge created, negotiated, and refined by designers. Specifically, insufficient collaborative grounding will cause project knowledge to gradually disintegrate, but the activities required to avoid this may be costly in terms of scarce resources such as the time of key designers.

**Keywords:** User-centred design, Design process, Software development, Software-project complexity, Muddling through, Collaborative grounding.

## 1 Introduction

Software design is replete with projects that are cancelled, late, over budget, or result in systems with fewer features than originally specified [e.g., 5, 20]. Further, large numbers of systems are rejected by users or produce a merely marginal gain over former systems and work practices [e.g., 14, 28]. As an example, a recent national system for the Danish public administration was more than 100% late, more than 50% over budget, and reduced employee productivity by about 50% for several months after it was released. Six months after release an expert assessment concluded that considerable revisions of the system were immediately necessary, increasing the over-spending to almost 100% compared to the original budget [12]. Troubled projects come about in spite of concerted efforts to the contrary, and they demonstrate the complexity of software design. Managing this complexity requires that its core constituents are well-understood.

This study analyses three constituents of software design and illustrates the analysis with empirical data from two projects. Each of the constituents is indicative of

considerable complexity and – unless managed – entails serious risk to successful project completion. The analysed constituents of software design are:

- *The formative element*, which concerns articulating and reaching closure on a design
- *The progress imperative*, which concerns making estimates and tracking status
- *The collaboration challenge*, which concerns learning within and across projects

The formative element is at the core of human-computer interaction (HCI) and the two other constituents are crucial characteristics of the context in which practical HCI work takes place. Whereas the progress imperative has been acknowledged in much HCI work, for example the work on discount usability engineering [31], the implications of the collaboration challenge have not received nearly the same attention. This study aims to outline implications for user-centred design resulting from an analysis of the three constituents. For HCI researchers, the study intends to point out issues that may seem mundane but nevertheless hamper real-world projects, at least small to medium-size projects. For HCI practitioners, the study identifies some of the problems and tradeoffs they face in their work, and thereby offers an opportunity for reflection and pointers to means of alleviating some of the problems.

## 2 Empirical Data

To illustrate how practicing software designers approach the three software-design constituents that are analysed in this paper empirical data were collected from two software projects. The two projects are small to medium-sized and in this sense represent the majority of software projects [8, 17]. Neither of the organizations in which the projects took place follows a mandated design method but they have successfully completed a range of software projects.

The first project concerns a browser interface to a document-management system. Over a period of two decades the organization has developed, marketed, and continuously evolved a generic document-management system. The organization has 120 employees and a base of more than a hundred longstanding customers. Thousands of people use the document-management system on a daily basis. One high-level goal of this system is to provide professionals, as opposed to secretaries and document clerks, with easy access to organizational documents. In support of this goal it was decided to develop a browser interface to the system. The browser-interface project involved three designers and was successfully completed in seven months. The project was completed on time and within budget but this was partly achieved by reassessing and reducing the functionality of the browser interface halfway through the project.

The second project concerns a common user-interface platform developed by an organization that started by providing consultancy in hydraulic engineering but now increasingly develops and sells software instead of or along with the consultancy. The organization has 270 employees and has undertaken projects in more than a hundred countries. Over a period of three decades the organization has developed a number of hydraulic models and modelling tools as standalone software applications, but these applications generally have crude and inconsistent user interfaces and they must be ported individually to new operating systems. To mitigate these drawbacks a project

was established to provide a common user interface for the applications and handle their interaction with the operating system. The project, which involved 10-15 persons, took longer than planned and consumed more resources, but it was eventually completed.

For both projects two designers – the project manager and a programmer – were interviewed for a total of three hours. The obtained data are retrospective, though both projects were completed recently. In this sense the empirical studies are like post-project reviews. The interviews, which were audio recorded and subsequently transcribed, were loosely structured by a set of guiding questions. These questions concerned the major difficulties and information needs experienced during the project and the means in place to handle these information needs and communicate lessons learned. The interviewees' statements were compared and contrasted for purposes of validation. All interviewees were for the most part positive about their project but they also raised critical issues. Toward the end of the interviews, the interviewees were asked about their views on what had been the most significant risk factors in their project. This part of the interviews was based on a walkthrough of the 11-item list of top software-project risks identified by Schmidt et al. [36].

### 3 Three Constituents of Software Design

The project knowledge created, utilized, modified, embodied, shared, sought, and otherwise relied upon by designers must enable them to manage three complex and interrelated constituents of software design: the formative element, the progress imperative, and the collaboration challenge. Mapping these three constituents of software design to the lists of top software-project risks identified by Boehm [4] and Schmidt et al. [36] shows that the three constituents encompass the bulk of complexity that must be managed in software projects (Table 1). Of the 21 top risks on either of the two lists ten concern the formative element, five the progress imperative, and three the collaboration challenge. Only three risks, about limitations of technology, are not covered by the three constituents.

#### 3.1 The Formative Element

The formative element is about articulating and reaching closure on a coherent design. After discussing this constituent of software design it is illustrated with data from the two empirical studies.

**Articulating and Reaching Closure on a Design.** The need for new systems can manifest itself in manifold ways, such as dissatisfaction with present ways of working, demands for new outputs, and knowledge of new technological options. This initial need provides only a vague or high-level specification of what is required from a new system and, consequently, software design involves a process of articulating the requirements toward the system in detail. The task-artefact cycle (Fig. 1 [9]) illustrates this cyclic and nontrivial process, in which designers respond to user requirements by building artefacts, which in turn present or deny possibilities to users. Users' understanding of their current artefacts is shaped by the tasks for which they are using the artefacts and, at the same time, their understanding of their tasks is shaped by the

**Table 1.** The coverage of the three constituents of software design in terms of the top software-project risks identified by Boehm [4] and Schmidt et al. [36]

Constituent	Boehm's top-10 [4]	Schmidt et al.'s top-11 [36]
The formative element: articulating and reaching closure on a design	<ul style="list-style-type: none"> <li>▪ Continuing stream of requirements changes</li> <li>▪ Developing the wrong functions and properties</li> <li>▪ Developing the wrong user interface</li> </ul>	<ul style="list-style-type: none"> <li>▪ Changing scope/objectives</li> <li>▪ Misunderstanding the requirements</li> <li>▪ Lack of frozen requirements</li> <li>▪ Lack of adequate user involvement</li> <li>▪ Failure to gain user commitment</li> <li>▪ Failure to manage end-user expectations</li> <li>▪ Conflicts between user departments</li> </ul>
The progress imperative: making estimates and tracking status	<ul style="list-style-type: none"> <li>▪ Unrealistic schedules and budgets</li> <li>▪ Gold-plating</li> <li>▪ Shortfalls in externally furnished components</li> <li>▪ Shortfalls in externally performed tasks</li> </ul>	<ul style="list-style-type: none"> <li>▪ Lack of top-management commitment to the project</li> </ul>
The collaboration challenge: learning within and across projects	<ul style="list-style-type: none"> <li>▪ Personnel shortfalls</li> </ul>	<ul style="list-style-type: none"> <li>▪ Insufficient/inappropriate staffing</li> <li>▪ Lack of required knowledge/skills in the project personnel</li> </ul>
Other: limitations of technology	<ul style="list-style-type: none"> <li>▪ Real-time performance shortfalls</li> <li>▪ Straining computer-science capabilities</li> </ul>	<ul style="list-style-type: none"> <li>▪ Introduction of new technology</li> </ul>

artefacts they currently use. Likewise, designers' understanding of the technological options is shaped by their knowledge of tasks that need to be performed and, at the same time, their understanding of users' tasks is shaped by the possibilities and restrictions of the artefacts they currently know of. Thus, people's familiarity with certain artefacts and certain tasks shape their understanding of what their tasks are and what technology has to offer, and this understanding, in turn, constitutes a perspective that points to certain technological options and makes people blind toward others [30]. This makes it inherently difficult for people to transcend their current way of perceiving things and envision how tasks, users, and technology should interact in constituting the future use situation.

The information needs inherent in the task-artefact cycle concern three areas of knowledge [27]: the users' present work, the technological options, and the new system. In a sense, the users' present work and the technological options are only of

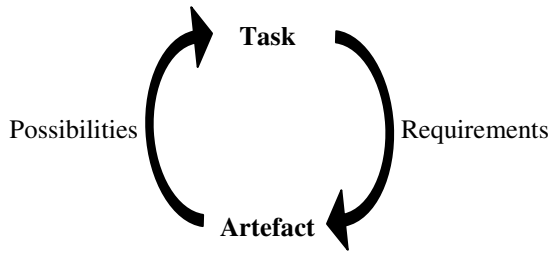


Fig. 1. Task-artefact cycle

interest because designers have no direct way of getting information about the new system and use situation. This is interesting from a project-knowledge point of view because it points out the massive indirectness of the information-seeking process in software design. Designers seek information about the users' present work, as opposed to their future work, and the technological options, as opposed to the future system, because they have no direct way of getting the information they really need. When designers are asked to design a new system they are, at the same time, prevented from getting crucial information about what properties this new system should have because people's familiarity with their present tasks and artefacts blocks their ability to envision radically new solutions. Further, software projects are frequently hampered by fluctuating and conflicting requirements because the learning process inherent in the task-artefact cycle continues throughout the projects and because the needs of different stakeholders may point toward different designs [4, 10, 36]. Apart from untangling these issues, which add to the difficulties of reaching convergence on a common project vision, requirements must not only be articulated they also need advocates. These advocates can be designers, users, or other people involved in a project. Eodice et al. [16] divided the requirements in a project they studied into those with and those without an advocate. They report that whereas virtually all the requirements with an advocate were eventually implemented not a single one of the requirements without an advocate were implemented.

Potts and Catledge [34] find that the process of reaching closure on the design of a new system is painfully slow and punctuated by several reorientations of direction. Lack of an agreed-upon understanding of what a system is to achieve complicates the development process because it leads to disagreements among designers as to the focus of the system and the best utilization of their resources. As a result, users may not be provided with any good system image [32] that presents the system facilities and their interrelationships in a clear and coherent manner. To provide insight about the use situation and thereby obtain a good match between user needs and system image prospective users must be actively involved in articulating and reaching closure on a design [e.g., 3, 18, 19]. At the same time requirements articulation is also a negotiation process in which designers need some level of control over the scope of projects to be able to balance their management of the contractual aspect of requirements specification against the facilitation of users in an open-ended search for requirements [23].

**Browser-Interface Project.** Two of the three designers involved in the browser-interface project had considerable knowledge of the users' work domain from previous

projects and could, thus, readily enter into discussions of requirements. The initial forum for these discussions was an annual two-day customer seminar hosted by the development organization to get feedback on released systems and discuss needs and ideas for new system facilities. For one of these seminars, which are attended by about 300 persons, a free-lance consultant made a prototype of a browser interface. Based on the feedback and discussions at the seminar it was decided to make the browser interface a top-priority project. This project was to provide platform-independent access to the document-management system without the need for installing additional software on users' computers. Further, the browser interface should be sufficiently undemanding to be usable without formal training, in contrast to the primary interface which requires a two-day course. While these high-level goals were clear from the outset a more detailed requirements document was never produced. Rather, the designers started coding early on and kept the evolving design partly in their heads and partly reflected in the code they produced. The intermediate outcomes of their work, in the form of system prototypes, were presented to and discussed with a group of user representatives with whom the designers met 4-5 times during the project. This led to the identification of a series of more detailed requirements, but the primary interface of the document-management system provided a default structure that significantly reduced the uncertainty and complexity involved in specifying the browser interface. The presence of the primary interface may, however, have rendered the designers and user representatives blind toward new possibilities and solutions. In continuation of this, one of the interviewees was concerned that the user representatives did not experience the prototypes in sufficient depth at the meetings and that actual use of the released browser interface might, therefore, give rise to many new requirements and change requests.

**Common-Platform Project.** At the overall level the common-platform project had a clear product vision from the very start, namely to provide a common, state-of-the-art graphical user interface for the individual hydraulic-engineering applications. Initially, the key person on the project was knowledgeable about both the hydraulic engineering that forms the basis for the applications and the user-interface programming that forms the basis for the common platform. This person has, however, left the organization and the remaining people on the project knew little about hydraulic engineering. Though the project members continually interacted with colleagues knowledgeable about hydraulic engineering this interaction was largely informal and the outcomes of these interactions remained in the heads of individual project members. No requirements specification was produced, discussed, iterated, and agreed upon, and apart from some code-level documentation the only up-to-date design documentation has been the project members' personal notes. The absence of systematic user involvement and requirements analysis provides strong candidate reasons for two of the three software-project risks identified by the interviewees as particularly relevant in relation to this project: failure to gain user commitment and failure to manage end-user expectations. The absence of design documentation such as an agreed-upon requirements specification also entailed that the project members were not supported in maintaining a shared understanding of the scope and objectives of the project. As a consequence there was no authoritative source in discussions about the functionality expected from different software modules and the project members repeatedly experienced difficulties in determining whether and when a module was complete.

**Reasons for Observed Practices.** Recommendations about how to articulate and reach closure on a design include principles such as “early focus on users and tasks” [18], techniques such as interpretation sessions [3], and artefacts such as requirements specifications. While such recommendations have been advocated for decades they are often not followed in practice [18, 34]. In the browser-interface and common-platform projects the main reasons for using proven design practices only sparingly were:

- *Believing high-level project goals are sufficient.* High-level goals like “providing platform-independent access to the document-management system” may provide a product vision but without complementary details the design is severely under-specified. Nevertheless, the designers in the two studied projects seemed to consider the high-level goals a satisfactory specification of their work in that they made no concerted effort to involve prospective users in producing a more detailed requirements specification.
- *Not knowing how to bring about more detailed requirements.* The designers seemed uncertain about how to get detailed requirements information from users and whether users would be able to provide such information. In the browser-interface project this uncertainty also included a fear of losing control over the process; that is, of eliciting requirements that went substantially beyond what they had the resources to deliver.
- *Focusing on the tasks they know best.* In a situation characterized by uncertainty and schedule pressure the designers concentrated on the tasks they knew how to do, primarily coding. This gave rise to a sense of progress though they were aware that important activities were being glossed over.

These reasons suggest that if given a structured process of clearly defined tasks for working systematically with requirements, designers will tend to follow this process [25]. But until such a process has become an established part of their repertoire many designers will likely muddle through the activities involved in articulating and reaching closure on a design.

### 3.2 The Progress Imperative

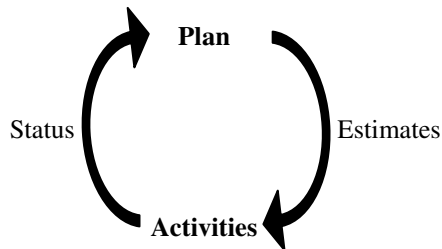
The progress imperative is about making estimates and tracking project status. After discussing this constituent of software design it is illustrated with data from the two empirical studies.

**Making Estimates and Tracking Status.** DeMarco [13] states that without estimates software projects cannot be managed. Estimation is a prerequisite for project planning which, in turn, provides for the coordination and management of design activities. Accurate estimates are, however, hard to make because the cost and time of developing both software modules and complete systems depend on multiple, interacting factors. Considerable experience is required to recognize the factors that warrant particular attention in a specified situation. Additional complicating factors include that individual differences in the productivity of experienced designers may be as large as 25:1 [15] and that requirement changes may necessitate rework. Inaccurate estimates of development cost and time impede the coordination of activities and allocation of resources both within and across projects. This may, ultimately, lead to

badly informed decisions about whether to continue or cancel projects. Consequently, the task of managing software projects involves that estimates are regularly checked against actual progress (Fig. 2). Estimates enforce plans by stipulating the amount of time and other resources allocated to a specified activity and must, at the same time, preserve realism by allocating enough time and resources to complete the activity. Conversely, status information enforces realism by accounting for how far the project has actually progressed and presupposes plans by assuming a shared understanding of what the outcome of specified activities should be.

Project-completion rates are low in software design [20, 36], and designers may thus be tempted to make optimistic estimates to avoid project cancellation, or they will simply direct their early efforts toward producing quick progress rather than spend their time on the planning that is necessary to make accurate estimates. DeMarco [13] finds that among software engineers an estimate is generally thought of as “the most optimistic prediction that has a non-zero probability of coming true”. This leads to frequent underestimation. With appropriate training designers become better at estimating their work and the tendency to underestimate time and size is reduced, resulting in a more evenly balanced number of overestimates and underestimates [21]. These improvements are, however, inconsequential unless used, and it appears that estimates are often supplanted by performance goals, which are used to create incentives, or deadlines dictated by market pressures or other considerations external to the design effort. This implies that a consistent move toward more accurate estimates may require profound changes at the organizational and project levels in addition to an improvement in individual designers’ ability to estimate their work [26].

Whenever a module is added or revised, ripple effects or previously undetected defects may emerge in other modules. Such changes to the status of modules are hard to predict and quantify ahead of time. In the absence of good estimation skills individual estimates may be made by increasing base estimates by a fixed percentage determined on the basis of accumulated experience. This is the approach taken by for example Microsoft, which adds 20-50% buffer time to base estimates [11]. Averaged over a number of activities such coarse-grained approaches may work well, but for individual activities designers will, at least occasionally, experience deviations that leave them idle for a period or block further progress on other activities. Organizations seem to work around these periods of waiting by assigning their designers to more than one project [33]. This, however, introduces additional dependencies that further complicate the plan-activity cycle (Fig. 2).



**Fig. 2.** Plan-activity cycle



**Browser-Interface Project.** The major means of managing the browser-interface project was two milestones. First, a working prototype should be ready for a meeting with the user representatives halfway through the project. Second, the system should be released at a fixed date. No tools or other formal means were in place to keep track of project status and support the designers in judging whether the project was on schedule. Rather, the designers relied on their personal sense of their progress and on extensive informal communication. Even formal meetings were few because the three designers were located close to each other – for part of the project they were in the same office. The designers' loose grip on status tracking was particularly evident in relation to testing. No established procedures for testing were in place and it remained, for example, largely untested whether system response times were acceptable and how platform-dependent they were. Similarly, the designers had no tools for managing their collaborative access to the source code, and there were incidents where they accidentally overwrote each other's files and thereby lost revisions. In the gradual process of setting the functionality of the browser interface the designers made explicit use of a multi-release strategy. That is, the top priority was to meet the project deadline whereas the functionality of the browser interface was considered malleable. This multi-release strategy exploited that the organization's document-management system already had an established position on the market and a base of customers that were as interested in being assured that the system grew in directions they considered relevant as in getting a specific piece of new functionality at a specific date.

**Common-Platform Project.** In the common-platform project progress toward satisfaction of requirements was not tracked systematically. Confidence in estimates gradually deteriorated and absence of shared agreement about the precise functionality of modules further eroded the basis for assessing module status. Contrary to this, an automatic mechanism was in place to track status at the code level and make updated versions of the code available to the designers. In total, the modules of the common-platform project comprise more than a million lines of code. The size of the code and the number of designers involved created a need for regularly establishing the code-level status of the modules and checking cross-module compatibility. This was achieved by a nightly build; that is, every night the latest version of each module was automatically compiled and linked with all the other modules. Whenever the nightly build succeeded the designers had a running version of their system. If a module contained errors that prevented its compilation or linking, it was automatically added to an intranet page listing the modules that failed the build, and an auto-generated email was sent to the designer responsible for the module. Thus, when the designers arrived at work in the morning they had access to a version of the code that included all designers' work up until yesterday evening and they had a complete list of the modules that failed the build. The nightly builds promoted a work practice in which people made an effort to check the correctness of their module before they went home. Further, some tests were run automatically every night with standard data sets and checks of system output against reference data. Finally, in-code comments were extracted from the code during the nightly build and a set of intranet pages generated. These web pages contained documentation of individual functions but rarely

covered interactions among functions or issues above the function level. Thus, while this documentation was regenerated every day it was insufficient as a means of making sense of the code. However, little design documentation exists apart from these web pages. The main reason for this is that the project group was under an unrelenting pressure to produce progress, and to be perceived as productive a designer had to be writing source code, not documentation. For similar reasons the status information resulting from the nightly builds was not accompanied by careful estimation and re-estimation of activities.

**Reasons for Observed Practices.** Reluctance or failure to make estimates and track status is widespread in software design. Common reasons for this are schedule pressure, fluid requirements, and limited experience with estimation [e.g., 4, 13, 25]. In the browser-interface and common-platform projects prominent reasons for the absence of systematic estimation and status assessment were:

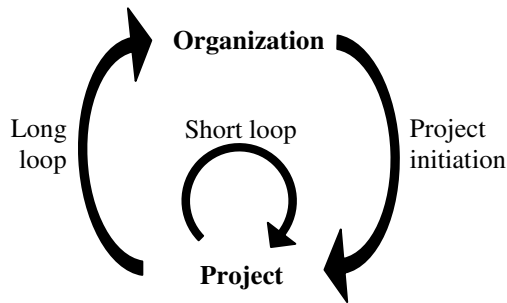
- *Accurate estimates presuppose detailed requirements.* In the absence of clear requirements it is futile to attempt to estimate the time and resources required to complete a system or module. Rather, the designers in the browser-interface project reversed the process and used deadlines, which were stated more clearly than requirements, as a pragmatic basis for ‘estimating’ the functionality they would be able to deliver.
- *Not knowing how to handle estimates that are not met.* The designers in the common-platform project gradually lost confidence in estimation when they realized that they repeatedly failed to meet their estimates. Merely replacing old estimates with new made the whole effort seem pointless to them. Uncertainty and disagreements about the precise functionality of the modules further reduced their confidence in the estimates. Eventually, they largely abandoned estimation but kept tracking status.
- *Estimates are confronting for the individual designer.* Estimates create transparency with respect to whether the individual designer delivers on time or introduce delays that may have ripple effects on his or her colleagues’ work. Thus, while estimates are central to the management of collaborative work, an immediate consequence for individual designers is increased exposure of delays and thereby a risk of being perceived as a less competent professional.

The nightly builds in the common-platform project illustrate that keeping track of project status at the code level and at the requirements level are distinct issues. Abstaining from working systematically with requirements means that decisions about requirements are made by individual designers and may subsequently be contested by other designers and by users. This provides a fragile basis for making progress and assessing project status.

### 3.3 The Collaboration Challenge

The collaboration challenge is about learning within and across projects. After discussing this constituent of software design it is illustrated with data from the two empirical studies.

**Learning within and across Projects.** In general, no single designer possesses all the required project knowledge in the necessary detail. Thus, to accommodate the customers' needs as well as needs arising from stakeholders such as marketing, service, maintenance, and quality control, software design becomes a collaborative effort. Another reason for developing software collaboratively is that many activities can then proceed in parallel and thereby both reduce the time from a decision is made to its consequences become apparent and shorten total development time. However, the distribution of software design onto multiple individuals creates a need for communication and coordination, which increases drastically with the size of the collaborating group [5]. Communication and coordination take place both within and across projects, corresponding to a short and a long learning loop (Fig. 3).



**Fig. 3.** Short and long learning loops

The project knowledge held by a group of designers is constantly evolving and in this sense learning is an integral part of their work practice [6]. This learning-in-working is local, aimed at competent performance, and woven into a collaborative practice. First, it is local in that it consists of gaining a coherent understanding of issues pertaining to the project at hand. These project issues are rich in contextual detail specific to the concrete situation, and these specific details are of paramount importance to the successful completion of projects. Second, it is aimed at competent performance because the ability to produce useful and usable systems in a well-managed way is much more salient to designers than production of generalized, explicit knowledge. According to Allen [1] this is the distinctive difference between engineering work and the work of scientists. Third, it is woven into a collaborative practice in that the different experiences and competencies contributed by different project participants provide learning opportunities beyond those available to people working individually. These learning opportunities enable designers to replace project activities involving prohibitive amounts of individual experimentation with close collaboration among people with relevant prior experiences.

Within projects written communication can be minimal if the designers meet often. Design methods often prescribe that a number of design artefacts are produced and kept up to date, but actual use of the methods tends to be more opportunistic [2, 22]. Design artefacts tend to be used at selected points in projects when designers perceive that the artefacts may have a direct impact on the progress of their project. During the in-between periods where the design artefacts are not contributing directly to the

designers' current activities the refinement and maintenance of the artefacts is likely to be postponed or downgraded in favour of activities that yield more immediate gains. Instead, designers carry most project information in their heads [34, 38]. This increases the reliance on oral communication and the centrality of the few people on a project who are able to reason and argue about how local changes affect the overall design. Over the course of a project these key people extend and refine their knowledge of the project by repeatedly debating alternatives, resolving disagreements, and incorporating redirections. Sharing this knowledge within the project group is an important but time-consuming process [3], and other project activities are likely to be competing for the key people's time, including activities that may appear more important because they break new ground and thereby yield pertinent project progress.

Across projects the experiences gained and solutions devised by designers may remain untapped by their colleagues because they are unaware of them or uncertain about their applicability outside their original context. The long loop represents this crucial but often unmanaged flow of experiences, solutions, and other knowledge from individual projects back to the organization for reuse in other projects. Zedtwitz [37] reports that 80% of projects are not reviewed after completion or cancellation to systematically and regularly make acquired project knowledge available for organizational learning. Further, in the design documentation made during projects designers are likely to make extensive use of condensed writing, which leaves most of the context unsaid because the documentation will be understood by its primary readers – usually other project members – as belonging to a certain ongoing activity. To make documents understandable to people who are not familiar with the context the condensed forms of writing must be elaborated, often to the exasperation of the primary readers who can see the elaboration as redundant [7]. Also, the pressure to produce project progress often precludes that designers spend time expanding their writings into documents understandable to unknown future readers [20]. Instead, most of the information that flows from project to project is carried by people, and oral communication and project staffing become key elements in the cross-project management of knowledge. This has spurred increasing interest in systems directed at locating knowledgeable colleagues – people-finding systems [e.g., 29].

**Browser-Interface Project.** The initial browser-interface prototype, and the analysis leading up to it, was made by a free-lance consultant who was not otherwise involved in the project. Thereby the three designers on the project missed the opportunity to learn from the consultant's experiences, apart from what they could deduce from the prototype. Instead, the three designers started largely afresh and relied on oral communication in keeping each other informed about their work. Written design documentation was sparse and played a negligible role. One of the interviewees estimated that a total of 20-25 pages of documentation were produced, all at the very end of the project. Apart from the small size of the project the interviewees emphasized three core success factors, all of which concerning the distribution of and easy access to project-relevant knowledge. First, the physical proximity of the three designers made it quick and easy to ask for help, and supported them in maintaining a mutual awareness of each other's current activities. Second, the three of them were responsible for the entire project. The absence of third parties enabled a way of working in which a shared understanding of the evolving design was constructed and maintained orally

through numerous conversations in their shared office. Third, the project was assigned one of the organization's most competent designers. The interviewed project manager stressed the importance of the few especially competent people and had made it a precondition for accepting to become the project manager that one of these core people was assigned to the project. Along with informal communication, staffing appeared to be the major way in which experience was transferred from project to project. In most cases staffing also determined the possibilities for reuse of software components because sparse documentation limited reuse to components the individual designers had themselves been involved in developing. The only occasion on which the browser-interface project has been evaluated and the lessons learned from it discussed was at an informal, project-internal meeting shortly after the project deadline.

**Common-Platform Project.** In the common-platform project the interviewees expressed a need for better ways of managing how far they had progressed toward completion. On the one hand, the project manager was not sufficiently good at defining and enforcing project milestones. On the other hand, the designers were not sufficiently good at communicating the actual status of their modules – many modules were “almost completed” for extended periods of time. The interviewees found that this boiled down to (1) frequent opacity or disagreements as to the functionality required from a module for it to be complete and (2) inadequate estimation skills. The first issue is a combination of communication breakdowns and imprecision in the analysis that turned overall project goals into specific requirements. This analysis was largely left to the individual designer, and no artefacts or stipulated procedures were in place to support the designers in communicating, arguing about, and reaching closure on the outcome of these analyses. A core element of the second issue is that writing source code was perceived as the primary activity whereas the time required for activities such as testing and documenting the code was generally underestimated. For the people appointed to system testing this activity was a secondary activity and their primary task consumed the majority of their time. Thus, testing was patchy and errors were encountered and corrected in a piecemeal fashion. The project did not include a post-project evaluation, and the organization has no cross-project forum for communicating lessons learned in one project to the rest of the organization. That is, the experiences gained in the project have not been the subject of collaborative discussion, apart from informal exchanges among designers. Thus, as an example, the nightly build and its associated mechanisms for supporting the development work were invented and instituted within the common-platform project by a single person, who has subsequently left the organization.

**Reasons for Observed Practices.** Projects are ubiquitous in software design, indicating that organized collaboration is biased toward the short loop whereas collaboration across projects tends to be informal [35, 37]. This is clearly illustrated by the browser-interface and common-platform projects. Apart from general cognitive and motivational factors [e.g., 24] reasons for having few artefacts and forums in place in support of the long loop include:

- *Short-term costs overshadow long-term gains.* Extra work is required to make project knowledge available to colleagues on other projects, and the reuse benefits of such work are hard to assess and more distant than the immediate tasks competing

for designers' time and attention. In small projects the extra work may be prohibitive and in highly dynamic settings reuse may seldom happen. However, the members of the browser-interface and common-platform projects felt that they ought to invest more in the long loop.

- *Project knowledge is context sensitive.* Designers interact repeatedly with their colleagues to get information, trusted opinion, and impetus for creative discourse. In these interactions, colleagues are not simply sources of information but actively involved in interpreting the applicability of their knowledge to the concrete situation. Conversely, designers are reluctant to engage in project post mortems and other activities that evolve around the context in which knowledge was gained because they are uncertain whether it will be applicable to future projects.
- *Not knowing how to make the long loop more effective.* A need for process support has been noted in relation to the two other constituents of software design but it is even more apparent in relation to the long loop. With the exception of documentation, the designers on the browser-interface and common-platform projects lacked knowledge of and experience with means of collaboratively managing the flow of knowledge across projects.

The collaboration challenge – especially the long loop – is the constituent of which the designers on the browser-interface and common-platform projects were least aware. At the same time, methods for managing the long loop appear to be less developed than for the short loop [24], though activities such as learning are crucially important to successful completion of software projects.

## 4 Implications for User-Centred Design

Based on the analysis of the three constituents of software-project complexity, this section aims to identify and discuss selected challenges to organizations' successful use and continued elaboration of practices for user-centred design.

### 4.1 Collaborative Grounding

In both empirical studies many of the troubles experienced by the designers concern collaborative grounding; that is, the active construction by actors of a shared understanding that assimilates and reflects available information. Project activities are rarely performed by the entire group of designers but typically by varying subgroups of the involved designers. Deliberate efforts of collaborative grounding are required to extend the knowledge acquired by a subgroup to the remaining designers on a project. The designers in the two empirical projects often under-recognized this need for collaborative grounding. Collaborative grounding is central to contextual design [3] and some participatory-design techniques [e.g., 19] but most techniques for user-centred design are biased toward information-seeking activities to the extent of largely bypassing collaborative grounding. For example, most usability evaluation methods focus on problem identification and largely evade the subsequent grounding of the evaluation results in the entire project group. This amounts to assuming that a project group is one unitary actor, rather than a network of actors that need to actively construct a shared understanding. The two studied projects vividly illustrate that the

designers struggled with collaborative grounding in relation to all three constituents of software design. Examples include that a shared understanding of module functionality was a long time in the making, that estimates were consequently inaccurate and difficult to interpret, and that no forums for long-loop learning were in place to prevent these issues from recurring in the next project.

## 4.2 Long-Loop Learning

Small project groups with around five members are widespread in software design, and many organizations actively opt for small project groups, for example by dividing development tasks onto multiple projects [8]. The browser-interface project is a case in point. In such small groups the communication and collaborative grounding necessary to cope with the short loop is manageable. Conversely, the common-platform project was staffed with 10-15 people, and this alone made it much more demanding to cope with the short loop. However, the size of a project group is also a means to shift the balance between the short loop and the long loop. A small project group needs frequent communication with project-external sources to exploit lessons learned in other projects. A larger project group will have access to more of these lessons by means of communication among project members and the long loop will, thereby, be partly subsumed in the short loop. Apart from project staffing, the organizations in both empirical studies relied on informal exchanges among designers as the principal means of exploiting experience from one project in other projects. Given the frequent recommendations of small projects [8, 11] and the ensuing reliance on an effective long loop it is noteworthy that methods for user-centred design focus almost exclusively on individual projects. Thus, methods as well as practitioners appear to devote most of their attention to the short loop and in so doing they render the long loop comparatively invisible. In both empirical projects the designers seemed to devote little time and attention to collaborative activities directed at improving their practices from one project to the next. Concrete guidance is needed on how to work effectively with the long loop in relation to user-centred design. Activities involving a more systematic pull of information, practices, and other resources into projects are probably more likely to become successful than activities aimed at pushing information and so forth from ongoing toward future projects.

## 4.3 Intimidation Barriers and Project Knowledge

The small to medium size of the projects and organizations in the two empirical studies could be an important factor in understanding their practices. The size may create an intimidation barrier toward software-process and long-loop initiatives that introduce (1) a new mindset promoting the longer-term effects of present practices rather than their more visible, immediate effects, (2) more systematic and regulated work processes, and (3) methods that are generally associated with large projects and organizations. The two empirical studies point toward a need for lightweight techniques and practices for managing the complexities inherent in the three constituents of software design. Discount usability engineering [31] suggests that unthreatening starting points and modest steps may be important to the adoption of such techniques and practices. However, practitioners also need to realize that as the systems they engage in

designing grow increasingly complex so does their need for techniques and practices that can match this complexity. A more managed process appears necessary. For user-centred design this seems to point toward further work on reaching closure on a design, integrating the task-artefact and plan-activity cycles, and communicating experiences across projects. Improved practices and a more managed process should, however, not be achieved by starting to consider methods an alternative to the project knowledge created by designers in response to the particularities of their current project.

## 5 Conclusion

Software design is a complex undertaking as evidenced by the frequency with which projects are cancelled, late, over budget, or resulting in marginal gains and systems disliked by users. Three major constituents of software-project complexity have been analysed in this study: the formative element, the progress imperative, and the collaboration challenge. Empirical data from two small to medium-size projects illustrate that practitioners struggle to manage these constituents. While each of the empirical studies is based on only two informants, the studies provide patent illustrations of a gap between the state of affairs in these software projects and the state of the art regarding software-process management. The designers in the two studied projects had few techniques and other means in place to support their work. Instead, they relied on an informal approach in which requirements, estimates, status information, and other design information were largely kept in the designers' heads and exchanged with close-by colleagues on an ad-hoc basis. The exceptions to this informal approach were carefully selected and mainly consisted of the nightly builds in the larger of the two projects and the annual customer seminar hosted by the organization in which the other project took place.

In many organizations, the principal means of coping with the long loop is project staffing. This reflects that project knowledge often unfolds around a few people with knowledge of relevant prior projects and the ability to take in the various pieces of information involved in a design, make out how they hang together, and articulate this clearly. A main challenge for user-centred design is to provide support for a more managed design process while avoiding that methods become seen as an alternative to project knowledge.

**Acknowledgements.** Johannes Knigge contributed to the empirical studies. Special thanks are due to the interviewees who agreed to participate in this study in spite of their busy schedules.

## References

1. Allen, T.J.: Distinguishing engineers from scientists. In: Katz, R. (ed.) *Managing Professionals in Innovative Organizations: A Collection of Readings*, Ballinger, Cambridge, MA, pp. 3–18 (1988)
2. Bansler, J.P., Bødker, K.: A reappraisal of structured analysis: design in an organizational context. *ACM Transactions on Information Systems* 11(2), 165–193 (1993)
3. Beyer, H., Holtzblatt, K.: *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann, San Francisco (1998)



4. Boehm, B.W.: Software risk management: principles and practices. *IEEE Software* 8(1), 32–41 (1991)
5. Brooks, F.P.: *The Mythical Man-Month: Essays on Software Engineering*, Anniversary edn. Addison-Wesley, Reading (1995)
6. Brown, J.S., Duguid, P.: Organizational learning and communities-of-practice: toward a unified view of working, learning, and innovation. *Organization Science* 2(1), 40–57 (1991)
7. Brown, J.S., Duguid, P.: The social life of documents. *First Monday* 1, 1 (1996), <http://firstmonday.org/issues/issue1/documents/index.html>
8. Carmel, E., Bird, B.J.: Small is beautiful: a study of packaged software development teams. *Journal of High Technology Management Research* 8(1), 129–148 (1997)
9. Carroll, J.M., Kellogg, W.A., Rosson, M.B.: The task-artifact cycle. In: Carroll, J.M. (ed.) *Designing Interaction: Psychology at the Human-Computer Interface*, pp. 74–102. Cambridge University Press, Cambridge (1991)
10. Curtis, B., Krasner, H., Iscoe, N.: A field study of the software design process for large systems. *Communications of the ACM* 31(11), 1268–1287 (1988)
11. Cusumano, M.A., Selby, R.W.: How Microsoft builds software. *Communications of the ACM* 40(6), 53–61 (1997)
12. Danish Board of Technology: *Erfaringer fra statslige IT-projekter – hvordan gør man det bedre?* Report No. 10, Copenhagen, DK (2001)
13. DeMarco, T.: *Controlling Software Projects: Management, Measurement and Estimation*. Yourdon Press, Englewood Cliffs (1982)
14. Eason, K.: *Information Technology and Organisational Change*. Taylor & Francis, London (1988)
15. Egan, D.E.: Individual differences in human-computer interaction. In: Helander, M. (ed.) *Handbook of Human-Computer Interaction*, pp. 543–568. Elsevier, Amsterdam (1988)
16. Eodice, M.T., Fruchter, R., Leifer, L.J.: Towards a theory of engineering requirements definition. In: Lindemann, B., Meerkamm, V. (eds.) *Proceedings of ICED 1999*, vol. III, pp. 1541–1546. Technische Universität München, Garching, DE (1999)
17. Fayad, M.E., Laitinen, M., Ward, R.P.: Software engineering in the small. *Communications of the ACM* 43(3), 115–118 (2000)
18. Gould, J.D., Lewis, C.: Designing for usability: key principles and what designers think. *Communications of the ACM* 28(1), 300–311 (1985)
19. Greenbaum, J., Kyng, M. (eds.): *Design at Work: Cooperative Design of Computer Systems*. Erlbaum, Hillsdale (1991)
20. Grudin, J.: Evaluating opportunities for design capture. In: Moran, T.P., Carroll, J.M. (eds.) *Design Rationale: Concepts, Techniques, and Use*, pp. 453–470. Erlbaum, Mahwah (1996)
21. Hayes, W., Over, J.W.: *The Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers*. Technical Report No. CMU/SEI-97-TR-001. Carnegie Mellon University, Pittsburgh, PA (1997)
22. Hertzum, M.: Making use of scenarios: a field study of conceptual design. *International Journal of Human-Computer Studies* 58(2), 215–239 (2003)
23. Hertzum, M.: Small-scale classification schemes: a field study of requirements engineering. *Computer Supported Cooperative Work* 13(1), 35–61 (2004)
24. Hinds, P.J., Pfeffer, J.: Why organizations don't know what they know: cognitive and motivational factors affecting the transfer of expertise. In: Ackerman, M.S., Pipek, V., Wulf, V. (eds.) *Sharing Expertise: Beyond Knowledge Management*, pp. 3–26. MIT Press, Cambridge, MA (2003)

25. Humphrey, W.S.: Why don't they practice what we preach? *Annals of Software Engineering* 6, 201–222 (1998)
26. Humphrey, W.S.: Three process perspectives: organizations, teams, and people. *Annals of Software Engineering* 14, 39–72 (2002)
27. Kensing, F., Munk-Madsen, A.: PD: structure in the toolbox. *Communications of the ACM* 36(6), 78–85 (1993)
28. Landauer, T.K.: *The Trouble with Computers: Usefulness, Usability and Productivity*. MIT Press, Cambridge, MA (1995)
29. Mockus, A., Herbsleb, J.D.: Expertise browser: a quantitative approach to identifying expertise. In: *Proceedings of ICSE 2002*, pp. 503–512. ACM Press, New York (2002)
30. Naur, P.: The place of programming in a world of problems, tools, and people. In: Kalenich, W. (ed.) *Proceedings of IFIP Congress 65*, Spartan Books, Washington, DC, pp. 195–199 (1965)
31. Nielsen, J.: *Usability Engineering*. Academic Press, San Diego (1993)
32. Norman, D.A.: Cognitive engineering. In: Norman, D.A., Draper, S.W. (eds.) *User Centered System Design: New Perspectives on Human-Computer Interaction*, pp. 31–61. Erlbaum, Hillsdale (1986)
33. Perry, D.E., Staudenmayer, N.A., Votta, L.G.: People, organizations, and process improvement. *IEEE Software* 11(4), 36–45 (1994)
34. Potts, C., Catledge, L.: Collaborative conceptual design: a large software project case study. *Computer Supported Cooperative Work* 5(4), 415–445 (1996)
35. Schindler, M., Eppler, M.J.: Harvesting project knowledge: a review of project learning methods and success factors. *International Journal of Project Management* 21(3), 219–228 (2003)
36. Schmidt, R., Lyytinen, K., Keil, M., Cule, P.: Identifying software project risks: an international Delphi study. *Journal of Management Information Systems* 17(4), 5–36 (2001)
37. von Zedtwitz, M.: Organizational learning through post-project reviews in R&D. *R&D Management* 32(3), 255–268 (2002)
38. Walz, D.B., Elam, J.J., Curtis, B.: Inside a software design team: knowledge acquisition, sharing, and integration. *Communications of the ACM* 36(10), 63–77 (1993)

## Questions

### *Jan Gulliksen:*

*Question: This kind of work usually focuses on projects that have failed. Did you try to find successful projects and see how they work? Or find out whether changing practices would make projects more successful?*

Answer: We didn't select our projects for success or failure. Others have looked at success. Also looking at projects that have used user-centred methods will tell us something more.

### *Annelise Mark Pejtersen:*

*Question: Can you make such a sharp distinction between successful and unsuccessful projects?*

Answer: I agree. If you ask different people they will also have different views about the project. Some people focus on process, and others on product.

# Applying Graph Theory to Interaction Design

Harold Thimbleby<sup>1</sup> and Jeremy Gow<sup>2</sup>

<sup>1</sup> University of Swansea  
h.thimbleby@swansea.ac.uk

<sup>2</sup> University College London  
j.gow@ucl.ac.uk

**Abstract.** Graph theory provides a substantial resource for a diverse range of quantitative and qualitative usability measures that can be used for evaluating recovery from error, informing design tradeoffs, probing topics for user training, and so on.

Graph theory is a straight-forward, practical and flexible way to implement *real* interactive systems. Hence, graph theory complements other approaches to formal HCI, such as theorem proving and model checking, which have a less direct relation to interaction.

This paper gives concrete examples based on the analysis of a real non-trivial interactive device, a medical syringe pump, itself modelled as a graph. New ideas to HCI (such as small world graphs) are introduced, which may stimulate further research.

## 1 Introduction

A fundamental idea in HCI is that users build mental models of the devices they interact with. Often one can do useful work with quite vague notions of mental and device model, but low-level device features have high-level cognitive effects [11]. For rigorous HCI work, and particularly with safety critical devices and tasks, then, it is essential to have a very clear notion of what the device model is. Unfortunately much work in design, specification and verification of interactive systems uses abstract or incomplete models of devices. What is needed is an approach that can represent full, concrete devices and which has value for analysis of interaction.

If we restrict ourselves to devices that are implemented by computer programs, then the programs (in their given languages) are the final arbiters of the device models. Unfortunately, typical programs do not lend themselves to defining *clear* device models. Programs (and their specifications) are for instructing computers, not for defining user interface behaviour, which in fact happens as a side-effect of running them. Hardly any code in a typical program has anything explicitly to do with the behaviour of the user interface, and typically the code for the user interface is widely distributed throughout the program: there is no single place where interaction is defined.

Graphs are a mathematical concept that lend themselves to analysis and interpretation by program. A large class of interactive system can be built concisely

from graphs—and it is a trivial theorem that any digital computer system is isomorphic to a graph and a simple state variable. Significantly, as this paper shows, graphs lend themselves very well to a wide variety of analysis highly relevant to HCI concerns. For example:

- Sequences of user actions are *paths* in a graph. A standard graph theoretic concept is the shortest path between two vertices, which defines the most efficient way a user can achieve a particular change of state. If there is no such path, then a user cannot achieve the state change.
- The transition matrix  $M$  of a graph gives the number of ways a user can cause a state transition by doing exactly one action. The matrix  $M^n$  is the number of ways of achieving any state transition with exactly  $n$  actions; and  $\sum_{i=1}^k M^i$  is the number of ways of achieving any transition with  $1, 2, 3 \dots k$  actions. The higher the number of ways of achieving a state transition, the easier the state is for the user to reach. A safe (a secure interactive device) would typically have only 0 and 1 entries in  $\sum M^i$ , whereas a permissive device [15] would have comparatively large entries.

In short, graphs very readily *simultaneously* define interactive systems and usability properties. Graph theory connects formal specification, runnable programs (or prototypes) and HCI. This paper backs up this claim with a wide-ranging analysis of a working simulation of a real, non-trivial interactive device.

## 1.1 Graph-Based Approaches

Although the use of transition systems to specify interactive systems was proposed as early as 1960 [10], they did not catch on as a ‘pure’ formalism because of their apparent limitations for user interface management systems (UIMS)—leading to a line of research [20, etc] that was overtaken by modern rapid application development (RAD) environments [9]. However, the drive behind both UIMS and RAD environments was programmability and flexibility rather than rigor. In rigorous HCI, one needs a programming framework that is both analytic and close to the user interface, if not identical with it: graphs achieve this goal. Graph theory was proposed for use in HCI in [13,14] as a means of analysis; other work includes using graph theory for providing interactive intelligent help [18], and using flowgraph concepts to analyse user manuals as structured programs [17].

Graph theory is a substantial area of mathematics, and many interesting theorems and properties are known for graphs that can readily be programmed on a computer (see, e.g., [2,7,12]). A graph is readily represented by drawing vertices as dots, and arcs as arrows joining dots. Vertex and arc labels are written as words adjacent to the vertices and arcs. If vertices are drawn as circles or other shapes, their labels can be written inside the shapes. Small graphs are easy to draw by hand and larger graphs can be drawn automatically using appropriate tools [3]. To avoid clutter labels are sometimes omitted. Reflexive arcs (also called trivial arcs) that point back to the same vertex are also often omitted for clarity.

## 2 Graphs and Interactive Systems

We use labeled directed multigraphs in this paper, but what is a graph and how does it relate to an interactive device?

A labeled directed multigraph is a set of objects called vertices  $V$ , a collection of arcs  $A \subseteq V \times V$  which are ordered pairs of vertices, and two total functions  $\ell_V: V \rightarrow L_V$  and  $\ell_A: A \rightarrow L_A$  that map vertices, respectively arcs, to sets of labels, which name the vertices and arcs.

The graph theoretic terms are vertices and arcs, but the device or programming terminology usually refers to vertices as states and arcs as transitions; the user terminology refers to arcs as actions. Formally there is no difference. However, for most devices, the user cannot uniquely identify the state of the device. Instead, the user can observe (hear or feel) indicators. We model this as a mapping  $O$  from vertices to the powerset of available indicators  $I$ ,  $O: V \rightarrow \mathbb{P}I$ . That is, in a given state  $s$ ,  $O(s)$  is the set of indicators that are ‘shown’ to the user.

An interactive device can be represented straight forwardly as a directed graph assuming: user actions are mapped into arcs, states are mapped into effects the user can observe (for instance with sounds or indicator lights) and the device must track the current state using a variable. When the user performs an action, the current state  $A$  is changed to the next state  $B$  where there is a directed arc from  $A$  to  $B$  labeled with that action. Arcs may point back to the same state, and the transition then does not change the state; if the next state is  $A$  we say that the action is *guarded* in  $A$  as no non-trivial transition occurs.

Graph models may be non-deterministic—either because of the underlying system or because of constraints on the modeling process—in which case one of several possible next states will be arrived at. Although useful, non-determinism complicates many of our graph metrics, and is beyond the scope of the current paper.

Graph models can be extended with other concrete representational details to relate them to actual interactive systems. For example an image can act as a device’s *skin*, e.g., as used with the Java model shown in figure 1. Changes to the skin during use can be captured by *indicator skins*—changes to the skin which correspond to the activation of individual indicators. Although an important practical consideration, skins make little impact on our approach.

To be formal, devices are considered finite state automata represented by a 10-tuple  $\langle V, L_V, \ell_V, A, L_A, \ell_A, O, I, s_0, S, I_S, i_S \rangle$ , with (in addition to the components already introduced above)  $s_0$  the initial state (the state a device is in before it is first used),  $S$  the skin (which for our purposes is a colour image), and  $I_S$  a bijection from vertices to indicator skins  $i_S$ . This level of formality may look pedantic, but there is an important point: *precisely* this information is sufficient to build a functioning interactive simulation (and even a user manual) and to analyse its usability and other properties in depth. The fruitfulness of this approach is explored throughout this paper.

In what follows, we will use the terms *state* and *vertex* interchangeably, but stylistically we use state for user-related issues and vertex for graph theoretic issues. Similarly, we will use *action*, *press*, etc, for user actions, but *arc* for the corresponding graph concept. Typographically, we shall write **State** and Action.



**Fig. 1.** Partial screen shot of the simulation—a user can mouse click on the buttons, which are animated to give simple visual feedback of pressing. Note that graph theory does not address all HCI issues, such as the naming or confusibility of buttons.

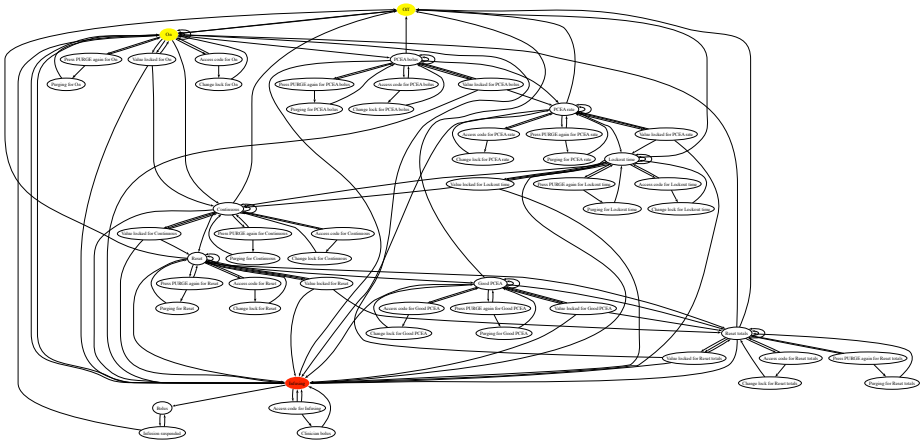
## 2.1 Case Study

A syringe is used to give patients injections of drugs. A syringe pump is an automatic device that uses a motor to drive the syringe, and gives a patient an injection usually over a period of hours or even days. The pump is set up by a nurse or anaesthetist to deliver drugs for various conditions: for example, so that it can be used on demand by a patient for pain management. Some pumps have detailed models of drug uptake in the patient (the patient weight having been entered), and may be used for anaesthesia. An ambulatory pump is one that a patient can wear or carry around, and is typically used for pain management by delivering calibrated dosages of drug on demand—within parameters set up by the nurse, particularly so that the patient cannot overdose.

This paper uses as a running example a simulation of the main features of the Graseby ambulatory syringe pump type 9500 [5]. The simulation of the Graseby pump has been implemented as a Java program, constructed explicitly from a graph model (of 54 vertices and 157 non-trivial arcs)—it is an example of a realistic-scale, safety critical interactive system, and thanks to its graph-based definition, with a formal specification that corresponds *directly* to its interaction behaviour. See figure 1 for a screen-shot of the Graseby simulation, and figure 2 for a representation of its graph.

For reasons of space, we only use this one example system; in general a designer would have a collection of systems and compare properties for variations of the basic design. Clearly a very important practical use of graph theory is to compare designs, particularly a design and iterative variations of it. For reasons of space, we make no design comparisons here.

The remainder of the paper discusses some of the user issues that can investigated using graph theoretic properties—some of them standard, others of special interest to HCI, and some of the potentially opening up new research areas within HCI.



**Fig. 2.** Illustrative visualisation, drawn by Dot concurrently with a running simulation. Each state has a textual description shown in the diagram, but reproduction at the scale necessary for these proceedings may have made the descriptions illegible; although the reduced diagram here is not particularly readable, the graph visualisation program allows the diagram to be zoomed and scrolled, so very large graphs can be handled conveniently. In our system, previously visited states are shown in yellow, and the current state is in red (though monochrome reproduction of this paper will may make all such states look grey).

### 3 Navigation

First, we look at graph metrics related to the user’s ability to navigate the device’s state space.

#### 3.1 Reachability

A graph is *strongly connected* if there is a directed path connecting each pair of vertices; in other words, the user can get from any state to any other state. There are no dead-ends, and no unreachable states. The Graseby is indeed strongly connected.

For many real devices, a weaker property is important: every state can be reached from a certain set of states  $S$ , typically including a standby or off state. For example, it is important that a fire extinguisher can be used from **Standby**, but once used it cannot be returned to **Standby** by the user—it needs recharging. This property can be expressed in many ways, for example for every vertex  $v \in S$  there is a spanning tree rooted at  $v$ . An example from desktop PCs is that one wants to be able to write any document starting from a new, empty document.

If a graph is not strongly connected, it will have at least two strongly connected components. If each strongly connected component is contracted to a single vertex, the resulting graph must be acyclic (in fact a DAG). A designer may use

this concept in three ways: first, to check that all states are reachable (otherwise the device has features that cannot be used); secondly, to determine the set of states that can reach selected states.

All connectivity properties can be conveniently determined from the all-pairs shortest paths matrix,  $P$ , readily found by Dijkstra's algorithm. If there is a path from  $u$  to  $v$ , then  $P_{uv}$  will be finite, and moreover  $P_{uv}$  is the minimum number of user actions to perform the appropriate state transition. A graph is strongly connected if and only if all elements of  $P$  are finite. The *characteristic path length*, a property we use below (see section 3.3), is the average of elements in  $P$ .

### 3.2 Diameter and Radius

The *diameter* and *radius* of a graph are defined in terms of eccentricities. The eccentricity of a vertex is the distance to the furthest vertex from it; more precisely it is the longest shortest path between it and all vertices. The diameter of a graph is then its greatest eccentricity, and the radius is its least eccentricity. In usability terms, the diameter represents the difficulty, counted in actions, to the user of the worst task (or tasks) they can do on the device. The radius is the difficulty of the 'easiest hardest' thing to do. Of course, 'difficulty' is a formal term; in fact, users will make mistakes, or not know the best way of achieving their tasks—the eccentricity represents an optimal, error-free, fully knowledgeable user, and thus a *lower bound* on difficulty. However, it is not difficult for graph measures to be conventional usability metrics, such as time; for example, the Fitts law can estimate the time for the user to execute all actions along any path.

The diameter and radius can be used to define two interesting sets of states, based on eccentricity. The *centre* of a graph is the set of vertices with eccentricity equal to the radius; whereas the *periphery* is the set of vertices with eccentricity equal to the diameter.

The diameter of the Graseby graph is 8 and its radius is 5. The centre of the Graseby is the single state **On**. This state is reached from **Off** by pressing the **On** button; in other words, as soon as the Graseby is switched on, it is in the (as it happens, unique) state where everything is as easy as it can be.

The Graseby has a periphery of 15 states, 8 of which are concerned with patient controlled analgesia (PCA). Arguably, the patient features of the device should be simpler in some sense than the nurse or anaesthetist features; the analysis highlights this potential design concern. On the other hand, the Graseby has several modes—it can be unlocked, half locked or fully locked—that restrict to varying degrees what a patient can do. It would be possible to work out the periphery under each lock condition, but we will not do so here (as we are illustrating the use of the graph theory techniques for usability analysis, not evaluating the device).

### 3.3 Small World Graphs

A *small world* graph is one that has an unusually small average shortest path between all pairs of vertices. The classic small world example is the social graph



of relationships: ‘six degrees’ is the (popular) mean least number of familiar relations between *any* two people. Whether the number is exactly 6 or not, for a graph with as many vertices as people and as sparsely connected, it is remarkable that this *characteristic path length* (the mean shortest path length) is so low.

Small world metrics are relevant to HCI because a device may have a huge number of states, but it should still have a modest expected cost of getting from any state to any other. In other words, a small world device is usable—and easier to use than an equal sized non-small world graph. There are many small world metrics, all of which are easy to measure. Thus the characteristic path length of the Graseby is 4.1, indicating a relatively small expected cost for navigating the device. We discuss more benefits of small world graphs in sections 4.5 and 5.2.

### 3.4 Completeness

A complete graph has an arc connecting each pair of vertices; it is possible for a user to get from any state to any other state in a single action. There must be at least  $N - 1$  user actions for an  $N$  state device. In particular, if there are at least  $N$  actions, they may be conveniently labeled with the name of the target state.

The complete graph  $K_n$  of  $n$  vertices is unique up to isomorphism. The complete graph  $K_2$  is familiar as the on/off graph, and indeed the states are usually called **On** and **Off**, and the action labels can be unambiguously called **On** and **Off**.

A designer may wish to check the property of *directness*, namely that every arc label  $\ell_A(uv)$  satisfies the property  $\ell_A(uv) \Rightarrow \ell_V(v)$ , with  $\Rightarrow$  appropriately defined to correspond to ‘perceptual’ or ‘cognitive’ implication. For example, in the on/off device described above, if the user does **On**, they might expect the device will enter the state **On**; or put formally, **On**  $\Rightarrow \ell_V(\mathbf{On})$ . Of course by design we should have **On** =  $\ell_A(\mathbf{Off On})$ , as well as  $\Rightarrow \ell_V(\mathbf{On})$ .

In general, directness will make a device easy to use but it implies the device has enough distinct actions, and for a complex device the designer will have to choose which actions are direct and which indirect. For many devices, however complex, **Off** is typically a direct action. On the other hand, directness permits a device to have more action labels than states, for instance to provide alternative ways to get to a state. A designer would probably require, further, that for every arc label there is an appropriately labeled out-arc from every vertex—otherwise some actions will not work in some states.

The advantage of a complete graph is that anything the user might want to do can be done in exactly one action; conversely, there is a problem: the user cannot be guarded from any side-effects, nor can there be any security as no states can specifically guard any others. Furthermore, since there are at least as many actions as states, the number of states may be limited for physical reasons: on a push button interface, 100 states would require at least 100 buttons which may be impractical simply in terms of space. A more interesting design issue for a direct complete graph is that in every state there is one button that does nothing—though the user can *always* press a button **X** to achieve state **X** regardless of whether the device is in state **X** already.

Most devices are not complete, however. In this case, we can automatically identify complete subgraphs, and then test the subgraphs for the appropriate properties.

## 4 Errors

Graph theory lends itself to analysing the nature and costs of various error scenarios a designer may be interested in.

### 4.1 Undo Cost

The *undo cost* of a device can be defined as the average cost of recovering from a single action error. If a user presses a button by mistake, on average, what is the recovery cost for them? The undo cost is the average of the least cost of recovering; in practice a user would take more than the undo cost because they will be unlikely to know the device perfectly (and in any case they may be stressed after making an error, and may make further errors). The undo cost of the Graseby is 2.0; if it had an **Undo** button, the undo cost would be 1, and the risk of user stress (and further keying errors) increasing the cost would be eliminated.

The undo cost is measured by finding the all pairs shortest paths using them to find the average cost of paths corresponding to every graph arc reversed. There are clearly two sorts of undo cost: the basic undo cost is the average cost of undoing any action—but of course, some actions do nothing (the arcs are loops), so the normal undo cost is the average cost of undoing an action that has done something. Further, the basic undo cost can be refined: if the user does not notice an action has no effect, but they still want to undo it, then the undo cost for that action is at least 1 not 0. We could also weight costs with the probability the device is in particular states—for example, if it is less likely the user will get the device in an **Alarm** state, then the cost recovering from errors in this state should be weighted less. Which undo cost is the most insightful measure for a device depends on the domain, or a designer may wish to compare different undo costs to improve device performance, particularly if some forms of undo cost are significantly higher than others which would indicate they deserve closer inspection by the designer or analyst.

### 4.2 Undo Equivalents

For a device like the Graseby, which does not have a specific **Undo** action, it may be interesting to know which action or actions most often behave like an undo. For example, one might expect **UP** and **DOWN** to be mutual undos.

For the Graseby, the most common action that behaves like **Undo** is in fact **Timeout**: in other words, to recover from many errors, the user should simply wait until the device times out. In graph theory terms, for all arcs ( $uv$ ) on the Graseby if there is a reverse arc ( $vu$ ) most such arcs are labeled **Timeout**. The user should be trained to know the significance of timeout, since trying to do

anything to recover from an error merely delays the device doing the timeout. Also, the design of the device might be modified to tell the user (e.g., by way of an indicator) that a non-trivial timeout is possible in the current state, and moreover when the timeout would in fact behave like **Undo**.

### 4.3 Overrun Cost

The undo cost of a device is the average cost of recovering from *any* error. In contrast, the *overrun cost* of a device is the undo cost assuming that the errors the user will undo are overrun errors: the average cost of recovering from doing an action once too often. Many tasks require a user to press a button repeatedly, and it is very easy to press a button once too often. Or the user may press a button and not be sure they pressed it hard enough, so they press it again; now they have pressed it twice.

The overrun cost is specified as the average over all possible recovery costs: for all labels  $l$ , for every arc  $(uv)$  labeled  $l$ , if there is an arc  $(vw)$  also labeled  $l$  find the cost of the shortest path  $w$  to  $v$ .

The overrun cost for the Graseby is 1.66, which is better than the undo cost (which is 2). In other words, certain sorts of error (overrun being one) are easier to undo than average. The designer should collect some empirical data to find out what sort of errors users typically make. It is also important to know how users typically recover from errors.

### 4.4 On/Off or Reset Recovery Cost

Often a user will switch a device off and on again in their attempt to recover from an error (interviews with anaesthetists confirm it is standard practice). The optimal cost of an off/on recovery procedure is the cost of getting to **Off** (in general, at least one action) followed by returning to the *previous* state—there's no point returning to the error state. The appropriate cost measure is therefore the average of: for every state  $u$  and arc  $(uv)$ , the cost of the shortest path from  $v$  to **Off** then **Off** to  $u$ . For the Graseby, this *reset recovery cost* is 4.85 with a worst case cost of 7. Interestingly, these figures are little different from the characteristic path length (4.1, and worst case 8), so a user switching this device off and on again is not much worse than the average cost of doing anything—the anaesthetists' strategy seems sensible (and maybe a strategy one wishes to deliberately support by design).

In all cases above, we have assumed the user knows the optimal ways to achieve everything and that they can do the sequence of actions accurately, else their choices of actions will not be optimal, as the measures above assume. It is possible to measure costs based of assumptions of stochastic user behaviour, and this has been done at length elsewhere [1].

### 4.5 Errors in Small World Graphs

One measure of small world graphs (discussed in section 3.3) is the *cluster coefficient* [21], the probability that two neighbours of a vertex are connected. The

cluster coefficient can be considered to represent how easy it is for a user to correct a single incorrect action: that is, by doing something, they move from a state to its neighborhood, and if they wanted to be somewhere else in the neighborhood (anywhere else one action away from where they were), the coefficient is the probability they can get there with just one further action. The Graseby's cluster coefficient is 0.6.

The cluster coefficient is the average of all vertex clustering, but it is interesting to find the worst cases, since low clustering makes a state harder to 'adjust,' certainly harder to move around in its neighborhood, than a state with high clustering. For the Graseby, the three worst cases in this sense are **Infusing**, **Infusion suspended**, and **Continuous**—interestingly, all these states occur when the device is clinically active, where we can assume the operator does not want to change its mode either easily or accidentally (and this property is indeed what we find in the graph); whereas high clustering states are in fact highly 'interactive' parts of the Graseby, like **Off**, **Purging** and **Bolus**, all states whose clinical use is transient.

## 5 Knowledge

We can expect interactive systems to be easier to learn and comprehend the smaller they are, and the more regular their structure. We now look at other graph properties that relate to user knowledge—and that identify key areas for training.

### 5.1 Edge Connectivity

The *edge connectivity* of a graph is the minimum number of edges whose deletion would disconnect the graph; one distinguishes between connectivity and strong connectivity (see section 3.1), depending on whether edge direction is taken into account. For the Graseby, the strong edge connectivity is 1. This means that if a user does not know one particular arc, the system (or, rather, the user's model of the system) is effectively disconnected, and therefore there are some operations the user does not know how to do.

The *minimum cut* is the set of arcs (namely the *bridges*) that disconnects the graph. For the Graseby, the minimum cut is a single arc, the **On** for the state transition **Off** to **On**. We have thus *automatically* discovered what is (in hindsight) an obvious fact: if a user does not know how to switch on the Graseby (i.e., they do not know this action in this state), there are some operations they certainly cannot do!

If a device is not going to be redesigned, the edge connectivity and its dual, the *vertex connectivity* (and the set of *hinges*, vertices whose deletion disconnect the graph), highlight potential training issues. For many applications, most important thing to teach the user is the minimum cut, for this is the 'simplest' knowledge not knowing which will make the device very hard if not impossible to use.

## 5.2 Knowledge in Small Worlds Graphs

Small world graphs (discussed in sections 3.3 and 4.5) have interesting properties relevant to usability. They are resilient to failure (‘network robustness’). If a user does not know about some state, (on average) they can still find short paths from where they are to where they want to go.

Small world graphs have characteristic vertices called *hubs*, which are very strongly connected. If a user knows of one or more hubs, they will find a device very easy to use, because knowing a hub makes connection to many other states very easy. While not knowing about a hub can make a device very hard to use, knowing it makes using it much easier. Hubs are therefore worth identifying for training purposes. Not surprisingly, the main hub for the Graseby is the **Off** state, followed by **On** and **Infusing**.

Small world graphs apparently have usability benefits (for reasons as outlined above), and interestingly they arise naturally through *incremental* product development. For example, a new feature is likely to be attached adjacent to an existing hub vertex, therefore strengthening its role as a hub. One might therefore expect an iterative design process to develop a small world graph—this may be another reason to suppose that iterative design is a central design method for good HCI [4].

## 5.3 Planar Graphs and User Comprehension

A *colouring* of a graph is an assignment of labels (e.g., red, green. . .) to vertices of a graph such that no adjacent vertex has the same colour. The *chromatic number* of a graph  $G$  is the minimum number of labels that colours  $G$ . The most famous theorem of graph theory is the Four Colour Theorem, first proposed in 1852 but only proved in 1976, which states that a *planar graph* (i.e., a graph that can be drawn in the plane without any cross-overs, bridges or tunnels) has a chromatic number at most 4. A graph with unavoidably crossing arcs may have a higher chromatic number.

One reason to think planarity and chromatic numbers are relevant to usability is a conjecture about user comprehension: if the transition diagram of a device can be drawn with no crossing arcs, the diagram must in some sense be easier to understand. In fact the Graseby is not a planar graph, so drawing it (as in figure 2) inevitably requires some crossing lines. We look at another application of chromatic numbers in the next section.

## 6 Observability

We can use chromatic numbers (section 5.3) to think about what the user can, in the best case, observe about an interactive system. Although the Graseby is not planar (see above), nevertheless its chromatic number is 4. If we imagine the user could see each state’s colour and nothing else, then if fewer than 4 colours had been used, the user would not be able to tell when the device changed between some states. If the device displays the current state by some combination of

lights (e.g., LEDs) or text such as ‘pumping,’ ‘alarm,’ ‘on’ and so on, then its chromatic number is the minimum number of combinations of indicators that are required to communicate every state change to the user. More specifically, a system with chromatic number  $k$  needs at least  $\lceil \log_2 k \rceil$  indicators, e.g. lights or different texts. In fact the Graseby has no lights, but it does have an LCD panel that helps distinguish adjacent states.

### 6.1 Trackable and Knowable Systems

We may define a continuum of usability, delimited by three important properties of a device being *untrackable*, *trackable* or *knowable*. A trackable device allows the user to keep track of which state it is in, provided the user knows what they are doing; a knowable device allows the user to determine which state the device is in. If the number of distinct indicators is  $n$ , then a device is untrackable if  $2^n < k$  the chromatic number. A device is in principle trackable if  $2^n \geq k$ , but it is not knowable at least until  $2^n > N$  where  $N$  is the number of states.

In practice a device may allocate the  $n$  indicators in a peculiar way, so that the bounds are not realised. Thus we distinguish between trackable in principle (i.e., there are enough indicators) and trackable in practice (the indicators work such that every adjacent state has a different permutation of indicators); knowable, of course, means that every state, whether adjacent or not, has a different permutation. If adjacent (respectively, any) states do not have different indicators, then this suggests to the designer either there are too many states, too many arcs, the indicators or the indicator mapping,  $O$ , are badly designed. The Graseby is trackable but not knowable (in the sense defined above).

We can characterise trackable systems more precisely by looking at the average cost of knowing the state, i.e., the average number of user actions required to uniquely identify the current state. The higher this is the more difficult a user will find it to orient themselves when coming to the system in an arbitrary state, say, after a distraction. The maximum cost of knowing the state is also of interest here.

### 6.2 Chinese Postman Tour

The *Chinese postman tour* (abbreviated CPT) finds the shortest tour that visits every arc of a graph [16]. A person (whether designer or user) who claims to know a device must *in principle* know a Chinese postman tour—though *in practice* they need not be able to describe it (a standard psychological issue of being skilled but unable to explain the skill in detail—see the discussion below on the ‘practical’ CPT). The length of a CPT is a strict lower bound on the knowledge needed to be certain a user (or designer) knows a device. Reducing the CPT cost will therefore in general suggest or highlight potential improvements to a designer.

The length of the CPT for the Graseby is 710 button presses, not counting details such as password entry. This seems very long, and suggests the Graseby is unlikely to be understood fully by any users unless it has been designed with some systematic structure (which the CPT does not exploit). For example, the CPT must check every `Off` action for every state; presumably most devices are designed in such a way to ensure this property without needing to check it explicitly.

The nature of the CPT is clear from the following extract from the middle of a tour of the Graseby:

```

:
478 Try ON from "Off"
    goes to "On"
479 Try DOWN from "On"
    goes to "Value locked for On"
In state "Value locked for On", check unused buttons:
    DOWN, OFF, PURGE, UP, STOP, KEY, ON do nothing
487 Do ENTER from "Value locked for On"
    goes to "Continuous"
:

```

An implementor of a reverse-engineered device may wish to run through the CPT on both the device and the simulator to check that they correspond. Notice that doing a CPT may require testing many timeout transitions (24, or about five minutes total, for the Graseby or, rather, *24 as known from the simulation*—the real Graseby may require more), and therefore checking may take a very long time! Note, also, that the state names listed in the CPT are the implementation's state names, and these may or may not correspond closely to the device state names, if indeed the device makes it clear to users what state it is in (the Graseby uses a large LCD, which mostly displays text unique to the current state).

If a graph is *Eulerian*, it has a CPT of minimal length, namely a Eulerian tour, with each arc traversed exactly once (a CPT in general traverses some arcs more than once, therefore making it longer than a Euler tour). The Graseby is not Eulerian, and therefore some arcs must be revisited in a CPT. The CPT algorithm can determine the minimum number of arcs to adjoin to make a graph Eulerian; for the Graseby, this number is 30. Therefore long revisited paths could be designed-out of the CPT provided there are 'spare' out-arcs from vertices: namely, vertices with out-degree less than the number of user actions. It is trivial to modify a CPT algorithm to identify candidate pairs of vertices, but of course one would not necessarily want more arcs out of, say, the state **Off** than the single arc labeled On. Or again, some buttons have labels that characterise the states they go to, such as On goes to the **On** state (if the device was off); it does nothing else on the Graseby, but the CPT analysis suggests it could do more—but a user would probably *not* want On to do anything else.

The designer must therefore use discretion in interpreting the suggestions—for the Graseby, perhaps an arc labeled Start could usefully start an infusion even if the device was off, thus adding one more arc to **Off** and reducing the length of the CPT, and hence making the device easier to learn thoroughly.

### 6.3 Traveling Salesman Tour

The postman visits every arc (as it were, visiting every street/arc to deliver post), whereas the salesman visits every vertex (as it were, selling stuff in every city/vertex). The traveling salesman problem is to find the shortest tour that

visits every vertex. In user interface terms, this corresponds to visiting every state to check it works as intended (if a designer) or that it is understood (if a user). Assuming the actions are consistently designed, visiting every state may be sufficient to understand a device—the CPT is overkill on this assumption, as it assesses too much detail.

## 6.4 Practical Tours

If the CPT of the Graseby is 700+ user actions, this may be a useful indicator of the complexity of the user interface, particularly when compared to other designs or modified Graseby designs, but in practical terms the large number means a designer is unlikely to be able to follow the tour without making errors; they are also unlikely to be able to follow the tour in a single session. In either case, a more practical approach is required.

The Graseby simulation tracks which states and actions have been visited and used. Hence, rather than follow an error-free tour, the designer can follow a dynamically-generated tour that suggests their next action(s) to take the shortest path to the next unchecked part of the device, given that the simulation knows which states and arcs have already been checked (cf. figure 2).

More generally, since a design may change (or a simulation modified to be made more faithful to an actual device), the flags associated with every vertex and arc can be reset if the design changes and the change affects that item. Thus a designer can incrementally check a device, even while it changes, perhaps making errors or missing actions, and still know what needs doing—and eventually cover the entire functionality of the device.

The flags can be used in two further ways. During design, other documents may be produced, such as user manuals. A technical author may wish to flag that they have already documented certain parts of the device, and therefore that they must be notified if the flagged parts of the device change. Another use is for an auditor, who checks whether an implementation conforms to its specification. Again, they can use flags to assert that a vertex (or arc) has been checked out and must not be changed gratuitously. Both these ideas are implemented in [19].

## 7 History and Undo

A disadvantage of graph theoretic formalisms is that there are some standard user interface features that are cumbersome (but not impossible) to represent: history and undo.

Many devices ‘remember’ what they were doing before they were switched off; when they are switched on again, they go back to the state they were in before being switched off. (Statecharts represent this history by using a special notation.) Graphs can only represent this remembered history by embedding it as a subgraph within the **Off** state. If there is only one state that maintains a history, this is not a serious issue, but when there are several, the complexity of the subgraphs becomes hard to manage without help.



Many desktop applications, but surprisingly few interactive devices, support undo—which is curious given that undo has considerable benefits for users, and is particularly easy to implement for interactive devices. The simplest way to implement a device based on a graph was described above: the device tracks the current state using a variable  $s$ . To implement undo, the device model is changed from finite state automaton (section 2) to push down automaton, such that on every state change  $s$  is pushed on the stack. The action **Undo** simply pops the stack to update  $s$ . If undo is implemented like this, then the graph model does not represent undo, and it would be transparent to any analysis based on the graph.

An alternative approach is to modify the basic graph to support undo. (This is an example of the general procedure of taking a device specification as a graph and introducing some required feature, in this case undo.) An undo graph can be defined informally: given a graph  $g$ , the undo graph  $U(g)$  replaces every vertex  $v$  of  $g$  with a set of  $n$  vertices  $U(v)$  where  $n$  is the in-degree of  $v$ . Each vertex in  $U(v)$  has exactly one incident arc, and the same out arcs as  $v$  together with an additional arc labeled undo that returns to the source of the incident arc. Generally  $U$  will be applied to a subgraph—for example, we do not generally require **Undo** to work if the last action was **Off**.

History (as in statecharts) is much harder to conceptualise in graph theoretic terms. For every component of  $n$  vertices with a history,  $n$  copies of every other vertex must be made; essentially if a graph has two components  $U$  and  $V$ , with  $V$  having a history, then  $U$  must be replaced by  $U \times V$ . In practice many devices have history. A common example is a TV that returns to the last channel watched when it is switched on: implying the **Off** state is a set of 100 or so vertices, so the single on transition from each off vertex can return to the last-used channel.

## 8 Misconceptions

One might imagine that graphs have disadvantages because many graph properties are computationally hard. For example, if we allow arcs to be conditional on arbitrary conditions (as they are in statecharts and Kripke models) then many otherwise routine graph theoretic properties turn on undecidable questions. Or finding the largest cycle in a graph is an NP-complete problem. On the other hand, any such property would be correspondingly hard in any other formalism too. In short, the disadvantage of graphs, if any, is not that some properties are hard, but that it can be deceptively easy to express hard properties!

An astronomical number of vertices may be needed to represent some programs. One might therefore imagine that graphs for real systems would necessarily be enormous, and impractically so. This, however need not be a problem in practice, for at least two reasons. First, we do not need to represent graphs explicitly: for example, SMV has an underlying model (a Kripke model) but a typical user of SMV would never see it, nor its efficient representation as a BDD. Second, whatever the theoretical potential for detailed representation, we as HCI evaluators need only use graphs to model the user interface behaviour (not the underlying model in the MVC sense). Such graphs are *much* smaller;

indeed, a user interface that required a user to know or model billions of states would certainly be unusable! Instead, users model an abstraction of the implementation; to the extent we can capture that abstraction graphs will be an ideal tool to model user interfaces.

## 9 Further Work

Further work can be divided into three areas: the development of convenient APIs, CASE tools or languages for programming interactive systems, the development of convenient analysis tools (particularly ones that do not require mathematical expertise to use), and further research into the underlying principles and the usability/model correspondences.

As for specific further research, the following ideas might be considered:

- There are many ways in which user testing could validate the use of graph theory in HCI and to provide a better understanding of its use in redesign, e.g., priorities in different design contexts, relationship to other methods. Although graph theory has strong face validity, and there are cases where its use may be critical to safety, we do not know how useful it is given the huge number of other pressing design issues that confront real projects; on the other hand, all graph theoretic measures can be automated, and doing so would be a first step towards testing validity experimentally.
- Of the ‘off the shelf’ graph theoretic properties that are useful for HCI, define them in CTL or other logic (see [8] for some examples). Doing this would produce a useful collection of design principles, and perhaps even a benchmark collection for proposed HCI methods.
- Since history and undo are operations on graphs, an interesting research project would be to optimise algorithmic graph theory for such graphs. For example, shortest paths are unchanged by undo, and therefore can be found as efficiently in a graph with undo as without provided the underlying graph is known.
- The user model and the user manual can be represented as graphs. What properties do such graphs have, and what are useful relations between these graphs and the system implementation graph? For example: if the user model is a subgraph of the system, the user need ‘never’ make a conceptual mistake with it; if the user manual is a spanning tree of the system, it describes it ‘fully.’
- We identified small world graphs as being relevant to navigation, error and knowledge. These graphs, and scale-free networks, seem highly relevant to HCI, but this relationship has not yet been explored thoroughly.
- The states **On** and **Off** occur frequently in results, which may reassure us that the methods are picking up interesting states (graph theory does not know what the names of these states mean, nor their purpose—so these states are picked out by their *structural* significance), but it suggests that more useful analyses could be made of subgraphs, for instance by deleting vertices the designer knows about, such as **Off**. This is easy to do (unfortunately this

paper did not have space to explore the results), but it is not obvious how to generalise the idea, and therefore raises a specific graph theoretic research agenda.

- Many of our analysis techniques could be extended to more accurate models of interactive systems by using weighted vertices and edges, as we discussed for the undo cost (see section 4.1), and by accounting for non-determinism. And where average metric values are used, more detailed information about an interactive system could be found by looking at the distribution over all vertices or arcs.

## 10 Conclusions

Generally, working programs, user interfaces, HCI concerns and formal specifications live in different worlds. If a program works and is therefore available for user testing, iterative design and so forth it is very unlikely to still have an accurate specification. Thus, programming, usability and formal methods in HCI have traditionally diverged, and have few overlapping applications or case studies. This paper has shown that graph theory provides an easy way to implement programs *and* to retain an explicit specification, even as programs undergo modification; and that specification can be readily analysed for various HCI concerns. Although graph theory is not unique in this respect (e.g., consider statecharts and Statemate [6]), graph theory does provide a very rich and fruitful domain to explore HCI properties *as well as* a very efficient model to implement user interfaces. Unlike systems like Statemate, graph theory is standard mathematics and is non-proprietary.

Our claims have been substantiated in this paper by providing a variety of graph theoretic properties and discussing their significance to HCI design decisions, including several diverse applications of small world graphs. We evaluated these properties from a working implementation, namely a simulation of a Graseby 9500 syringe pump. The case study showed how graph theoretic analysis raises many potential design questions, as well as many user training issues. Our analysis introduced many interesting new research questions, such as the relevance of small worlds models to HCI.

*Acknowledgements.* Harold Thimbleby was a Royal Society-Wolfson Research Merit Award Holder, and acknowledges this support for the research described here. The authors are grateful for many collaborations with Michael Harrison and Paul Cairns, and comments from several anonymous reviewers.

## References

1. Cairns, P., Jones, M., Thimbleby, H.: Usability analysis with markov models. ACM Transactions on Computer-Human Interaction 8(2), 99–132 (2001)
2. Chartrand, C., Lesniak, L.: Graphs & digraphs. Chapman & Hall, Boca Raton (1996)

3. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. *Software—Practice and Experience* 30(11), 1203–1233 (2000)
4. Gould, J.D., Lewis, C.: Designing for usability: Key principles and what designers think. *Communications of the ACM* 28(3), 300–311 (1985)
5. Graseby Medical Ltd. 9500 Ambulatory Infusion Pump for Epidural Analgesia: Instruction Manual (2002)
6. Harel, D., Politi, M.: *Modeling Reactive Systems with Statecharts*. McGraw-Hill, New York (1998)
7. Knuth, D.E.: *The Stanford GraphBase*. Addison Wesley, Reading (1994)
8. Loer, K.: *Model-based Automated Analysis for Dependable Interactive Systems*. PhD thesis, Dept of Computer Science, University of York, UK (2003)
9. Myers, B.: Past, present, and future of user interface software tools. In: Carroll, J.M. (ed.) *Human-Computer Interaction in the New Millenium*. Addison-Wesley, Reading (2002)
10. Parnas, D.L.: On the use of transition diagrams in the design of a user interface for an interactive computer system. In: *Proceedings 24th. ACM National Conference*, pp. 379–385 (1964)
11. Payne, S.J., Squibb, H.R., Howes, A.: The nature of device models: The yoked state space hypothesis and some experiments with text editors. *Human-Computer Interaction* 5, 415–444 (1990)
12. Pemmaraju, S., Skiena, S.: *Computational discrete mathematics*. Cambridge University Press, Cambridge (2003)
13. Thimbleby, H.: Combining systems and manuals. In: *BCS Conference on Human-Computer Interaction VIII*, pp. 479–488. Cambridge University Press, Cambridge (1993)
14. Thimbleby, H.: Formulating usability. *ACM SIGCHI Bulletin* 26(2), 59–64 (1994)
15. Thimbleby, H.: Permissive user interfaces. *International Journal of Human-Computer Studies*, 54(3), 333–350 (2001)
16. Thimbleby, H.: The directed chinese postman problem. *Software—Practice & Experience* 33(11), 1081–1096 (2003)
17. Thimbleby, H., Addison, M.A.: Manuals as structured programs. In: Cockton, G., Draper, S., Weir, G. (eds.) *People and Computers IX, Proceedings of HCI 1994*, pp. 67–79. Cambridge University Press, Cambridge (1994)
18. Thimbleby, H., Addison, M.A.: Intelligent adaptive assistance and its automatic generation. *Interacting with Computers* 8(1), 51–68 (1996)
19. Thimbleby, H., Ladkin, P.B.: A proper explanation when you need one. In: Kirby, M., Dix, A., Finlay, J. (eds.) *People and Computers X, Proceedings of HCI 1995*, pp. 107–118 (1995)
20. Wasserman, A.I.: Extending state transition diagrams for the specification of human-computer interaction. *IEEE Transactions on Software Engineering* 11(8), 699–713 (1985)
21. Watts, D.J., Strogatz, S.H.: Collective dynamics of ‘small-world’ networks. *Nature* 393, 440–442 (1998)

## Questions

**Prasun Dewan:**

*Question: What can graph theory do that extends beyond dialogue models developed for command-based systems?*

Answer: It is consistent with that work. I'm doing work that was proposed in the 1960s. The value of that early theory has been lost in the complexity of other HCI issues. A typical usability study wouldn't find many of the issues that can be found in a few minutes using a graph theoretic approach.

**Ann Blandford:**

*Question: How would your approach deal with your early example of the nurse pressing the wrong button?*

Answer: It doesn't deal with that issue directly, but it can explore all possible ways of pressing wrong buttons and the consequences. It can help to design more generally usable interfaces.

**Michael Harrison:**

*Question: Could you characterize the scope of what you propose in terms of the kinds of property it will identify versus those it won't?*

Answer: They're graph-theoretic problems! It answers some dependability problems where you want to be certain that a system doesn't have certain problems. But it won't find every problem, such as perceptual issues.

**Kirstin Kohler:**

*Question: What happens when the number of nodes is too large (e.g., business applications)?*

Answer: Size isn't in practice a problem. Colleagues are working with systems of millions of states. However, users need to have a model of the system so such complex systems are almost certainly not usable.

# Mathematical Mathematical User Interfaces

Harold Thimbleby and Will Thimbleby

Department of Computer Science, University of Swansea, SWANSEA, Wales  
harold@thimbleby.net, will@thimbleby.net

**Abstract.** Taking *Mathematica* and *xThink* as representatives of the state of the art in interactive mathematics, we argue conventional mathematical user interfaces leave much to be desired, because they separate the mathematics from the context of the user interface, which remains as unmathematical as ever. We put the usability of such systems into mathematical perspective, and compare the conventional approach with a novel declarative, gesture-based approach, exemplified by *TruCalc*, a novel calculator we have developed.

## 1 Introduction

*TruCalc* is a new calculator, with a gesture-based handwriting recognition user interface. This paper reviews its design principles and relates them to the requirements of mathematical user interfaces.

## 2 The Development of Mathematical User Interfaces

For thousands of years, we've been doing maths by using pencil and paper (or equivalent: quill and scroll, stick and sand—whatever). When calculating devices were invented, this helped us do calculations faster and more reliably, but we still did maths on paper. Comparatively recently, computers were invented, and for the first time we could replace pencils with typed text and get results written down automatically, and then, later, we replaced paper with screens. Mathematics displayed on screens can be manipulated more freely than ever before, yet most calculators running on computers emulate mechanical devices.

Turing famously presented a formal analysis of what doing mathematics entailed [17]. He argued any pencil and paper workings could be reduced, without loss of generality, to changing symbols one at a time from a fixed alphabet stored on an unbounded one dimensional tape. Symbols are changed according to the current state of the device, the current symbol on the tape, and elementary rules. The Turing Machine, which can be defined rigorously (and in various equivalent forms), was a landmark of mathematics and computing. Indeed, the Church-Turing Thesis essentially claims that all forms of computing, and hence mathematics, can be 'done' by a Turing Machine in principle.

Turing introduced his machine with the following discussion:

“Computing is normally done by writing certain symbols on paper. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will

be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper.”

A. M. Turing [17]

Here, Turing’s use of the term ‘computing’ is historical; he is referring to human computation on paper.

While Turing is formally correct, good choice of notation is crucial to clear and efficient reasoning. Moreover, almost all notations (for example, subscripts) are two dimensional, as suits pencil and paper—and the human visual system. One view of the present paper is that the power—the ‘Turing equivalence’—of typical mathematical user interfaces has blinded us to the importance of notation and interactive notation properly integrated with the way the user interface works. Users put up with one-dimensional and other limitations to interaction because the deeper ideas appear sufficiently well supported. A very interesting discussion of Turing Machines and interaction is [3], but the focus of this paper now turns to the design of interactive mathematical systems.

## 2.1 Conventional Mathematical Interaction

Without loss of generality, mathematicians use pencil, paper and optionally erasers. Pencils are used to draw forms, or to cross them out. Typically, adjacent forms are related by a refinement. Harder to capture formally, the mathematician’s brain stores additional material, which is typically less organised than the representation on paper. One might argue that much of the mathematician’s work is to find a relation between what is in their head and marks on paper. This is an iterative process. Finally, the concepts and previously unstated thoughts are mapped to some representation such as  $\text{\LaTeX}$ , so that the organised and checked thoughts can be communicated effectively to other brains.

When this process is computerised, the forms are linearised into some character sequence. A string, typed onto ‘paper’ or a VDU left to right, is transformed by the computer inserting the values of designated expressions. A typical hand-held calculator is an example of this style of interaction, though most only display numbers and not the operators—one of their limitations is that the user does not know whether the display is the current number being entered or a result from a previous computation.

Around the 1970s, the sequential constraint became relaxed: the underlying model remained incremental as before, but the user could ‘scroll back’ and edit any string. Now the values computed may have no relation to the preceding strings, because the user may have changed them: the old output may be incorrect relative to the current string.

More recently, from the late 1980s on, the user interface supported multiple windows, each separately scrollable and editable, each with an independent user interface much like a typographically tidied up 1970s VDU. Of course, this gives enormous flexibility for managing various objects of mathematical concern (proofs, tactics, notes. . .) [10], especially when supplemented with menus and keyboard commands, but the generality and power should not distract us from

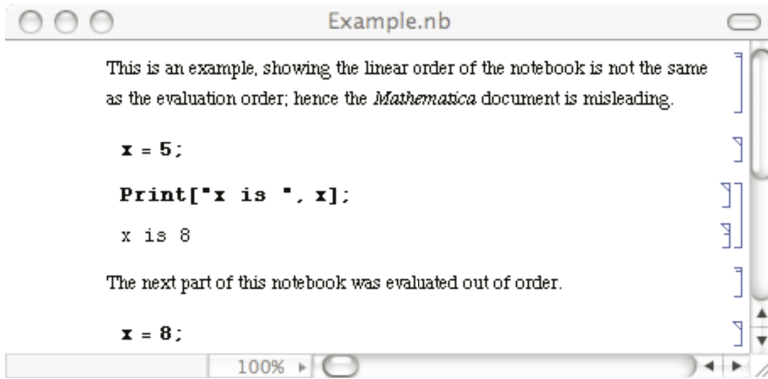


Fig. 1. Example of problematic interaction in *Mathematica*

the relation of the user interface to doing the mathematics itself. Normally we focus on the maths, and ignore the interface; it is just a tool to do the maths, not of particular mathematical interest itself.

Consider *Mathematica* [18]. A *Mathematica* notebook is a scrollable, editable document representing the string. Certain substrings in the notebook are identified, though the user can edit them at any time and in any order. A set of commands, typed or through menu selection, cause *Mathematica* to evaluate the identified substrings, and to insert the output of their evaluations. It is trivial to create *Mathematica* notebooks with confusing text like that shown Figure 1, which illustrates the inconsistency problem (is  $x$  5 or 8?) as *Mathematica* separates the order of the visible document from the historical order of editing and evaluation. In the example above, the  $x = 5$  may have been edited from an earlier  $x = 8$ ; the `Print` may have been evaluated after an assignment  $x = 8$  evaluated anywhere else in the notebook; or the `Print` may have been edited from something equivalent to `Print["x is 8"]`—and this is not an exhaustive list. In short, to use *Mathematica* a user needs to remember what sequence of actions were performed. (In fact, *Mathematica* helps somewhat as it can show when a result is possibly invalid.)

Although the presentation can be confusing, the flexibility is alluring. While the mathematician can keep the editing and dependencies clear in their head, the notebook (or some subset of it) will make sense.

*Mathematica* and many other systems add notational features so they can present results in conventional 2D notation. Instead of writing a linearised string, such as  $1/2$ , the user selects a template  $\frac{\blacksquare}{\blacksquare}$  from a palette of many 2D forms. The  $\blacksquare$  symbols can then be over-typed by 1 and 2, to achieve (in this example),  $\frac{1}{2}$ . Such mechanisms allow the entry of forms such as

$$\int_0^{\infty} \sin x^2 e^{-x} dx \quad \text{and} \quad 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \dots}}}}$$

as shown with relative ease. However, a problem is that the template continues to exist even though the user cannot see it. A simple example illustrates the



problem: editing  $\frac{1}{2}$  to 12 is difficult, because the initially hidden template will reappear explicitly in intermediate steps such as  $\frac{1}{12}$  or  $\frac{1}{1}2$ .

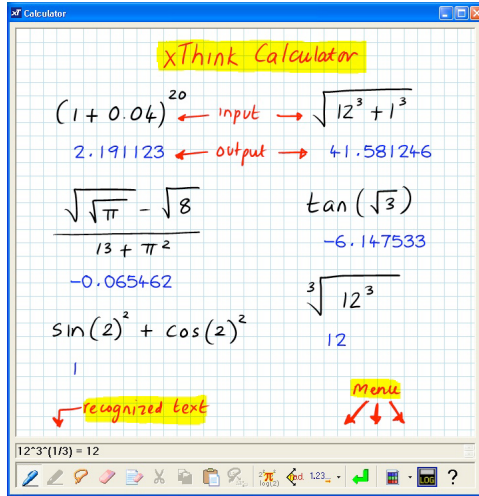
In *Mathematica* a function `TraditionalForm` achieves the inverse: presenting evaluations using standard 2D notation. While these 2D notations look attractive (and indeed are considerably clearer for complex formulae, especially for matrices, tensors and other such structures), they do not alter the semantics or basic style of interaction.

Padovani and Solmi [5] provide a good review of the interaction issues of using 2D notations, such as *Mathematica* and other systems use. They argue that 2D notation requires a model, namely the internal representation of the structure, which is not visible in the user interface. Hence, for the user to manipulate the 2D model new operations are required. The model itself is not visible, so inevitably 2D notation introduces modes and other complexities. That is, it looks good, but is hard to use. Editing operations are performed on non-linear structures (e.g., trees), but the displayed information does not uniquely identify the structure. Like the criticisms of *Mathematica* above, to use a 2D structure requires a user to remember how they built it; worse, what the user has to remember (Padovani and Solmi argue) does not correspond with the user's mental image of the mathematics being edited.

*xThink* is a different mathematical system [19], and its model is directly based on a 2D representation. *xThink* recognises the user's handwriting in standard notational format, and can compute the answer which is displayed adjacent to the hand-written sum. Provided *xThink* recognises the user's writing reliably, the internal model of the formula is exactly what the user wrote. Nothing is hidden. In this sense, *xThink* solves the problems Padovani and Solmi elaborate, though not all of the problems we attributed to *Mathematica* (as we shall see below).

A typical "page" from *xThink* is shown in Figure 2. Its advantage over *Mathematica*'s template-based approach is the ease and simplicity of entering mathematics, however its interaction style retains the problems of *Mathematica*'s—there is no guarantee the 'answers' are in fact answers to the adjacent formulae, and furthermore *xThink* has introduced new handwriting recognition problems; that is, the formula evaluated may not *ever* be one that was thought to have been written down!

*xThink* and *Mathematica* are only two examples, selected from a wide range of systems. Maple [2], for example, is closer to *Mathematica* in its computer algebra features, but closer to *xThink* in its handwriting recognition. However, Maple uses handwriting recognition to recognise isolated symbols which are written in a special writing pad—whereas *xThink* allows writing anywhere, but the writing has to be selected (by drawing a lasso around it) before it can be recognised. *xThink*, *Mathematica* and Maple are PC-based systems, and there are also many handheld mathematics systems, such as Casio's ClassPad [1], which allow pen-based input. However, rather than review individual systems, this paper now turns to principles underlying mathematical interaction.



**Fig. 2.** Example of *xThink*, showing natural handwriting notation combined with calculated output. Picture from *xThink*'s web site [19]; the original is in several colours, making the input/output distinctions clearer than can be shown in greylevels. In the picture, *xThink* has just parsed a handwritten  $\sqrt[3]{12^3}$ , shown its interpretation at the bottom of the screen (as  $12^3 \wedge (1/3) = 12$ ), and has inserted a result in a handwriting-like font below the formula.

## 2.2 Principles for Mathematical Interaction

With such a long and successful history of procedural interaction it is hard to think that it could be improved; systems like *Mathematica* are Turing Complete (upto memory limitations). Interactive mathematical systems, such as *Mathematica* and *xThink*, are clearly very powerful and have a very general user interface. The book *A = B* [6] gives some substantial examples of what can be achieved.

It is interesting to observe that the representations these mathematical system work with are *not* referentially transparent nor are they declarative. That is they only do mathematics that is 'delimited' in special ways, and the user has to 'suspend disbelief' outside of the theatre that is so delimited. As a case in point, we gave the example above of *x* not having the value it appeared to have (see Figure 1); even allowing for the semantics of assignment, there is no model like lvalues and rvalues that maintains referential transparency [9], without some subterfuge such as having a hidden subscript on all names—which, of course, must exist in the users' mind (if at all) if users are to do reliable mathematical reasoning.

Such Fregean properties as referential transparency<sup>1</sup> are key to reliable mathematical reasoning. Another is his idea of 'concept' that has no mental content, that is, a concept is not subjective. Most interactive systems *require* the user to conceptualise (i.e., make a mental model of) the interaction; they have modes, hidden state dependencies, delays, separated input and output and so on.

<sup>1</sup> Quine introduces the term referential opacity but attributes the idea to Frege [7].

It is ironic that modern mathematical systems are so flexible that they compromise the core Fregean principles—though [12] shows, under broad assumptions, any string-based (i.e., Turing equivalent) user interface interaction properties such as modelessness and undo are incompatible. Modelessness is, of course, an HCI term covering issues such as side effects, referential transparency, declarativeness, substitutivity, etc. Essentially, a purely functional interface is modeless; if one cannot have modelessness and undo (under the assumptions of [12]), any such user interface must be compromised for mathematical purposes. Such observations beg questions: is it possible to modify the style of interaction to preserve the core mathematical properties—and what would be gained by doing so?

### 3 Modern Mathematical Interaction

We will use *xThink* below to make a side by side comparison with our novel interface, *TruCalc*, to highlight the difference between a truly mathematical system and one that is not.

**Note.** *xThink* is a commercial application available from [19] (PC only), and *TruCalc* from [16] (Mac, PC, Linux).

Both our calculator and *xThink*'s calculator from first glance appear to do the same things. In fact *xThink*'s calculator seems to be more powerful, it can handle annotation, multiple sums, more complex mathematics. Yet ignoring a bullet point comparison and the superficial similarity of the two programs, they are in fact very different.

Both calculators provide a user interface based on handwriting recognition. But this is where the similarity ends!

Our calculator, *TruCalc*, was designed from generative user interface principles [12]; in contrast, *xThink* seems to merely add the idea of utilising the affordance [4] of pen and paper without escaping *Mathematica*-style problems.

To better illustrate the differences between these two superficially similar interfaces we will describe the interaction a user employs to solve a simple sum, along with the potential pitfalls.

#### 3.1 *xThink* vs. *TruCalc*

A first example problem we compare finding the value of “ $(4 + 5)/3$ ” in *xThink* and in our calculator, *TruCalc*. In both, the user starts by writing the sum on the screen, using a pen (or using their fingers on suitable touch-sensitive screens).

---

**1a** In *xThink*, the user must press a button to change *xThink* into selection mode. The user can then select what they have written. They must now press another button to get the selected handwriting recognised. The handwriting is recognised and represented in a separate window, which the user must read to check the accuracy of the handwriting recognition. If the handwriting is misrecognised by *xThink* then without checking the small text at the

bottom of the screen the user can easily be fooled into thinking they have the correct answer. The text at the bottom of the screen is both small and linearised, losing the benefit of the handwritten 2D notation—for example Figure 2 shows the cube root of twelve cubed being calculated, it is printed as  $12^{\wedge}3^{\wedge}(1/3)=12$ .

**1b** In *TruCalc*, as the user writes, the hand-written characters and numbers are converted to typeset symbols *without any further user action*. The user feels as if they are writing in typeset characters, and confirming recognition is as natural as checking your own handwriting is legible.

**2a** In *xThink*, to determine the answer, the user must now press another button to evaluate the recognised formula, and the answer is then displayed somewhere on the screen. In Figure 2 all such answers have been positioned under their respective formulae.

**2b** In *TruCalc*, the typesetting *includes* solving the equation. In this case, the screen will show a typeset  $\frac{4+5}{3} = 3$ —the user wrote  $\frac{4+5}{3}$  and the computer inserted  $= 3$  *in the correct position*.

**3a** In *xThink*, to determine the answer, the user's input must be syntactically complete (an expression). For example, to find the value of  $\sqrt{4}$  the user must write exactly this (and it must be recognised correctly).

**3b** In *TruCalc*, answers are provided even with incomplete expressions, as well as with equations. For example, to find the value of  $\sqrt{4}$  the user can write  $\sqrt$  then 4, or 4 then  $\sqrt$ , and they can write  $=$  if they wish. In any case, the value 2 or  $=2$  is also displayed. Furthermore, if the user wrote  $\sqrt = 2$ , then *TruCalc* would insert 4 appropriately, thus solving a type of equation where *xThink* would require the user to write  $2^2$  (which is notationally different).

**4a** In *xThink*, the user's handwriting can be altered and hence make the answer (here, 3) invalid—and it will remain invalid until the handwriting is re-selected, recognised and re-evaluated (and the old answer removed). Or several answers may accumulate if the user evaluates formulae and does not remove old answers.

**4b** In *TruCalc*, as typesetting *includes* solving the equation, the user could continue and write  $=$  or  $= 3$  themselves. In particular, if they wrote an equation, such as  $\frac{4+}{3} = 3$ , *TruCalc* would solve it, and insert (in this case) 5.

**5a** *xThink* provides no other relevant features for the purposes of this paper.

**5b** In *TruCalc*, the editing of the user's input is integrated into its evaluation. Thus the user can then continue to write over the top of this morphed equation, adding in bits that they consider are missing. For example, if the RHS 3 is changed to 30, the display would morph to  $\frac{4+86}{3} = 30$ . It is possible to edit by inserting, overwriting and by drag-and-drop to a bin to delete a selection, or to other parts of the equation to move it. In all cases, the equation *preserves* its mathematical truth, as *TruCalc* continually revises it. *TruCalc* also provides a full undo function, which animates forwards and backwards in time—also showing correct equations.

### 3.2 In-Place Visibility

With *TruCalc* the replacement of the user's handwriting with typeset symbols not only provides an immediately neat and tidy (and correct) equation but also provides immediate visible feedback of what was recognised. The displayed typeset equation *is* the equation that the answer is shown. This in-place visibility removes confusion and misunderstanding over what the calculator is doing, and whether it has misrecognised bad handwriting.

In our experiments with *TruCalc* [14], one of the outstanding results was that whilst users made intermediate errors, *no* user stopped on a wrong answer. We believe this was because the calculation they were performing was entirely visible and unambiguous to them in an in-place 2D notation.

Without in-place visibility, the user may be unsure which results correspond with which inputs. This compromises mathematical reliability; the user has to rely on their head knowledge.

### 3.3 No Hidden State; Modelessness

Hidden state and modes compromise mathematical reasoning. Hidden state affects how to interpret input and output; specifically, modes are hidden state (e.g., knowledge of history) in the user's head that is needed to know how to control the user interface predictably.

Typically, a system does not show what mode it is in, but the mathematical interpretation of its display depends on the user knowing some hidden state. For example, in *xThink* to erase or move parts of the equation the user has to select different tools at the bottom of the screen, then when they have finished they have to remember they are in a special mode and reselect the pen tool. The *xThink* interaction style makes this cumbersome approach unavoidable in principle. The relative meanings of displayed results obviously changes when other images are modified; simply, they may become wrong.

The *xThink* user also has to be aware that once they have finished an equation they have to do more (press several buttons, select their text) this time switching mental modes from “entering” to “getting the answer.” If they don't change modes (or of they don't change through the modes appropriately, or select inaccurately), there is either a wrong result or no result for the problem.

With *TruCalc* there are *no* hidden modes or state, and no user context switching. Not only is there no menu of different tools but there is no need to switch mental modes or to pause and press an **Enter** button to make things work. This greatly simplifies the user's mental model and reduces the effort required to use the calculator. *TruCalc* does have a few modes, for example a dragging mode, but these are clearly visible and they are directly initiated and controlled by the user.

Note that in-place visibility and modelessness together give a very strong—and easy to use—interpretation of WYSIWYG (what you see is what you get), as proposed in [11].

### 3.4 Instant Declarativeness

A system may show the mathematically right answer when the user asks for it; but until they ask for computation, the mathematics is strictly incorrect (or possibly shows a representation of a meta-‘undefined’). In *TruCalc* the results are ‘instantly’ correct, with no user action required.

Declarative programming was popularised through Prolog. Essentially, the programmer writes true statements, ‘declaring’ them, and Prolog backtracks to solve the equations (sets of Horn clauses in Prolog). Prolog is thus a declarative language—though its user interface isn’t.

Likewise, *TruCalc* is declarative. The user writes equations (or partial equations, taking advantage of the automatic syntax correction), and these are declarations that *TruCalc* solves (by numerical relaxation).

In Prolog, the user has to enter a query, typically terminated by a special character. Until that character is pressed, the output (if any) is incorrect. This inconsistency within the interface is what we are used to, even to the extent of accepting the sort of inconsistencies illustrated in Figure 1. But it requires the user to remember the past; they haven’t pressed return or some other special character or menu selection yet. If they forget confusion happens.

*TruCalc* extends declarativeness to *instant declarativeness*, that is, an interface that is always true all of the time. No matter what the user writes the answer shown is *always* correct.

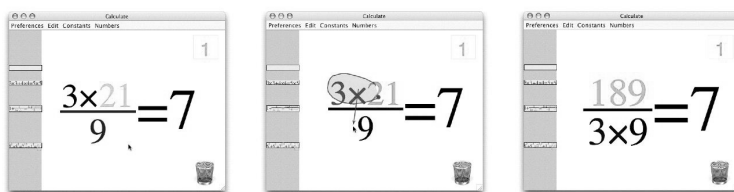
An instantly declarative interface implies that the calculator has to be showing something that is correct even if the user has not finished entering everything, or has a currently incorrect edit. Thus the calculator also has to cope intelligently with partial expressions like  $\div 3+2$ . In our case the calculator fills in place holders that alter the expression as little as possible. There are also problems like  $1/0$  or overflow like  $10^{10^{10}}$ —these too can be handled by correction (such as showing  $1/0$  as  $1/(0+1)$ ; see [13]), or by changing the algebra implemented by *TruCalc*.

This instant declarativeness removes the disparity between the input and the output, removing an enormous potential for user confusion and it also removes the need for the user remembering having to press the “equals” button (or some other change mode button) to get an answer.

The implementation of instant declarative user interfaces is only slightly more complex than conventional user interfaces; at least two threads are required, one for the user input, one for processing. Processing restarts every time the user extends or changes the input; in *TruCalc* there is a short delay, which allows the user to write an expression fluidly without visual interference from it being morphed into recognised text until they finish or pause.

### 3.5 Equal Opportunity

The power of *TruCalc*’s implementation of instant declarativeness combines powerfully with equal opportunity [8]. Unlike *xThink*, *TruCalc* does not distinguish in principle between the user’s input and its own output. Each has ‘equal opportunity’ in the equation. This makes it possible to write on both sides of an equality.



**Fig. 3.** Example of drag and drop interaction in *TruCalc*, shown as three consecutive screen-shots. Initially, the user has written  $\frac{3x}{9} = 7$ ; next, the user drags the  $3x$  numerator to the denominator; finally, *TruCalc* provides the correct numerator. The *only* user interaction to achieve this transformation is to draw the loop (shown in the middle figure) and drag it. Had the user had dragged the  $3x$  to the wastebasket, it would have been deleted, and the equation would be corrected to  $\frac{54}{9} = 7$ . (If a loop is drawn not containing anything to select, it is recognised as a zero).

The ability to change either the answer or the question lets a user solve problems simply that they would have struggled with otherwise. For example, “what power of 2 is 100” can be solved directly without logarithms. (For example, the user writes  $2 = 100$ , which is corrected to  $2 = 100 - 98$ , then writes a decimal point as the exponent of 2, which is where they want the answer.  $2 \cdot = 100 - 98$  then morphs to  $2^{6.643856} = 100$ .)

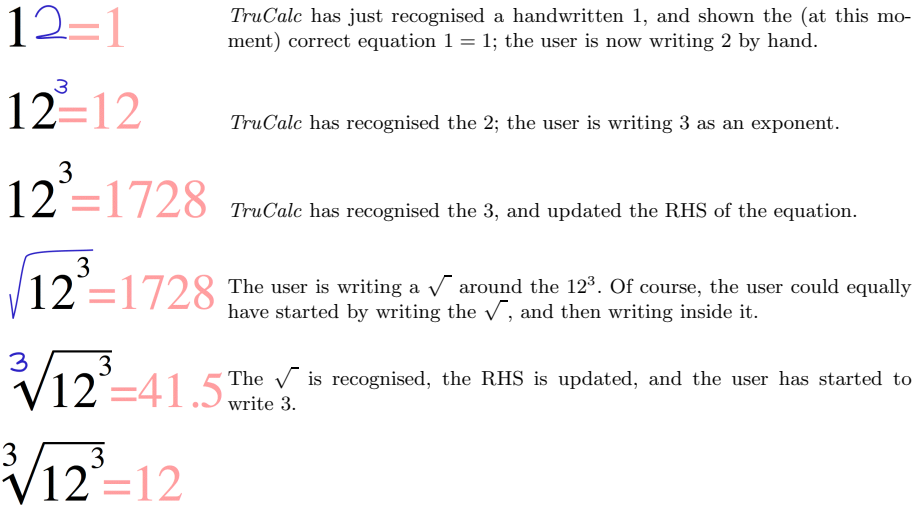
Equal opportunity is not in itself a feature that is required for a highly mathematical user interface, but it is a natural generalisation (from expressions to equations) that significantly increases the power of the user interface for the user.

### 3.6 Rearranging

In *xThink*'s calculator it is possible to delete things or move them around but it is always an awkward process involving many mode changes and it is fairly limited in what it achieves. Moreover, *any* editing in *xThink* breaks the relation between written input and calculated output, and the user has to remember to re-evaluate an edited formula. Hence, in *xThink* the ability rearrange introduces modes and hidden state.

In *TruCalc* the ability to drag and drop an arbitrary part of the equation elsewhere is synchronised by *TruCalc*'s ability to morph the result into a new typeset equation. It is therefore possible to move parts of the equation around without regard for their size or shape, and the user *always* sees a fully correct equation.

More specifically, in *xThink* drag-and-drop is achieved by choosing the selection tool, drawing around the object, then dragging, then choosing the next tool to use; however, once moved, the formula typically needs explicitly selecting, recognising, and evaluating, as further steps for the user. In *TruCalc* drag-and-drop is achieved by drawing around an object and moving it. No mode change is required, and no action needs to be taken to evaluate the new formula. Figure 3 illustrates some simple examples.



**Fig. 4.** A step-by-step, broken-down example of using *TruCalc* on the sum that *xThink* is shown solving in Figure 2, showing how a single equation changes as the user writes on it. This brief example does not show drag-and-drop, nor equational calculations. However, notice that *TruCalc* provides continual correct feedback; there are no hidden modes, no special commands—*TruCalc* just ‘goes ahead’ and provides in-place answers. The user feels as if they are writing in a formal typeface (here, Times Roman). This brief example does not show how *TruCalc* would handle solving equations, for instance if the user dragged the 12 onto the RHS. Had the user written an = themselves on the left of their formula, then the answers would have been shown on the LHS.

## 4 A Demonstration of *TruCalc*

Because *xThink* is not highly interactive, ironically, its screen shots (such as Figure 2) make it easier to understand than screen shots of *TruCalc*! *xThink*’s screen shots show handwriting input, the recognised input (shown in the bottom pane), and the result. Figure 2 shows several such examples. It looks straight forward—except, as we showed in Section 3.1, *constructing* the interesting display of Figure 2 requires transitions between many modes, and hence possible user errors. Figure 4 shows *TruCalc* solving the problem that *xThink* is shown solving in Figure 2; however, *xThink* solves the equation in one step and requires changing modes, whereas *TruCalc* solves continually, in place, and needs no modes at all. (In this short paper we do not illustrate how *TruCalc* can solve equations more powerfully than *xThink*—by combining rearranging with equal opportunity; see [13] for examples.)

## 5 Other Features of *TruCalc*

*TruCalc* provides other features that make it more powerful and easier to use. These features support, but are semantically unrelated to the highly interactive



way it does mathematics. Further discussion of *TruCalc*, beyond the scope of the present paper, can be found in [14] and [15].

### 5.1 Ink Editing

In *xThink*, the user writes a formula *then* asks for it to be recognised. In *TruCalc*, the formula being written is *continually* being recognised. This permits a very powerful, and natural, interaction style we call ink editing.

If the user writes ‘−’ it is recognised as a minus sign. If they write 2 above it, the minus sign becomes a division bar. If they cross it out by a vertical stroke, it becomes a + sign.<sup>2</sup> None of these natural ink editing operations makes sense in a batch recogniser.

### 5.2 Dock

*TruCalc* provides a dock, with functionality similar to the dock in Mac OS X. That is, a whole or partial equation can be dragged to the dock, and it will be stored as an item. Conversely, any item in the dock can be clicked on, and it will replace the current equation. If an item is dragged out, it ‘comes out’ as a picture representing its value. Hence an equation such as  $1 + 2 = 3$  might be dragged out of the dock and used, say, as an exponent, as in

$$2^{\boxed{1 + 2 = 3}} = 8$$

(the subequation is boxed, as it cannot be edited except by recalling it from the dock); such dock items can be used in many places in any other equation. The dock serves as a convenient declarative memory for the user.

The dock would be a very natural way to extend *TruCalc* to have variables, at least if entries in the dock could be named. Indeed, dock entries might be associated with URLs, and be able to represent internet resources—such as the current dollar/euro conversion rate, or standard numbers and equations, and so on.

### 5.3 Optionally Hidden Answers

*TruCalc* shows correct answers at all times, just as we have described it. However, for use in teaching, it is possible to hide the answer, and show an empty box. This indicates to a student that their answer is wrong or incomplete, and some correction is still required. Here is an example:

$$2 + \square = 3$$

where normally it would show  $2 + 1 = 3$ .

---

<sup>2</sup> The current implementation of ink editing is not complete; for example you cannot edit − to 4, or edit . to ! in the obvious ways yet.

## 5.4 Undo

*TruCalc* provides the ability to undo edits and alterations by means of a clock metaphor. A user grabs the clock hands and can ‘rewind the time,’ and as they do so the symbols and numbers animate back through time exactly as they were morphed. The morphing provides a temporal continuity between the different steps of the calculation, and it can be played backwards and forwards (i.e., undo and redo).

## 5.5 Possible Extensions to *TruCalc*

*TruCalc* can be extended in many ways. We give a few examples:

1. The dock could be on a web site, and made multiuser so several people can collaborate. The dock could also have a palette of functions (log, sin etc) that, like the current equations, could be dragged into the working equation.
2. The back-end could be replaced with (for example) the *Mathematica* kernel so it was extensible. Currently, *TruCalc* only does complex numerical arithmetic; it could provide an interface to anything *Mathematica* etc can do.
3. Unlike *xThink*, *TruCalc* currently provides no way for a user to write things that are *not* recognised; formulae cannot be annotated, arrows cannot be drawn, and so on. A teacher would probably like another colour which can be used to draw freely with but which *TruCalc* does not interpret.

There are many obvious developments: complete handwriting recognition, to extend *TruCalc* to standard function notation (such as log); restrictions for teaching purposes (*TruCalc* uses complex arithmetic); multiple equations on the screen, like *xThink*. And so on.

However, what *TruCalc* does is show how effective—both reliable and indeed enjoyable (see §6.1)—a user interface for mathematics can be when the interaction, the user interface, itself respects the principles of mathematics.

## 6 Mathematical Mathematical Interfaces Lead into HCI

HCI is the science and art of making user interfaces more effective (and enjoyable) for humans (though HCI techniques have also been used to improve user interfaces for farm animals!).

*TruCalc* allows the user to write an equation  $e$  involving complex numbers from  $\mathbb{C}$  and elementary arithmetic operators. *TruCalc* has no variable names, but uses slots; thus, in conventional terms, the equations can contain variables without repetition—future versions of *TruCalc* may include variable names as they are of course useful for many purposes, not least in providing mnemonics for the slots as currently used.

The variety of solutions  $S$  of  $e$  is intended to be  $S(e, \mathbb{C})$ , except the current version implements  $\mathbb{C}$  by  $\mathbb{C}_J$ , the obvious approximate representation of  $\mathbb{C}$  using pairs of Java double precision floating point numbers.

With these clarifications, we can express some important HCI issues:

1. What should *TruCalc* do when  $S(e, \mathbb{C}_J)$  does not determine a unique solution? Currently *TruCalc* uses heuristics to try to find solutions that are principal values, identities of operators, and so on. For example  $\times = 10$  will be solved by  $10 \times 1 = 10$ , using the right identity of  $\times$ . On the other hand,  $10^{\frac{1}{7}} \times 10^{\frac{1}{7}} = 10$  has no solution as currently implemented, because *TruCalc* effectively tries to solve  $1/x = 0$ .
2. What should *TruCalc* do when  $S(e, \mathbb{C}_J) = \emptyset$ ? *TruCalc*'s solution is to show ? symbols (or  $?+?i$ ); however, an earlier version [13] modified the equation so that at least one solution could be found. Neither solution, we feel, is entirely satisfactory, since  $S(e, \mathbb{C}_J) = \emptyset$  can occur as a transient step in entering a solvable equation—for example, to enter  $1/0.1$  either requires contortions or the intermediate step  $1/0$ .
3. What should *TruCalc* do when there is a *humanly*-obvious algebraic solution, but  $S(e, \mathbb{C}_J) = \emptyset$ ? For example, the very easily entered LHS

$$2^{2^{2^{2^2}}} = ?+?i$$

fails because it is a 19,729 digit decimal number, which is in  $\mathbb{C}$  but not in  $\mathbb{C}_J$ —but the equation could be solved as

$$2^{2^{2^{2^2}}} = 2^{65536}$$

or in many other equivalent symbolic ways. Which is best? Should the user have choices, and if so, how? A symbolic approach would also be a good way to solve equations the user enters containing  $1/0$  terms.

4. Can users choose  $S(e, \mathbb{R}), S(e, \mathbb{Q}), S(e, \mathbb{Z}), S(e, \mathbb{N})$ , for instance for elementary teaching? What about  $S(e, \mathbb{Z}_{12})$  for clock numbers, or  $S(e, F_p)$ , and other interesting domains, say predicate logic or even chess?
5. Improving the handwriting recognition would allow the solution of larger classes of equations, for instance that include transcendental functions.
6. *TruCalc* uses  $=$  as an operator over  $\mathbb{C}_J$ , not  $\mathbb{C}$ . This can result in (apparently) peculiar results such as the following:<sup>3</sup>

$$\begin{aligned} \pi &= 335/113 \\ \pi &= 3.142 \\ 3.142 &= 1571/500 \\ \pi &= 3.142 - 4.073 \times 10^{-4} \end{aligned}$$

Perhaps *TruCalc* should use an operator  $\simeq$  when the equality is approximate? (Although results that are approximate in  $\mathbb{C}_J$  may be exact in  $\mathbb{C}$ !)

---

<sup>3</sup> The last example shows  $4.073 \times 10^{-4}$  which in an earlier version would have been presented in the standard Java format as  $4.073E - 4$ , a ‘buggy’ notation, because a user could not enter  $E$  themselves, so it failed equal opportunity. Here, equal opportunity is seen to be a *generative* design principle: given the existing features, it suggested improvements.

7. *TruCalc* could explicitly show, where it is the case, that numbers are approximate. For example,  $\pi =_{[3]} 3.142$  could be the notation to indicate the equality is correct to three decimal places. If the user changed the subscript 3, they would be changing the precision of the displayed number. Chaitin however suggested that it would be more in keeping with the direct manipulation style of *TruCalc* to allow the user to drag the righthand extension of decimals: so if the user drags the ‘...’ to the right in the equation  $\pi = 3.142\dots$  it could become  $\pi = 3.141592653589793\dots$ ; and dragging the ‘...’ left would put it back to  $\pi = 3.1\dots$ , for example.

In summary, an interesting part of the ‘HCI of *TruCalc*’ can be expressed as the relation between  $S(e, \mathbb{C}_J)$ , the solutions the implementation provides for an equation  $e$ , and  $S(e, \mathbb{H})$ , what the user expects.

## 6.1 Enjoyment

Finally, it surprised us that *TruCalc* was fun to use—we had developed it from principles and had not anticipated the strong feeling of engagement it supports. It integrates body movement, handwriting, and instant *satisfaction*, that children and post-doc mathematicians find exciting. Elsewhere we have reported on our usability surveys, a topic that is beyond the scope of this paper [14]. More recently *TruCalc* was exhibited at the Royal Society Summer Science Exhibition, where it was used by thousands of visitors, children, parents, teachers, to math postdocs. An exit survey was completed by 420 participants (and we insisted that anybody who took a survey form completed it, to avoid under-reporting of negative results) had 90% **liked** or **really liked** *TruCalc*, and nobody (0%) **disliked** it.

## 7 Conclusions

Current leading mathematical systems are capable of a remarkable range of mathematics. With *Mathematica*, a market leading example of an interactive computer algebra system, we are able to solve problems we could not do without it. It is easy to confuse these mathematical capabilities with usability. So much power seems harnessed that the power seems usable.

This ‘power leverage’ blinds us to the fundamental non-mathematical nature of these user interfaces. Often clear mathematical principles like referential transparency and declarativeness are lost in modes, history dependence, context sensitivity, and so on. The failure of these principles in conventional mathematical user interfaces undermines our ability to reason reliably or mathematically.

*xThink* makes use of the affordance of pen and paper to create an interface that solves partially some of the interface issues. But it still ignores basic mathematical principles when applied to interaction. It gains the affordance of paper, at the expense of introducing evaluation modes (and uncertainty in the handwriting recognition).

We have shown in *TruCalc* that it is possible to create an interface that supports basic principles throughout the user interface; it has no hidden state, is modeless, instantly declarative, and so on—or in Frege *et al.*'s metamathematical terms, substitutional, referentially transparent, and so on. Adhering closely to these mathematical principles do not compromise the power of *TruCalc*; it is in principle as powerful mathematically as *xThink* and other conventional systems (though obviously the two systems vary in detail, such as in the choice of built-in functions they support)). Further, we have shown that by supporting these principles that the calculator is easier, more enjoyable, fun and usable—a paradigm shift in usability.

**Acknowledgements.** Harold Thimbleby was supported by a Royal Society-Wolfson Research Merit Award, and Will Thimbleby by a Swansea University studentship. The design of *TruCalc* is covered by patents. Paul Cairns, Greg Chaitin, James McKinna, John Tucker and very many anonymous participants in demonstrations and lectures gave us very useful comments. The Exhibition of *TruCalc* at the Summer Science Exhibition at the Royal Society was funded by EPSRC under grant EP/D029821/1, and Gresham College.

This paper was originally an invited talk at the Mathematical User-Interfaces Workshop 2006 (<http://www.activemath.org/~paul/MathUI06>), but did not appear in the proceedings.

## References

1. Casio, Casio ClassPad 300 Resource Center (2006), <http://www.classpad.org>
2. Garvan, F.: The MAPLE Book. CRC Press, Boca Raton (2001)
3. Goldin, D.Q., Keil, D.: Persistent Turing Machines as a Model of Interactive Computation. *Foundations of Information and Knowledge Systems*, 116–135 (2000)
4. Norman, D.A.: Affordances, Conventions and Design. *Interactions* 6(3), 38–43 (1999)
5. Padovani, L., Solmi, R.: An Investigation on the Dynamics of Direct-Manipulation Editors for Mathematics. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) MKM 2004. LNCS, vol. 3119, pp. 302–316. Springer, Heidelberg (2004)
6. Petkowsk, M., Wilf, H.S., Zeilberger, D.: *A = B*. A K Peters (1996)
7. Quine, W.V.O.: *Word and Object*. MIT Press, Cambridge (1960)
8. Runciman, C., Thimbleby, H.: Equal opportunity interactive systems. *Int. J. Man-Mach. Stud.* 25(4), 439–451 (1986)
9. Tennent, R.D.: *Principles of Programming Languages*. Prentice-Hall, Englewood Cliffs (1981)
10. Théry, L., Bertot, Y., Kahn, G.: Real Theorem Provers Deserve Real User-Interfaces. In: *Proc. Fifth ACM Symposium on Software Development Environments*, pp. 120–129 (1992)
11. Thimbleby, H.: What You See is What You Have Got—A User-Engineering Principle for Manipulative Display? First German ACM Conference on Software Ergonomics. In: *Proc. ACM German Chapter*, vol. 14, pp. 70–84 (1983)
12. Thimbleby, H.: *User Interface Design*. Addison-Wesley, Reading (1990)
13. Thimbleby, H.: A New Calculator and Why it is Necessary. *Computer Journal* 38(6), 418–433 (1996)

14. Thimbleby, W.: A Novel Pen-based Calculator and Its Evaluation. In: Proc. ACM NordiCHI 2004, pp. 445–448 (2004)
15. Thimbleby, W., Thimbleby, H.: A Novel Gesture-Based Calculator and Its Design Principles. In: Proc. BCS HCI Conference, vol. 2, pp. 27–32 (2005)
16. Thimbleby, W., Thimbleby, H.: TruCalc (2006), <http://www.cs.swan.ac.uk/calculators>  
<http://www.cs.swan.ac.uk/calculators>
17. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. In: Proc. London Mathematical Society, Series 2, 42, 230–265 (1936/7) (corrected Series 2, 43, 544–546 (1937))
18. Wolfram, S.: The Mathematica Book, 4th edn., Cambridge (1999)
19. xThink, xThink Calculator (2006), <http://www.xThink.com/Calculator.html>  
<http://www.xThink.com/Calculator.html>

# Coupling Interaction Resources in Ambient Spaces: There Is More Than Meets the Eye!

Nicolas Barralon and Joëlle Coutaz

Université Joseph Fourier  
385 rue de la Bibliothèque, BP 53, 38041 Grenoble Cedex France  
{nicolas.barralon, joelle.coutaz}@imag.fr

**Abstract.** Coupling is the action of binding two entities so that they can operate together to provide new functions. In this article, we propose a formal definition for coupling and present two complementary conceptual tools to reason about coupling interaction resources. The first tool is a graph theoretic and algebraic notation that can be used to identify the consequents of causal couplings so that the side-effects of the creation of a coupling can be analyzed in a formal and systematic way. The second tool formulates the problem of coupling using an 8 state automaton that models the life cycle of a coupling and provides designers with a structure to verify that usability properties have been satisfied for each state. We conclude with the concept of meta-UI, an overarching interactive system that shows that coupling is only one aspect of a larger problem space.

**Keywords:** Ubiquitous computing, ambient intelligence, ambient interactive spaces, devices assembly, devices coupling, meta-UI.

## 1 Introduction

Man is a natural builder. Babies love assembling cubes and objects into complex constructs. TV sets are augmented with high-fidelity loud speakers and wall-size screens to enhance the feeling of “being there”. Computer displays are increasingly augmented with additional external screens and exotic input devices such as iStuffs [1], etc. But as of today, human constructs are elaborated from (and for) two different worlds separated with clear boundaries: the computer world (with millions of PC’s interconnected over the planet) and the physical world (places, artifacts of all sorts, including cars fitted with hundreds of processors, but still insulated from the computer world). Each one of these worlds has its own well-established interaction paradigms and perceived affordances [17], making it easy to couple objects into useful and usable constructs. As we move to ubiquitous computing and ambient interactive spaces, the boundaries disappear, and the story is not as simple.

In his influential paper on ubiquitous computing, Mark Weiser envisioned technologies that “weave themselves into the fabric of everyday life until they are undistinguishable from it” [23]. The PC, as we use it today, will go out of its box, and will be part of the rest of the world. Many scenarios for ambient computing, including those envisioned by Mark Weiser, praise the power that will result from the interaction between “mixed-reality” (or “embodied-virtuality”). However, with this power arise new problems. Among these problems is how to understand coupling.

Recent research in ambient computing demonstrates that coupling opens the way to unbounded forms of interaction and services. For example, one can couple two objects, such as a wallet and home keys, by shaking them together [10]. As a result an alarm can signal when one is separated from the other. This way, the owner is less likely to forget one or the other along the way. But, how do we know that the keys can be coupled with (and decoupled from) the wallet? How do we know that they can be coupled by shaking them altogether? What should happen when the keys are coupled with a pair of shoes, are then the shoes coupled with the wallet?

This article is a scientific essay on coupling entities with special attention to entities that play the role of interaction resources. In the context of this work, an entity may be physical (denoted P), digital (or numerical, N), or mixed (M). A table, a keyboard are P entities; a keyboard driver is an N entity, a finger tracker is an N entity as well. A mixed entity results from coupling N and P entities. An M entity plays the role of an interaction resource when it allows users to exchange information with a computer system.

This article is structured in the following way: in the next section, we provide a formal definition for the notion of coupling illustrated with two systems serving as running examples. We then build upon analogies with chemistry in the following way: in Section 3, we define the valence of an entity and refer to the compatibility between entities. Then in Section 4, we propose to model mixed entities as N-P molecules, and in Sections 5 and 6, we reason on causal relationships between couplings using formal notations. In Section 7, we detail the life cycle of a coupling then show, in Section 8, how it can serve as a framework for usability investigation. In the last section, we show how coupling interaction resources is only one facet of a more general problem: that of providing end-users with a meta-UI to build, control, evaluate, and ultimately program their interactive space.

## 2 Coupling Entities

### 2.1 Definition

The word “coupling” may be used to denote an act, or the result of this act.

- As an act, *coupling* is the action of binding two entities so that they operate jointly to provide a set of functions that these entities cannot provide individually.
- As the result of an act, *a coupling* is an assembly of the source entities, that is, a new compound entity that provides a new set of functions that the source entities, alone, cannot provide.

In both cases, given two entities, the nature of the act determines the resulting set of functions. For example, in Hinckley’s dynamic display tiling [9], users obtain different functions depending on the way the tablets are bumped together: if one tablet rests flat on a desk surface, and a second tablet is bumped into the base tablet, then the resulting function is the creation of a larger display. Alternatively, if the two tablets are bumped symmetrically, the same content is replicated on both displays to support face-to-face collaboration.



We express the twofold acceptance of coupling (either as an act, or as an entity) in the following formal way. Let:

- $\mathbf{E}$  be a non-empty finite set of entities and  $\mathbf{F}$ , the set of functions that these entities provide individually,
- $func$ , the function that returns the set of functions  $f$  ( $f \in \mathbf{F}$ ) that an entity  $e \in \mathbf{E}$  provides:  $f = func(e)$ ,
- $\mathbf{A}$ , a non-empty set of sequences of actions  $a$ ,
- $\mathbf{C}$ , the set of couplings between entities belonging to  $\mathbf{E}$ , using sequences of actions  $a \in \mathbf{A}$ ,
- $e \in \mathbf{E}$ , the compound entity that results from binding  $e_1$  and  $e_2$  by the way of the sequence of actions  $a \in \mathbf{A}$ ,

then, the coupling  $c$  ( $c \in \mathbf{C}$ ) is defined as the Cartesian product  $\mathbf{E} \times \mathbf{E} \times \mathbf{A}$  in  $\mathbf{E}$ :

$$c : \mathbf{E} \times \mathbf{E} \times \mathbf{A} \rightarrow \mathbf{E}$$

and is denoted as:

$$c = (e_1, e_2, e) : \forall f_i \neq f_1 \wedge f_i \neq f_2 : (f_1 \cap f_i = \emptyset) \wedge (f_2 \cap f_i = \emptyset) \quad (1)$$

where  $e_1, e_2 \in \mathbf{E}$ ,  $f_1 = func(e_1)$ ,  $f_2 = func(e_2)$ ,  $f = func(e)$

or as:

$$c = (e_1, e_2, f) \quad (2)$$

or as:

$$(e_1, c, e_2) \quad \text{or} \quad e_1 \xrightarrow{c} e_2 \quad (3)$$

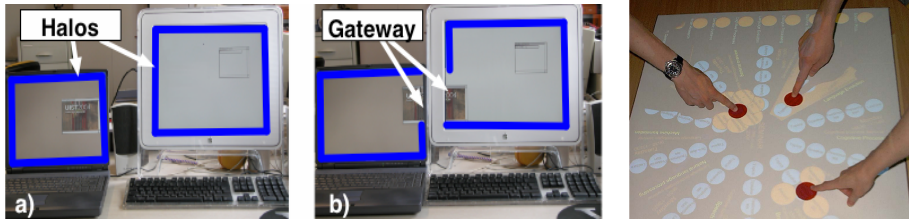
In notation (1), the focus of attention is the new compound entity obtained by the way of coupling. Notation (2) stresses the importance of the resulting set of functions while maintaining an explicit reference to the source entities. Notations 3 make the bond between the source entities explicit. Fig. 1 illustrates couplings that we will use as running examples in the following discussion.

## 2.2 Illustration

I-AM (Interaction Abstract Machine) is a middleware that supports the dynamic coupling of entities such as screens, keyboards and mice, to form a unified interactive space [15]. These entities may be distributed across multiple machines running distinct operating systems including MacOS X and Windows XP. In this space, users can distribute and migrate windows as if they were handled by a single computer<sup>1</sup>. The two screen entities of Fig. 1-a are coupled when the sequence of actions  $a$  that bring the screens in close contact is performed (detected by infrared-based proximity sensors). This sequence of actions is similar in spirit to Hinckley's synchronous gestures [9]. An alternative sequence of actions, inspired from SyncTap [18] called "Click'n Couple", consists in bringing the cursors of the mice face to face, and then click the mouse buttons simultaneously (i.e. within the same temporal window). The function  $f$

<sup>1</sup> The illusion of a unified space is provided at no extra cost for the developer who can reuse the conventional GUI programming paradigm. I-AM is similar in spirit to iRoom and i-LAND. Although iRoom supports heterogeneous workstations, windows in iRoom cannot cross screens boundaries. In i-LAND, windows can cross screens boundaries but the underlying workstations run the same operating system.

now available is an affine transform that supports different screen resolution and orientation, as well as bezels thickness so that windows and figures can overlap multiple screens without any deformation (See [15] for details).



**Fig. 1.** (a) The PC and the Macintosh screens are decoupled and run two applications. (b) The two screens are coupled to form a single display area by bringing them in close contact. (Halos have been artificially enhanced on the pictures to increase readability.) (c) Partial view of the FAME room. Selectable information ( $N$  entities) is rendered as round shape items that match the shape of the physical tokens (form factor compatibility between the  $N$ 's and  $P$ 's). A flower menu is obtained by placing a token on a round-shape  $N$  item. Here, users have opened three "flower menus".

The FAME table (see Fig. 1-c) is a component of an augmented room that supports the collaborative exploration of information spaces [11]. A table and two walls play the role of output interaction resources. Each one is coupled to its own video-projector to display digital information ( $N$  entities). In addition, the table is coupled to a camera that senses colored, 4 cm wide round shape tokens made of plastic. A token (a  $P$  entity) is coupled to the tracker of the table (an  $N$  entity), when the action  $a$  "put token down on the table" is performed. The coupling "token-tracker" results in an  $M$  entity that plays the role of an input interaction resource. This  $M$  entity is coupled with a round shape digital entity displayed on the table when the token (i.e. the  $P$  component of  $M$ ) is brought over the entity. A "flower menu" pops up around the token to inform the user that the function  $f$  "information selection" is now available. The user can now select digital information by moving the token to the appropriate petal of the flower.

Our definition, which involves two source entities, does not imply that coupling is exclusive. An entity may be coupled to several other entities. The possible configurations that can result depend on the valence and the compatibility of the entities involved. These are discussed next.

### 3 Valence of an Entity and Compatibility between Entities

The *valence* of an entity is an integer that measures the maximum number of entities that can be bound with it at a given time. For example, in I-AM as well as for Hinckley's tablets, the valence of a screen is 4: a screen can be coupled to a maximum of 4 screens (one on each side). The valence of a FAME token is 2: at a given time, it can be coupled to at most 1 table and 1 item of digital information.

Compatibility has been used in many ways in HCI to motivate design decisions [4, 12, 14, 24]. Here, the *compatibility* between two entities denotes the capacity for these entities to be coupled provided that they satisfy a set of constraints that apply to both of them. Constraints may apply to:

- Physical form factors. In I-AM, surfaces that can be coupled must be rectangular. In FAME, tokens must be round and red in order to be tracked by the system.
- Software discoverability and interoperability. In I-AM, MacOS and/or Windows platforms are compatible, but Linux is not supported.
- Cognitive compatibility at multiple levels of abstraction from physical actions to intentions and goals. In FAME, selectable information is rendered as round shape items that match the shape of the physical tokens (to enforce their perceived affordance) (see Fig. 1-c). In its current implementation, the FAME tracker is able to track about 12 tokens simultaneously with an 80ms latency on a dual PowerPC 7400 (G4) 1.4 Ghz machine. As a result, if more that 12 tokens are coupled with the table, then the latency of the system is not sufficient to support the feeling of tightly coupled interaction [3]. Adding a 13<sup>th</sup> token is technically feasible, but not compatible with human expectation at the physical action level.
- Contextual compatibility. The context in which the coupling of entities is created and evolves can influence their compatibility. In this article, we focus on coupling under the control of the user. Dobson and Nixon, in [7], provide approaches for adapting a system according to the context of use. Their approach can be applied to our problem, where compatibility between entities depends on context.

Valence and compatibility between entities determine conditions for the realization of couplings. In the next section, we illustrate the use of these characteristics for the construction of mixed entities.

## 4 Mixed Entities as N-P Molecules

P's and N's can be coupled in a number of ways to form new mixed entities. In particular, two basic mixed entities, denoted respectively as **P-N** and *P-N*, can be coupled by the way of their N component, or their P component, or by a mix of them. As shown in Fig. 2, one may obtain the following configurations: **N-P-P-N**, **P-N-N-P**, **N-P-N-P**, and **P-N-P-N**. Are all of them possible? The answer depends on the valence of the components and the compatibility between them.



Fig. 2. Basic N-P constructs

By analogy with chemistry, mixed entities are *N-P molecules* elaborated from any number of N and P atoms whose configuration satisfies their valence and compatibility. Intuitively, form factors matter in N-P-P-N configurations, whereas software compatibility prevails in P-N-N-P constructs. For example, in I-AM, only rectangular

screens can be coupled. In any case, the resulting assembly must be cognitively compatible with user's physical abilities and expectation. The assembly of N-P molecules may be performed either at design time, or at run time. We believe that the design/run time distinction is important in the context of ambient computing where dynamic reconfiguration under human control is key.

*Intrinsically-mixed* entities are those for which the coupling of numerical and physical entities has been performed before hand by designers so that end-users can exploit them directly without performing any additional binding. For example, a PDA is an intrinsically-mixed entity: it binds together digital and physical components that have been pre-packaged into a working unit.

Alternatively, entities are *constructively-mixed* when the end-user is in charge of performing some coupling before being able to use them. A FAME token must be coupled to the table in order to be used as a pointing device. Thus in FAME, pointing devices are constructively-mixed entities. Similarly, an external keyboard, which is a physical entity, needs to be coupled with a driver to play the role of an input interaction resource. Clearly, constructively-mixed entities can include entities that are intrinsically-mixed. The Nabaztag shown in Fig. 3, is an example of this type of assembly.

The Nabaztag (which means "rabbit" in Armenian) is an intrinsically-mixed entity built from a 9 inches tall plastic bunny shape object with a loud-speaker, moving ears, and colored lights that pulsate on its nose and belly. It includes a Wi-Fi connection to the Internet so that it can be coupled to Internet services such as the weather forecast, inter-personal messaging, and mood expression (the rabbit has a mood!). Using a Web server on a PC, users can couple any number of N entities (i.e. Web services) to the N component of the Nabaztag provided that these services interoperate with the N component. The result is a well-balanced star-like N composition coupled to a single P.

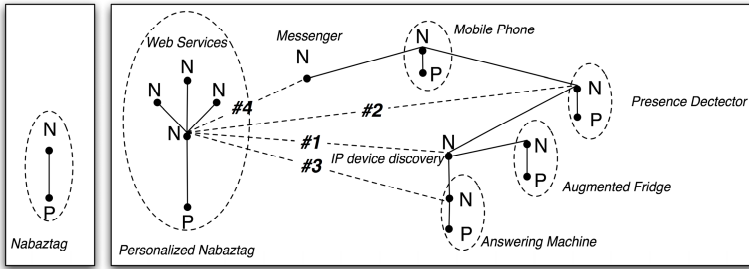
However, one may wonder how a single P can (simultaneously) render the state of a large number of N services and allow users to manipulate this state through a limited number of input means (i.e., the ears of the plastic rabbit and a push button located at the top of its head). One possible venue is for the Nabaztag to borrow interaction resources of the interactive space by the way of causal couplings.

## 5 Causal Couplings and Their Consequents: A Formal Analysis

As in chemistry, couplings may have *causal relationships*: coupling an entity with a compound entity may entail a chain of reactions: some bonds may be destroyed, possibly giving rise to multiple entities. Alternatively, additional couplings may be created as *consequents* of the *causal* coupling. In the following discussion, we use the Nabaztag as an informal illustration of the problem followed by two formal notations to reason about causal couplings and their consequents.

### 5.1 Illustration

Fig. 3 illustrates causal relationships between couplings when the Nabaztag is coupled with a smart home. The N components of this smart home include a presence detector, a surveillance system, and an IP-device discovery facility. It includes a number of intrinsically-mixed entities such as an augmented IP fridge and an IP answering machine. When the owner is away, any intrusion or abnormal situation is notified to the owner via the mobile phone.



**Fig. 3.** On the left, the original off-the shelf Nabaztag is an intrinsically-mixed entity. On the right, the personalized Nabaztag used in a smart home becomes a constructively-mixed entity. Couplings #2, #3, and #4 are the consequents of the causal Coupling #1.

The Nabaztag plays the messages sent by buddies using its speaker-phone, but it is unable to remember them. Thus, when there is nobody at home, one would like the Nabaztag to forward incoming messages to the recording facility of the answering machine and/or to the mobile phone. Because, the Nabaztag is an IP device, it can be detected automatically by the IP-device discovery facility resulting in the creation of Coupling#1. In turn, Coupling#1 entails three consequents (Couplings#2, #3 and #4) in order to provide the forward-to service: Coupling#2 to determine whether there is somebody at home, Coupling#3 to use the recording facility of the answering machine, and Coupling#4 to forward messages to the mobile phone.

Coupling the Nabaztag to the smart home raises a number of issues, in particular: what consequent couplings of a causal coupling make sense? The following formal analysis provides a systematic framework for answering this question using a graph theoretic notation and an algebraic notation.

## 5.2 Formal Analysis with a Graph Theoretic Notation

We represent couplings using the graph notation (3) introduced in 2.1 where nodes denote entities, and where edges express the existence of couplings. Symbols "\*" and "=" denote causal and consequent couplings respectively. A coupling is *causal* when its creation implies, as a side effect, the creation of additional couplings. These additional couplings are called consequent couplings or simply, *consequents*. The "?" symbol denotes the couplings that are under evaluation (i.e. keeping them as consequents or rejecting them has yet not been decided). To express their transitory state, causal couplings, as well as consequents and undecided couplings are represented as dotted edges. Let:

- $EDGE$  be the set of edges of the graph under consideration.
- $r_1(c)$  (resp.  $r_2(c)$ ) be the first (resp. the second) interaction resource involved in the coupling ( $r1, c, r2$ ).
- $F(r_1, r_2)$  be the set of function that result from ( $r1, c, r2$ ).
- $Compatible(f_1, f_2, f_3)$  returns TRUE if the functions  $f_1$  and  $f_2$  allow the existence of the function  $f_3$ . To be TRUE,  $Compatible(f_1, f_2, f_3)$  may require the suppression of existing couplings. Although important (and challenging), this possibility is not addressed in this article.

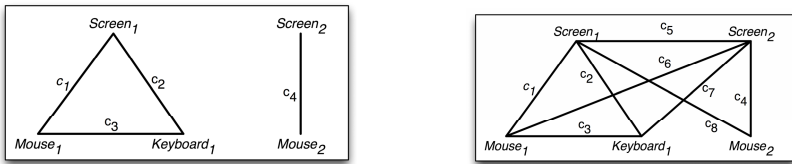
The principle of our algorithm is the following: consider every new edge that results from the transitive closure with paths of length 2 that contain both  $r_1(c)$  and  $r_2(c)$ . If this new edge corresponds to the creation of a coupling whose function is compatible with the functions provided by its neighboring edges, then it is created. In turn, the coupling that this edge denotes becomes a causal coupling and the algorithm is applied again. More formally:

```

For every causal coupling c
  Build the set of nodes Nc such that :
    n ∈ Nc ⇔ n ∈ path ∧ length(path) = 2
              ∧ r1(c) ∈ path ∧ r2(c) ∈ path

  For all n ∈ Nc and n ≠ r1(c) and n ≠ r2(c)
    if edge(n, r1(c)) ∈ EDGE
      if compatible(F(c), F(n, r1(c)), F(n, r2(c))) then
        EDGE = EDGE ∪ new edge(r2(c), n)
    else
      if compatible(F(c), F(n, r2(c)), F(n, r1(c))) then
        EDGE = EDGE ∪ new edge(r1(c), s)
    
```

To illustrate the algorithm, let's consider the initial configuration of couplings represented in Fig. 4: on the left image, *Screen1* is coupled with *Mouse1* and *Keyboard1*, and *Mouse1* is coupled with *Keyboard1* to provide *Keyboard1* with the input focus function. This configuration corresponds to a private workstation. On the right, a public *Screen2* is coupled with a public pointing device *Mouse2*. Because *Screen1* and *Screen2* are compatible by design (resulting in the enlarged display function),  $c_5$  is performed (for example, by a proximity detection service).



**Fig. 4.** Left image: the initial configuration that represents a private workstation. Right image: final configuration that corresponds to a public configuration where couplings  $c_6$ ,  $c_7$  and  $c_8$  are the consequents of the causal coupling  $c_5$ .

The final configuration that results from the causal coupling  $c_5$  is shown by the rightmost graph of Fig. 4: the owner of the private workstation can manipulate digital information displayed on *Screen2* and *Screen1* using the private interaction resources *Mouse1* and *Keyboard1*. In addition, information can be designated on both screens with *Mouse2*, but for privacy reason, *Mouse2* cannot be coupled to *Keyboard1*. In a different situation where the workstations would be owned by two distinct users who want to collaborate via a unified space, the compatibility functions would be different resulting in a distinct final configuration.

Fig. 5 shows the successive steps that lead to the final configuration of Fig. 4. Fig. 5 (top left) corresponds to the generations of  $c_6$  and  $c_7$  that result from the transitive closure with paths  $c_5-c_1$  and  $c_5-c_2$  respectively. Fig. 5 (top right) shows the

generation of  $c_8$  that results from the transitive closure with path  $c_5-c_4$ . Because the function that results from  $c_6$  is compatible with that of  $c_1$  and  $c_5$ ,  $c_6$  is created. The same holds for  $c_7$  and  $c_8$  whose resulting functions are compatible with that of  $c_5$  and  $c_2$ , and  $c_5$  and  $c_4$  respectively.  $c_6$ ,  $c_7$  and  $c_8$  are now causal couplings. Fig. 5 (bottom left) corresponds to the application of the algorithm to  $c_6$  with the evaluation of  $c'_6$  that results from the transitive closure with paths  $c_6-c_4$ . The function that results from  $c'_6$  is not compatible with that of  $c_6$  and  $c_4$  (coupling a private mouse with a public mouse to access any display area is considered as inappropriate for this particular situation). The same holds for  $c'_7$  and  $c'_8$  that result from the transitive closure with paths  $c_4-c_7$  and  $c_8-c_2$  respectively. For this particular situation, *Mouse2* cannot serve as input focus for *Keyboard1* (Fig. 5, bottom center and bottom right). To summarize, the causal coupling  $c_5$  has three consequents:  $c_6$ ,  $c_7$ , and  $c_8$ .

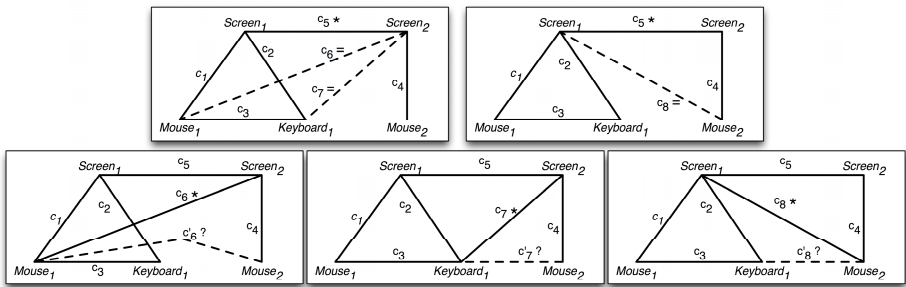


Fig. 5. Evaluation steps resulting from the causal coupling  $c_5$

### 5.3 Formal Analysis with an Algebraic Notation

Our algebraic notation is based on two operators over couplings:

- The generation operator:  $*$
- The union operator:  $+$
- $*$  is distributive over  $+$
- The priority of  $*$  is superior to that of  $+$ .

Then, the expression  $[c_1 + c_2 + \dots + c_n]$  denotes the set of couplings  $c_1, c_2, \dots, c_n$  that exist within a particular system. The creation of a new coupling  $c_p$  is a two-step process:

- First,  $c_p$  is added to the set by applying the *insertion rule* such that:  

$$c_p + [c_1 + c_2 + \dots + c_n] = [c_p + c_1 + c_2 + \dots + c_n]$$
- Then, the consequents of  $c_p$  are computed using *the generation rule* such that:  

$$c_p * [c_p + c_1 + c_2 + \dots + c_n] = c_p * c_p + c_p * c_1 + c_p * c_2 + \dots + c_p * c_n$$

Evaluating  $c_p * c_p + c_p * c_1 + c_p * c_2 + \dots + c_p * c_n$  is to evaluate each one of the terms  $c_p * c_i$ :

- $c_p * c_p = \emptyset$  (a coupling cannot be the consequent of itself).
- $c_p * c_i = (r_{p1}, r_{p2}, f_p) * (r_{i1}, r_{i2}, f_i)$

$c_p$  and  $c_i$  are *transitive* if and only if (iif)

$$(r_{p1}=r_{i1} \vee r_{p1}=r_{i2} \vee r_{p2}=r_{i1} \vee r_{p2}=r_{i2}) \wedge \text{Compatible}(f_p, f_i, f) \wedge c_p \neq c_i$$

otherwise,  $c_p$  and  $c_i$  are *intransitive*.

The condition  $(r_{p1}=r_{i1} \vee r_{p1}=r_{i2} \vee r_{p2}=r_{i1} \vee r_{p2}=r_{i2})$  expresses the fact that transitive couplings share one interaction resource. This is equivalent in the graph notation to the paths of length 2 that contain  $r_1(c_p)$  and  $r_2(c_p)$ .

If  $c_p$  and  $c_i$  are transitive then:

$$c_p * c_i = (r_{p1}, r_{p2}, f_p) * (r_{i1}, r_{i2}, f_i) = c_{res}$$

where  $\text{Compatible}(f_p, f_i, f) = \text{TRUE}$

$$c_{res} = (r_{p2}, r_{i2}, f) \text{ if } r_{p1}=r_{i1}$$

$$c_{res} = (r_{p2}, r_{i1}, f) \text{ if } r_{p1}=r_{i2}$$

$$c_{res} = (r_{p1}, r_{i2}, f) \text{ if } r_{p2}=r_{i1}$$

$$c_{res} = (r_{p1}, r_{i1}, f) \text{ if } r_{p2}=r_{i2}$$

$$\text{Transitive}(c_p, c_i) = \text{TRUE}$$

If  $c_p$  and  $c_i$  are intransitive then:

$$c_p * c_i = (r_{p1}, r_{p2}, f_p) * (r_{i1}, r_{i2}, f_i) = \emptyset$$

$$\text{Transitive}(c_p, c_i) = \text{FALSE}$$

The algorithm detailed in Fig. 6, makes it explicit the generation of the consequents of a causal coupling. Let's apply the algorithm to the example of Fig. 4:

Initial configuration :  $[c_1 + c_2 + c_3 + c_4]$

Causal coupling :  $c_5$

Insertion rule:

$$c_5 + [c_1 + c_2 + c_3 + c_4] = [c_5 + c_1 + c_2 + c_3 + c_4]$$

Generation rule:

$$c_5 * [c_5 + c_1 + c_2 + c_3 + c_4] = c_5 * c_5 + c_5 * c_1 + c_5 * c_2 + c_5 * c_3 + c_5 * c_4$$

$$= \emptyset + c_6 + c_7 + \emptyset + c_8$$

Insertion rule:

$$c_6 + c_7 + c_8 + [c_5 + c_1 + c_2 + c_3 + c_4] = [c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8]$$

Generation rule:

$$(c_6 + c_7 + c_8) * [c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8] =$$

1.  $c_6 * c_1 + c_6 * c_2 + c_6 * c_3 + c_6 * c_4 + c_6 * c_5 + c_6 * c_6 + c_6 * c_7 + c_6 * c_8$
  2.  $+ c_7 * c_1 + c_7 * c_2 + c_7 * c_3 + c_7 * c_4 + c_7 * c_5 + c_7 * c_6 + c_7 * c_7 + c_7 * c_8$
  3.  $+ c_8 * c_1 + c_8 * c_2 + c_8 * c_3 + c_8 * c_4 + c_8 * c_5 + c_8 * c_6 + c_8 * c_7 + c_8 * c_8$
1.  $\emptyset + \emptyset + \emptyset + \emptyset + \emptyset + \emptyset + \emptyset + \emptyset$  (couplings are intransitive)
  2.  $+ \emptyset + \emptyset + \emptyset + \emptyset + \emptyset + \emptyset + \emptyset + \emptyset$  (couplings are intransitive)
  3.  $+ \emptyset + \emptyset + \emptyset + \emptyset + \emptyset + \emptyset + \emptyset + \emptyset$  (couplings are intransitive)



```

Function []Coupling GenerationRule(
    //causal coupling provoking the generation of consequents
    Coupling causalCoupling,
    //set of effective couplings
    Coupling []effectiveCouplings){

    //insertion of the causal coupling to effectiveCouplings
    Coupling []couplings = {effectiveCouplings, causalCoupling } ;

    //init workingList, the list of couplings onto which
    //the generation rule must be applied
    Couplage []workingList = {causalCoupling } ;

    //traverse the workingList for possible generation
    For (int i=0 ; i< workingList.length ; i++){
        //get item of the list
        Coupling c1 = workingList[i] ;

        //traverse the effectiveCouplings
        For (int j=0 ; j<effectiveCouplings.length ; j++){
            //get item of the list
            Coupling c2 = effectiveCouplings [j] ;

            //test the transitivity between c1 and c2
            If (Transitive(c1, c2)){
                //generate a new coupling
                Coupling gen = composition(c1, c2) ;

                //insert new coupling to the effectiveCouplings
                effectiveCouplings [effectiveCouplings.length]= gen ;

                //the generation rule applies to the new created coupling
                workingList [workingList.length]= gen ;
            }
        }
    }
    Return effectiveCouplings;
}

```

**Fig. 6.** Generation of the consequents of a causal coupling

Thus, given the compatibility rules elicited for that particular situation, the final configuration that results from the causal coupling  $c_5$  is:  $[c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7 + c_8]$ . Because compatibility evolves over time, the life cycle of a coupling cannot be a simplistic dual state (coupled/uncoupled) Finite State Automaton (FSA). This aspect is discussed next.

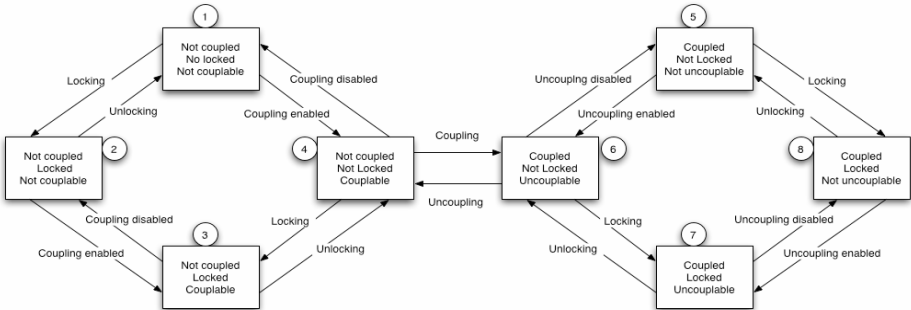
## 6 Life Cycle of Couplings

As shown in Fig. 7, the life cycle of a coupling includes eight states where a state is defined by the conjunction of the following set of predicates:

- *Coupled* ( $e1, c, e2$ ) = TRUE if and only if  $f \neq \emptyset$  where  $f$  is the set of functions that results from  $(e1, c, e2)$ . If  $f = \emptyset$ , then *Coupled* ( $e1, c, e2$ ) = FALSE and *NotCoupled* ( $e1, c, e2$ ) = TRUE.
- *Locked* ( $e1, c, e2$ ) = TRUE if the state of  $e1$  does not permit to modify the state of  $(e1, c, e2)$ . This predicate can be used to express that  $e1$  is not “socially” or “technically”

available to enter or exit its coupling  $c$  with  $e2$ . For example, a user does not want to connect his private PDA to a public screen. The state of  $(e1, c, e2)$  is kept unchanged until  $Locked(e1, c, e2) = FALSE$  or  $NotLocked(e1, c, e2) = TRUE$ .

- *Couplable*  $(e1, c, e2)$  is an expression of predicates  $P$ , where  $P \neq Coupled(e1, c, e2)$  and  $P \neq Locked(e1, c, e2)$ . This expression specifies the conditions (different from  $Coupled(e1, c, e2)$  and  $Locked(e1, c, e2)$ ) that are necessary for  $(e1, c, e2)$  to happen. For example, valence and compatibility can be used to express *Couplable*. Symmetrically, *Uncouplable* expresses the conditions (different from  $Coupled(e1, c, e2)$  and  $Locked(e1, c, e2)$ ) that are necessary for  $(e1, c, e2)$  to end.



**Fig. 7.** Coupling  $(e1, c, e2)$  as a Finite State Automaton. For the sake of readability, the transitions between states 1 and 3, 2 and 4, 5 and 7, 6 and 8 are not represented.

The automaton shown in Fig. 7 corresponds to the coupling  $(e1, c, e2)^2$ . It is comprised of two sub-automata: one that includes the states 1, 2, 3, 4 where  $Coupled(e1, c, e2)$  is TRUE, the other one that covers the states 5, 6, 7, 8 where  $Coupled(e1, c, e2)$  is FALSE. States 4 and 6 serve as gateways between the two sub-automata. State 4 corresponds to the situation where all the conditions for realizing  $(e1, c, e2)$  are satisfied. Only a coupling request event is missing to enter state 6.

Because of the multitude of states, the study of such automata provides fertile ground for usability investigation.

## 7 The Life Cycle as an Analytic Framework for Usability

As an illustration, we analyze I-AM and the FAME table with two of the IFIP properties: observability and predictability [8]. Other usability frameworks (such as the Cognitive Walkthrough [22], Nielsen’s [16] or Bastien-Scapin’s criteria [2]) could be used as well.

### 7.1 Observability of Couplings in the FAME Table

Observability is the ability for the user to evaluate the internal state of the system from its perceivable representation. When applied to the life cycle of a coupling, this property requires that every state of the automaton be made observable to users.

<sup>2</sup> A similar automaton models  $(e2, c, e1)$ .

As a counter-example, let's consider the coupling of the FAME tokens with the N entities displayed on the table. Let  $t_1$  and  $t_2$  be two tokens, and  $i_1$ , a selectable N item projected on the table. At the beginning, the user is holding the tokens in his hands, and  $i_1$  is rendered as a round shape graphics. Thus,  $(t_1, c, i_1)$  is in State 4. By dropping  $t_1$  on  $i_1$ , one couples  $t_1$  with  $i_1$  making the select function available: the automaton for  $(t_1, c, i_1)$  enters State 6. To make this state observable,  $i_1$  opens itself as a flower where each petal is couplable to  $t_1$ . On the other hand, this action locks  $i_1$  for tokens different from  $t_1$ :  $(t_1, c, i_1)$  then enters State 7. As a result, dropping  $t_2$  on any petal of  $i_1$  will have no effect (since  $(t_1, c, i_1)$  is locked) but dropping  $t_2$  on another selectable item  $i_2$  would work correctly.

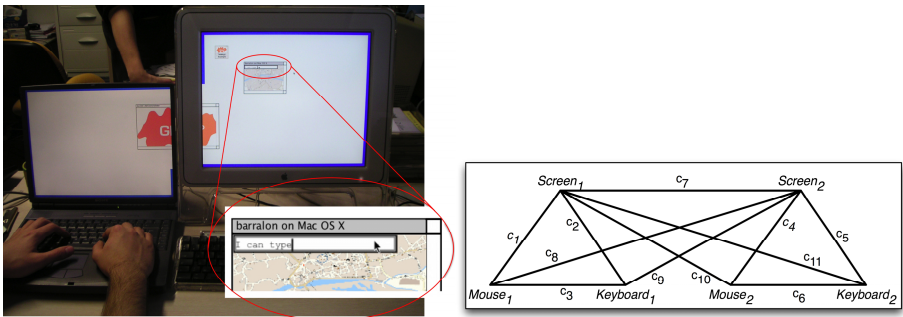
Two semi-formal user studies with 30 subjects unfamiliar with the FAME table showed that some people selected the petals using additional tokens instead of traversing the flower menu with the coupled token. If we had this analytical model at the time of the development of FAME, we would have been able to spot this problem and take corrective actions such as making the *Locked* state observable or allowing coupling a flower menu with multiple tokens.

## 7.2 Observability of Couplings in I-AM

In Fig. 1-a, two applications are running on two independent workstations. The closed blue halo that outlines each screen denotes the possibility for currently uncoupled screens to be coupled (State 4 is made observable). The absence of halos would mean that the screens are not couplable. On the other hand, the distinction between States 1 or 2 (locked/unlocked) is not observable which may cause a problem in a collaborative situation. As shown in Fig. 1-b, once the screens are coupled, the new shape of the halo indicates the gateway through which windows can migrate between the two screens (State 6 is made observable).

## 7.3 Predictability of Couplings in I-AM

Predictability is the ability for the user to determine the effect of future actions based on past interaction history. Applied to coupling, users should be able to anticipate the set of functions  $f$  that will result from the set of actions  $a$ .



**Fig. 8.** Entering characters in a text field located on a Macintosh screen using a PC keyboard: to do so, the user has selected the text field with the PC touchpad (left). The corresponding configuration (right) that results from the causal coupling  $c_7$  between the two screens.

I-AM preserves the conventions of the GUI paradigm. Windows can sit between two coupled screens although these screens may be connected to different workstations and may differ in resolution and orientation. Mice and keyboards are coupled to provide the input focus function. But, can users predict the final configuration shown in Fig. 8 (right) that results from coupling the two screens? This configuration expresses the capacity for any interactor displayed on the unified surface to be coupled to the mouse-keyboard of the Macintosh as well as to the mouse-keyboard of the PC.

Suppose that the user has selected the input text field displayed on the Macintosh screen using the mouse-keyboard of the Mac. The user can then enter text with the Macintosh keyboard. So far, the system behavior is compliant with GUI conventions: in this regard, predictability is satisfied. On the other hand, can the user predict the situation depicted in Fig. 8 (left): the input text field is coupled to the mouse-keyboard of the Macintosh as well as to the mouse-keyboard of the PC (as a result of a PC mouse click in the text field). In this situation, characters can be entered simultaneously from any keyboard. What will happen when the screens are decoupled? This is where things get complex with regard to predictability even in simple situations like the one described below.

Let S1 be a screen coupled by construction (i.e. GUI conventions legacy) to a PC workstation and a mouse M. Let S2 be a screen connected to a second computer with no input device. S1 is now coupled to S2 by bringing S1 and S2 close to each other. According to the I-AM model, M can get coupled to S2 as well: it can be used to modify the information space mapped on S2. Thus the cursor of M can be mapped on S2. Can the user predict what will happen if S1 is uncoupled from S2 while the cursor of M is mapped on S2? Will M be uncoupled from S1 and stay coupled with S2? Or, alternatively, will it follow its home surface? If so, where will the cursor re-appear on S1? This type of problem was spotted by the developers of PointRight [13] who stated that “a free-space device (such as a wireless mouse) needs an explicit starting screen”. Translated into our framework, this means that when a wireless mouse is dynamically coupled to the interactive space, its associated cursor must be mapped onto a predefined home screen in order to support predictability.

As this example shows, by transitivity, multiple entities are bound together to form an interactive space whose functionalities depend on the set of functions that each coupling delivers. Do these functions, all together, form a “consistent story” for the user? Since the management of the interactive space corresponds to the interplay of multiple automata, how many of them can the system (and the user) reasonably handle at a time? How can this be controlled by end-users? We propose the concept of meta-UI as a coherent framework to address these issues.

## 8 The Concept of Meta-UI

A *meta-UI* is an interactive system whose set of functions is necessary and sufficient for end-users to control and evaluate the state of an ambient space. This set is *meta-* because it serves as an umbrella *beyond* the domain-dependent services that support human activities in this space. It is *UI-oriented* because its role is to allow users to control and evaluate the state of the ambient space. In the context of this article, a meta-UI *is not* an abstract model, nor a language description, whose transformation/interpretation would

produce a concrete effective UI. It is an over-arching interactive system whose role is to ambient computing what desktops and shells are to conventional workstations.

The notion of meta-UI is described in detail in [5]. As summarized in Fig. 11, a meta-UI is characterized by its *functional coverage* in terms of *services* (including coupling), and *object types* (including mixed entities). In turn, the services and objects are invoked and referenced by the way of *interaction techniques* (or UI) that provide users with some *level of control*: who has the initiative (users or the system?), and once a service is launched what kind of control do users have (observability only, traceability only, or dynamic and incremental control?).

An interaction technique is a language (possibly *extensible*) characterized by the *representation* (vocabulary) used to denote objects and services as well as by the way users can construct sentences and assemble these sentences into programs. Given the role of a meta-UI, the elements of the interaction technique of the meta-UI must co-habit with the UI's of the domain-dependent services that it governs: these elements may be embedded within the UI of the domain-dependent services, or they may be external to the UI of these services. Using the Nabaztag and smart home example, we illustrate the concept of meta-UI for coupling.

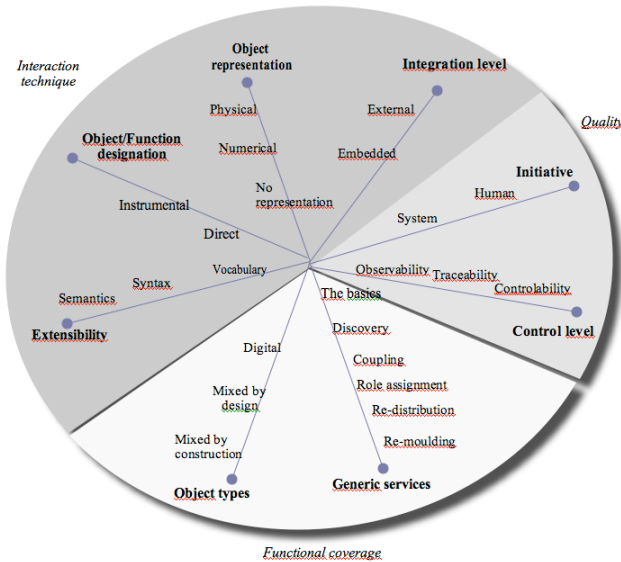


Fig. 11. The dimension space of Meta-UI's

Forwarding messages to the answering machine or to the distant SMS, may be pre-programmed within the N component of the Nabaztag or the Nabaztag may not hold this program at all. In the first case, the program may be triggered when Coupling#1 is performed. As mentioned above, this coupling (and its consequents) may be performed on the system initiative, and pursued autonomously with no human control. Alternatively, the user may be kept in the loop: from implicit, the process becomes explicit. The level of control that end-users have on couplings is fundamental. At

minimum, *observability* should be supported, i.e. users should be able to evaluate the internal state of the coupling from its current perceivable representation. The next step is *traceability* by which users can observe the evolution of the coupling over time, but they cannot modify this evolution. With *controllability*, users can observe, trace, and intervene on the evolution of couplings. They can even program couplings.

For example, if the Nabaztag does not host the “forward-to” program, the smart home may include an *end-user development environment* (EUDE) that would allow users to build programs, i.e. new N entities, to modify the behavior of the smart home. Powerful meta-UI’s must include an EUDE. Based on visual programming, tools like Jigsaw support the construction of simple sentences such as “if someone rings the bell, take a picture and send it to my PDA”[19]. Using a rule-based paradigm, a CAPpella [6] and iCAP [20] go one step further by allowing end-users to elaborate programs to control the behavior of ambient spaces. End User Programming has been around for many years [21]. It is now becoming a key challenging issue to be addressed in the near future.

## 9 Conclusion

Coupling is not a new phenomenon. In the GUI computer world, most couplings are pre-packaged and immutable. Typically, mice are coupled with display screens, while mice and keyboards are coupled for the input focus function. As a consequence, coupling is taken for granted by HCI designers and developers. However, in ambient computing, there is more than meets the eyes: a coupling is not an insulated dual state phenomenon.

First, coupling two interaction resources requires that they meet a number of conditions including their mutual compatibility, valence, and availability. Are these conditions observable, predictable, traceable, and controllable? We propose an 8 state automaton that models the life cycle of a coupling and that provides designers with a framework to verify whether usability properties are satisfied for each state of a particular coupling.

Second, the creation of a new coupling may have side effects on existing couplings. In this article, we have not investigated the destruction of couplings. On the other hand, we propose two formalisms, using a graph theoretic and an algebraic notation, to reason about the consequents of causal couplings in a systematic way. Here, we use the compatibility between the functions returned by a consequent and the functions provided by its two neighbors. Other rules could be used. In any case, are the consequents of a causal coupling observable, predictable, traceable, and controllable?

To provide a preliminary answer, we propose the concept of meta-UI as a unifying overarching interactive system that leads into end-user development. Ideally, the yet-to-be-invented meta-UI will allow users to construct and program powerful N-P molecules of any shape that will make sense for them. Progressively, patterns like the star-like construct of the Nabaztag will emerge. We are only at the beginning. And coupling is only one aspect of this large problem space.

**Acknowledgments.** This work has been partly supported by Project EMODE (ITEA-if4046) and the NoE SIMILAR- FP6-507609.

## References

1. Ballagas, R., Meredith, R., Stone, M., Borchers, J.: iStuff: A Physical User Interface Toolkit for Ubiquitous Computing Environments. In: Proc. Of CHI 2003, Ft.Lauderdale, Florida, pp. 537–544 (2003)
2. Bastien, J.M.C., Scapin, D.L.: Critères Ergonomiques pour l'Évaluation d'Interfaces Utilisateurs, Technical report 1993. INRIA (1993)
3. Bérard, F.: Vision par Ordinateur pour l'Interaction Homme-Machine Fortement Couplée, Thesis, Université Joseph Fourier, p. 200 (November 1999)
4. Card, S.K., Mackinlay, J.D., Robertson, G.: The design space of input devices. In: Proceedings of the SIGCHI, Seattle, Washington, United States, pp. 117–124 (1990)
5. Coutaz, J.: Meta-User Interface for Ambient Spaces. In: Coninx, K., Luyten, K., Schneider, K.A. (eds.) TAMODIA 2006. LNCS, vol. 4385, pp. 1–15. Springer, Heidelberg (2007)
6. Dey, A.K., Hamid, R., Beckmann, C., Li, I., Hsu, D.: A CAPpella: programming by demonstration of context-aware applications. In: Proceedings of the SIGCHI, CHI 2004, pp. 33–40. ACM Press, New York (2004)
7. Dobson, S., Nixon, P.: More principled design of pervasive computing systems. In: Bastide, R., Palanque, P., Roth, J. (eds.) DSV-IS 2004 and EHCI 2004. LNCS, vol. 3425, pp. 292–305. Springer, Heidelberg (2005)
8. Graham, C., Cockton, G.: Design Principles for Interactive Software. Chapman & Hall, London (1996)
9. Hinckley, K.: Synchronous gestures for multiple persons and computers. In: Proc. of UIST 2003, Vancouver, Canada, pp. 149–158 (2003)
10. Holmquist, L.E., Mattern, F., Schiele, B., Alahuhta, P., Beigl, M., Gellersen, H.W.: Smart-Its Friends: A Technique for Users to Easily Establish Connections between Smart Artefacts. In: Abowd, G.D., Brumitt, B., Shafer, S. (eds.) UbiComp 2001. LNCS, vol. 2201, pp. 116–221. Springer, Heidelberg (2001)
11. IST-2000-28323 FAME European project, <http://isl.ira.uka.de/fame/>
12. Jacob, R.J.K., Sibert, L.E., McFarlane, D.C., Mullen Jr., M.P.: Integrality and separability of input devices. ACM Transactions on Computer-Human Interaction 1(1), 3–26 (1994)
13. Johanson, B., Hutchins, G., Winograd, T., Stone, M.: PointRight: experience with flexible input redirection in interactive workspaces. In: Proceedings of the 15th annual ACM symposium on User interface software and technology UIST 2002, France, pp. 227–234 (2002)
14. Kurtenbach, G., Baudel, T.: Hypermarks: Issuing Commands by Drawing Marks in Hypercard. In: Proc. ACM SIGCHI Adjunct Proceedings, p. 64. ACM, New York (1992)
15. Lachenal, C.: Modèle et Infrastructure Logicielle pour l'Interaction multi-instrument, multi-surface. Thesis of University Joseph Fourier (December 2004)
16. Nielsen, J.: Usability engineering at a discount. In: Salvendy, G., Smith, M.J. (eds.) Designing and Using Human-Computer Interfaces and Knowledge Based Systems, pp. 394–401. Elsevier Science Publishers, Amsterdam (1989)
17. Norman, D.: User Centered System Design. Lawrence Erlbaum, Mahwah (1986)
18. Rekimoto, J., Ayatsuka, Y., Kohno, M.: SyncTap: An Interaction Technique for Mobile Networking. In: Proc. of MOBILE HCI 2003, Udine, Italy, pp. 104–115 (2003)
19. Rodden, T., Crabtree, A., Hemmings, T., Koleva, B., Humble, J., Akesson, K.P., Hansson, P.: Configuring the Ubiquitous Home. In: Proc. of the 2004 ACM Symposium on Designing Interactive Systems (DIS 2004), Cambridge, Massachusetts. ACM Press, New York (2004)

20. Sohn, T., Dey, A.: iCAP: an informal tool for interactive prototyping of context-aware applications. In: CHI 2003 Extended Abstracts on Human Factors in Computing Systems, Ft. Lauderdale, Florida, pp. 974–975. ACM Press, New York (2003)
21. Sutcliffe, A., Mehandjiev, N.: End-User Development. In: Communication of the ACM, special Issue on End-User Development. ACM publ., New York (2004)
22. Wharton, C., Rieman, J., Lewis, C., Polson, P.: The Cognitive Walkthrough Method: A Practitioner’s Guide. In: Nielsen, J., Mack, R.L. (eds.) Usability Inspection Methods, pp. 105–141. John Wiley & Sons, New York (1994)
23. Weiser, M.: The computer for the 21st century. *Scientific American*, 94–104 (September 1991)
24. Zhai, S.: User performance in relation to 3D input device design. *SIGGRAPH Comput. Graph.* 32(4), 50–54 (1998)

## Questions

**Peter Forbrig:**

*Question: Is coupling a feature of the interaction resource or of both resource and application?*

Answer: That is an open question.

*Question: Do you have a semantic specification of what coupling means?*

Answer: Coupling is either an action or a result; it’s really informal. They have articulated the problem.



# Building and Evaluating a Pattern Collection for the Domain of Workflow Modeling Tools

Kirstin Kohler and Daniel Kerkow

Fraunhofer IESE

Fraunhofer-Platz 1, 67663 Kaiserslautern

{kirstin.kohler,daniel.kerkow}@iese.fraunhofer.de

**Abstract.** In this paper, we present the results of a case study conducted together with a small company that develops a workflow modeling tool. During the case study, we created a pattern collection for the domain of workflow modeling tools and evaluated a subset of these patterns. Beside the pattern description itself, the contribution of our work is a systematic process for identifying patterns. The results of the case study showed, that the identified pattern are a valuable instrument for software developers to improve the usability of their software in the given domain. Additionally this finding shows that the process of pattern identification is valuable as well.

**Keywords:** User interface pattern, case study, design methodologies.

**ACM Classification Keywords:** H5.2. Information interfaces and presentations, Miscellaneous theory and methods, D.2.2 Software Engineering, Design Tools and Techniques.

## 1 Introduction

In the RL-KMU Project, we were in search of methods that support small and medium-sized companies in improving their usability competence. These companies usually do not have their own usability department; also, they cannot effort expensive usability training or consultancy [1]. Software engineers in these companies usually have to do user interface design and coding, but never learned how to systematically do this as part of their education. All these constraints made us investigate the use of user interface patterns in more detail.

User interface patterns are a promising approach to transfer knowledge about user interface design to user interface designers [2] [3] [4]. Due to the fact that patterns are a commonly accepted approach in the area of software engineering [5], they seem especially well suited to train/support software engineers [6]. In addition, several libraries are publicly available without extra charge and provide access to a large set of patterns [7] [8]. While validating the suitability of these libraries for the small enterprise in our project, it turned out that the libraries were not specific enough for the software applications developed in the company. The most popular libraries offer patterns for unspecific software systems. For specific domains, these patterns are often not sufficiently tailored. Some recently developed libraries have addressed more

specific domains, like web systems [9], e-business applications [10], or museum websites [11]. But none of the available pattern libraries addresses the design problems of our company, which develops a graphical workflow modeling tool.

In this paper, we present the results of a case study during which we developed a pattern collection for the domain of workflow modeling tools.

The contribution of our work is twofold:

- It includes the pattern description of 40 patterns identified in the domain of workflow modeling tools. We evaluated these patterns to ensure their validity.
- We developed a process to systematically derive patterns by abstracting them from best solutions found in software applications of the same domain.

In section two, we will elaborate the process we applied to derive the patterns. We will show one of the extracted patterns as an example.

Section three presents the results of the pattern evaluation, which gives evidence that the presented process to create patterns is valuable as well.

## 2 Derive Patterns from Best Solutions

In order to identify the workflow-specific patterns, we followed the steps shown in Figure 1. First, we created a usage model for novice users and phrased functional and nonfunctional requirements for workflow modeling tools based upon this usage

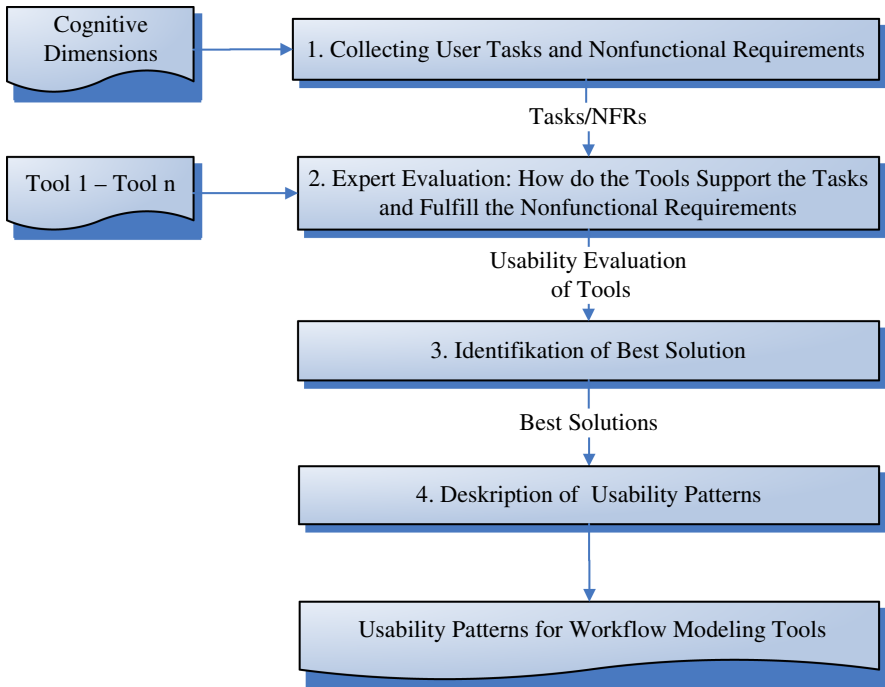


Fig. 1. Process for identifying patterns

model. In the next steps we evaluated strengths and weaknesses in the usability of four tools (cp. Table 2) and identified the best solutions among the four tools for each functional and nonfunctional requirement. In step 4, we described these solutions in a pattern notation in order to get a collection of 40 workflow-specific patterns.

In the following subsections, we will elaborate each of the 4 steps in more detail.

**Step 1: Collecting User Tasks and Nonfunctional Requirements**

The first step to gain the pattern was the creation of a usage model derived from the requirements of the workflow management tools. This usage model assumes the following imaginary inexperienced user, e.g., an employee of a small or medium-sized enterprise whose goal is to improve the effectiveness of a specific process within the company.

In order to improve the process, s/he has to implement the process into a workflow tool. The employee has not modeled business processes before and is familiar with standard PC applications, but not with workflow modeling tools.

The main reason for the choice of this actor is the productivity related goal of enabling non-experts to customize workflow tools and thus the higher degree of user support needed. The usage model represented by a use-case diagram is shown in Figure 2.

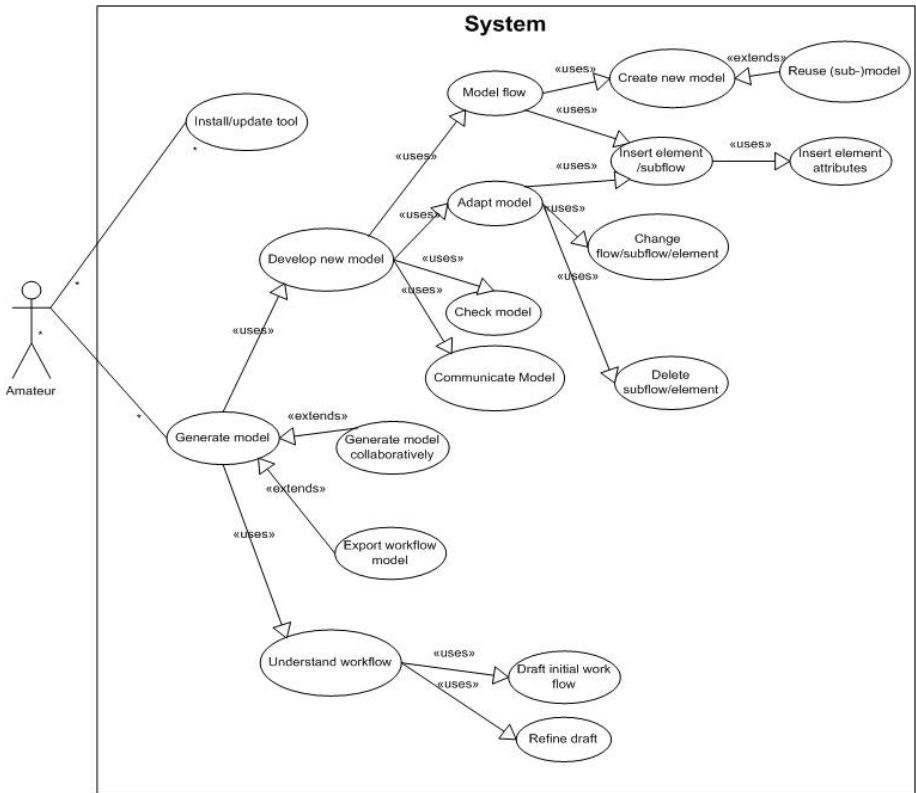


Fig. 2. Use-case diagram for workflow modeling tools

The employee, after having installed the workflow modeling tool, first of all wants to create or refresh his/her implicit mental model of the business process. Since s/he needs visual representations, the first activity is to build a sketch of the process. Sketching means prototyping on a high level of abstraction and requires many changes and refinements. The ability to get tool support in the sketching activity is a very strong requirement. Afterwards, the actual modeling begins. Elements representing the workflow (such as activity, role, or artifact) have to be inserted, refined, changed, and extended by certain attributes. It would be nice to have the possibility to check the model for correctness and adapt elements by deleting, changing, or inserting new elements. Since a process has many process stakeholders, the employee might want to create the workflow model collaboratively or at least export or communicate the model.

After having described each use case in detail, we added usability requirements. These usability requirements were derived from two kinds of sources: Quality models (e.g., ISO9126; ISO9241) and the “Cognitive Dimensions Framework” [12] (cp. Table 1). The latter recommends a set of criteria for the evaluation of notations, programming environments and data visualization.

The result of step 1 was a complete specification of functional and usability requirements for a workflow modeling tool. The next step was to evaluate existing tools against these requirements.

**Table 1.** Quality criteria used to derive usability requirements

<b>Usability Attributes from ISO 9126/ 9241</b>	<b>Cognitive Dimensions [12]</b>
Understandability	Viscosity: resistance to change
Learnability	Visibility: ability to view components easily
Operability	Premature commitment: constraints on the order of doing things
Attractiveness	Hidden dependencies: important links between entities are not visible
Usability compliance	Role-expressiveness: the purpose of an entity is readily inferred
Customizability	Error-proneness: the notation invites mistakes and the system gives little protection
Error tolerance	Abstraction: types and availability of abstraction mechanisms.
Conformity with user expectation	
Self descriptiveness	
Efficiency	

## Step 2: Expert Evaluation

In the next step, we explored four existing tools (cp. Table 2) with regard to their capability to accomplish the requirements. For this purpose usability experts from Fraunhofer IESE evaluated two scenarios. In each, they modeled a complete workflow. In Scenario 1, a large and complex workflow was modeled and in Scenario 2, it was a short workflow that was easy to oversee. The scenarios were chosen in order to capture each of the use cases introduced in Figure 2.

For each use case, the fulfillment of the functional requirements, the usability requirements, and the nonfunctional requirements derived from the “cognitive dimensions framework” was evaluated. Table 2 lists the analyzed tools. We will not rate the tools, since our purpose was not to compare the tools, but to identify the best solutions for our (tool-independent) requirements.

**Table 2.** List of the evaluated workflow modeling tools

Tool	Description	Source
Oracle BPEL Process Manager 2.0	Infrastructure for creating, deploying and managing BPEL (standard for assembling process flows) business processes.	www.oracle.com
Microsoft BizTalk Server 2004	The MS Visio-Add-In “Orchestration Designer” makes it possible to model business processes for execution in MS BizTalk Server 2004.	www.microsoft.com/biztalk
Essential Business Modeler 1.5	EBM 1.5 is a tool that combines proven techniques for modeling processes, enabling Model Driven Development of Enterprise Architectures.	www.essmod.com
IBM WBI Workbench	IBM WebSphere Business Integration Workbench V4.2.4 is a process modeling tool that makes it possible to test, analyze, simulate, and validate business process models	www-306.ibm.com/software/integration/wbimodeler/workbench/

**Table 3.** Example for a requirement and its corresponding best solution

Requirement	In the use case “create new model“, we phrased the nonfunctional requirement “enable a condensed representation of the complete process”. This requirement was derived from the dimension “visibility” according to the “Cognitive Dimensions Framework” and refers to the ability to view components easily. A complex workflow model can become too large to fit on a single screen. In order to get an overview of every component, a condensed view was required.
Solution	One of the tools offered an elegant solution to this requirement, the Condensed View Feature, which can always be utilized to gain an overview.

**Table 4.** Overview of the identified workflow modeling patterns

<b>Basic Pattern</b>	Autosave Templates
<b>Business Process Pattern</b>	Scopes; Complement attributes; Automatic coupling; Unambiguous attribute names; Unambiguous types of elements; Define types of elements; Facilitate connections; References to other process flows;
<b>Collaborative Work Pattern</b>	Automatic matching of different versions; Color markup; General markups; Multi-user-developing;
<b>Create / Debug Pattern</b>	Auto alert; Auto-correction; Automatic insert; Drop-down boxes; Isolated element deletion; Decisions on demand; Compare screens; Add comments; Context menu; Simulate and test; Sketch; Search; Name symbols; Validate logic;
<b>Documentation &amp; Help Pattern</b>	Documentation & tutorials; Help for attributes; Online help;
<b>Drawing Pattern</b>	Rulers; Conformity to graphic tools;
<b>Format Pattern</b>	Export; Import; Reports
<b>View Pattern</b>	Abstraction levels; Layer; Condensed view; Visibility; Full screen view; Zoom;
<b>Workspace pattern</b>	Adapt workspace; Insert workspace; Notations and working modus; Systematic divisions; Unlimited workspace;

**Step 3: Identification of the Best Solution**

For each positively evaluated functional requirement and for each positively evaluated nonfunctional requirement, the design solution implemented in the tool was

analyzed. Every good solution that met our requirements would be a candidate for a workflow-usability pattern. We will give an example for such a best solution in Table 3:

For each best solution, we derived a pattern by abstracting from the concrete solution and describing the principles of that specific solution. Table 5 demonstrates an example of one of the workflow patterns.

#### Step 4: Description of Usability Pattern

In this way, we were able to identify 40 different patterns. The patterns were classified into several types of patterns as listed in Table 4. Some of these patterns seem to be useful in other domains as well. The basic patterns, for instance, are applicable to almost any kind of application, while the drawing patterns should be found especially in graphic tools. The complete set of categories is probably applicable to any graphical modeling tool.

For a detailed description of all patterns, see [13]. To get an impression of the pattern description, in Table 5 we present the view pattern “Condensed View”.

**Table 5.** Example for the pattern “Condensed View”

<b>Name</b>	Condensed View
<b>Category</b>	View Pattern
<b>Related to</b>	<related pattern names, not only from the workflow pattern collection>
<b>Problem</b>	User wants to gain an overview, or wants to navigate within the workflow model. The model has too many elements and levels of abstractions and cannot be represented on a single screen.
<b>Forces</b>	Condensed representation of the workflow model (visibility); the exact position to insert a new element has to be identified; a specific position within the workflow has to be found; the position of an erroneous element within the workflow has to be identified; easy cognitive walkthrough activity.
<b>Context</b>	Workflow does not fit on a single computer screen and is smaller than 400 symbols.
<b>Solution</b>	Present the complete (downsized) process in a separate window, without scrollbars. Upon double-clicking on a specific spot, center the same spot in the main screen showing the details.
<b>Known Uses</b>	<name of the tool with the best solution>

### 3 Pattern Evaluation

Usually, pattern descriptions end up in libraries without any empirical validation. Few empirical results are published about the usage of user interface patterns in general [4] [14]. Very few pattern authors set up rules to assure a certain level of quality for their patterns [15]. For example, at “Yahoo!” [16], a solution has to be used in at least two software systems before it becomes a “pattern”.

To validate the usefulness of the pattern in our project, we wanted to go far beyond this. We formulated the following research hypotheses as a foundation of our investigation:

H: The identified patterns support the software developers in improving the usability of their application.

In order to elaborate this hypothesis we divided it up into the following sub-questions:

Q1: Do the identified patterns match the design challenges in the domain of workflow modeling tools?

Q2: Do software developers understand the pattern description?

Q3: Does the solution proposed in the pattern description solve the problem stated in the pattern description? (Internal consistency of the pattern)

Q4: Can the solution proposed in the pattern description be transferred to a concrete solution in a software system? (Concretization)

Before exploring each of the hypotheses in more detail by stressing their meaning and showing the results of the case study concerning the questions, we will explain the general design of the case study.

### 3.1 Design of the Case Study

The case study was conducted in three steps. We will elaborate the steps while referring to Figure 3 to clarify the rationale of the case study design. Figure 3 illustrates the relationship between the usability problems and the pattern description, as well as the hypothesis/questions that drove the case study.

1. We did a usability test to identify the current weaknesses of the workflow modeling tool. The test was performed with three test users. As part of the test, they had to modify and extend a given workflow, presented in the graphical environment of the workflow modeling tool. In total, 37 usability bugs were identified as result of the usability test. Figure 3 shows the usability problems as little circles in the software.

The purpose of the usability test was to identify the weaknesses in the current design that could be solved by using the design patterns. Neither the person conducting the usability test nor the test users knew the pattern before.

2. In a second step, we matched the usability problems to problem descriptions in our pattern collection. This step is represented as an arrow with the title “pattern matching”.

3. Afterwards, we conducted an expert evaluation by running guided interviews with three experts. Two interview partners were usability experts. The third one was the lead software engineer of the workflow modeling tool company. Involving experts from both areas was important, because we believe that for the different research questions listed above, expertise from different areas is necessary. The “Understandability” is especially important from the view of the software developer, whereas the question of whether a pattern solution solves a pattern problem should be validated by a usability expert (this question refers to the arrow “concretization” in Figure 3). During the interviews, the 10 patterns were presented one after the other to our interview partners. For each pattern, we investigated Q2, Q3, and Q4. Additionally we asked whether we matched the patterns correctly.



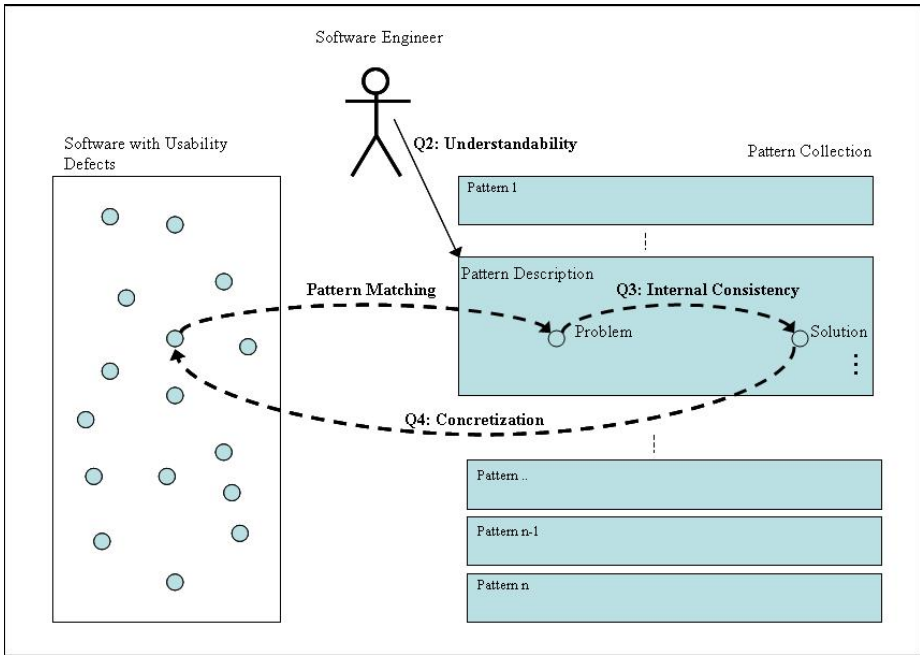


Fig. 3. Research questions of the case study

### 3.2 Contribution to Design Challenges in the Domain of Workflow Modeling Tools

We investigated the question of whether the identified patterns match the design challenges in the domain of workflow modeling tools. Only if the proposed patterns solve challenges in that domain is the pattern collection of any value.

Our case study showed that 10 of the 40 identified patterns matched one or more of the 37 usability bugs. Some bugs matched more than one pattern. In summary, 12 bugs could be linked to patterns. Those patterns not matching any of the identified design problems either do not cover tasks that were set up in the usability test, or the system contained already a usable solution. For example, none of the tasks conducted in the usability test covered the task of debugging a workflow for execution or working on a workflow collaboratively, but 6 of the 40 patterns support these tasks.

For one of the 10 patterns our experts judged the matching between usability defect and pattern as wrong. Of the remaining nine, seven were judged to be valuable contributions to the domain of workflow modeling tools.

The following investigation refers to the 9 “matching” patterns of our collection. The case study has to be extended in order to make a statement concerning the remaining 30 patterns.

### 3.3 Understandability of Patterns

We investigated the question of whether software developers understand the pattern description. The appropriate wording is the precondition for their usage by software engineers.

8 of the 9 pattern descriptions were judged to be understandable. Nevertheless, our interview partners gave us hints onto improve the description for all nine patterns. Terms from the domain of workflow modeling tools have to be worked out more properly in order to improve the understandability of the current description. Also, the names of the pattern should be made more specific and recognizable.

### 3.4 Internal Consistency of the Patterns

We wanted to find out whether the solutions described as part of the pattern descriptions really solve the problems stated in the problem description of the pattern. We call this internal consistency of the pattern. Only if the internal consistency is given for a pattern, it can improve the usability of a system.

8 of the 9 patterns under investigation were judged to be internally consistent. For one of the patterns, we got a suggestion to improve the solution, and for one pattern, an additional, alternative solution was proposed.

### 3.5 Applicability of Abstract Pattern Solution to a Concrete Software Design Solution

Even if a pattern is internally consistent, its description of the solution is understandable, and the selected pattern could be matched to a given usability problem of the software system, it might happen that the pattern cannot be transferred to a usable solution in the software system. What remains as a possible pitfall is the step of concretization, which means the transfer of the abstract pattern description to a concrete solution in the software system. This step has to be made by the software developer when applying the pattern. Figure 4 elaborates this problem in more detail.

The solution  $\langle s \rangle$  given in a pattern description  $\langle P \rangle$  is an abstraction of a concrete solution  $\langle s^* \rangle$  found in another software system X. For example, layout or color details as well as very detailed interaction steps were not specified as part of the pattern description. When applying pattern P to software system Y, the solution  $\langle s' \rangle$  is a concretization of  $\langle s \rangle$ .  $\langle s' \rangle$  may differ in many details from the original solution  $\langle s^* \rangle$ . If the pattern description is not complete or leaves design decisions open that are important to address Problem  $\langle p' \rangle$ , concretization might fail and end up in a solution  $\langle s' \rangle$  that does not improve the usability of system Y.

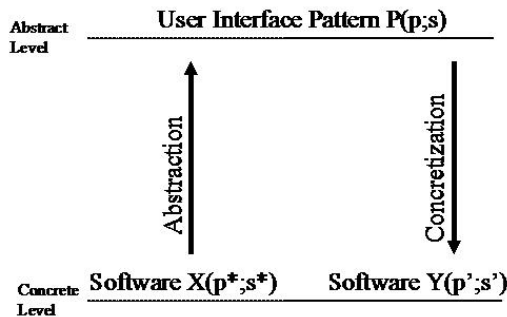


Fig. 4. Abstraction and concretisation of patterns

Our experts judged 8 of the 9 patterns to be suitable for deriving a concrete design solution. For a second pattern, one expert proposed a more concrete description.

We believe that we need to run additional experiments to gain more insights into the problem of concretization. A guided interview is not well suited for investigating this. Running usability tests on the next version of the software that contains implementations of the suggested pattern could show us whether the concrete solutions really improve usability.

## 4 Conclusion and Future Work

The findings support our main hypothesis: For 7 of the 9 patterns, the evaluation showed that the identified patterns support the software developers in improving the usability of their application. As a positive side effect, we found a lot of valuable hints to improve the pattern descriptions. The positive judgment of the patterns under investigation can be interpreted as evidence for the quality of the process we used to identify the patterns.

The quality requirements for user interface patterns – like understandability, internal consistency, and ability for concretization - worked out in the design of the case study, gave us further ideas how to improve the process of pattern identification. With our future work, we want to enrich the process with guidelines for pattern description in order to improve step 4 of the pattern identification process. The guidelines should formalize the consistency between solution and problem and the process of abstraction. As a consequence, the probability of deriving “high quality” pattern collections will increase.

Controlled experiments should investigate the contribution of our patterns to the quality of the end product in terms of statistically valid data. We are also thinking about evaluating the identified pattern in a “design from scratch“-experiment. This experiment will investigate how patterns not only improve a given design, but also support the new design of a system.

In future projects, we want to extend our research to process guidance for software developers in finding the right pattern description in a given pattern collection or library. Only if this step is sufficiently well supported can user interface patterns support software developers in improving user interface design in their daily work.

**Acknowledgements.** The authors wish to acknowledge the contributions of Steffen Hess with respect to the process of pattern identification and Jörg Grimm in conducting the pattern evaluation study.

The project was performed with financial support from „European Regional Development Fund“ and the state “Rheinland-Pfalz” (Förderkennzeichen: MWVLW, Az.: 8315 38 51 04 IESE, Kapitel 0877 Titel 892 02).

## References

- [1] Kerkow, D., Schmidt, K., Wiebelt, F.: Requirements for the Integration of UE Methods in SE Processes from the Perspective of Small and Medium-sized Enterprises (SMEs). In: INTERACT Workshop: Integrating Software Engineering and Usability Engineering, Rome (2005)
- [2] Griffiths, R.N., Pemberton, L.: Don't write guidelines write patterns (2006)

- [3] Dearden, A., Finlay, J., McManus, L.A.B.: Using Pattern Languages in Participatory Design. In: Participatory Design Conference, Palo Alto (2002)
- [4] Cowley, N.L.O., Wesson, J.L.: An Experiment to Measure the Usefulness of Patterns in the Interaction Design Process. In: Costabile, M.F., Paternó, F. (eds.) INTERACT 2005. LNCS, vol. 3585, pp. 1142–1145. Springer, Heidelberg (2005)
- [5] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, Reading (1995)
- [6] Seffah, A., Desmarais, M., Metzker, E.: HCI, Usability and Software Engineering Integration: Present and Future. In: Seffah, A., Gulliksen, J., Desmarais, M. (eds.) Human-Centered Software Engineering. Springer, Heidelberg (2005)
- [7] Tidwell, J.: COMMON GROUND: A Pattern Language for Human-Computer Interface Design, vol. 2006 (1999) (last updated)
- [8] Welie, M.V.: Patterns in interaction design (2003)
- [9] Graham, I.: A Pattern Language for Web Usability, London (2003)
- [10] Richter, A.: Generating User Interface Design Patterns for Web-based E-business Applications. In: Interact Workshop: Software and Usability Cross-Pollination: The Role of Usability Patterns, 2nd IFIP WG13.2 Workshop on Software and Usability (2003)
- [11] Borchers, J.: Interaction Design Patterns: Twelve Theses. Position Paper, Workshop Pattern Languages for Interaction Design: Building Momentum. In: Workshop Pattern Languages for Interaction Design: Building Momentum, CHI 2000, The Hague, Netherlands (2000)
- [12] Green, T.R.G., Petre, M.: Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. Journal of Visual Languages and Computing 7, 131–174 (1996)
- [13] Kohler, K., Kerkow, D., Hess, S., Schmid, K.: Best Practices und Usability Pattern für Geschäftsprozess-Modellierungswerkzeuge, 060.05/D (2005)
- [14] Wessen, J., Cowley, L.: Designing with Patterns: Possibilities and Pitfalls. In: Interact Workshop: Software and Usability Cross-Pollination: The Role of Usability Patterns, 2nd IFIP WG13.2 Workshop on Software and Usability (2003)
- [15] Todd, E., Kemp, E., Phillips, C.: What makes a good user interface pattern language? In: Proceedings of the fifth conference on Australasian user interface, Dunedin, New Zealand, vol. 28 (2004)
- [16] Leacock, M., Malone, E., Wheeler, C.: Implementing a Pattern Library in the Real World: A Yahoo! Case Study (2005)

## Questions

### **Gerrit van der Veer:**

*Question: The presented approach is very systematic, based on (1) finding a problem, (2) analyzing the problem, (3) finding a solution and (4) validating the solution. Unfortunately the approach has not been applied and tested in different domain. This raises the issue of its generality.*

Answer: There is no doubt that the approach and the presented ideas should be tested in different domains as well. This may be part of future work.

### **Peter Forbrig:**

*Question: How does the pattern specification relate to workflow specifications?*

Answer: The solution part of the patterns may (informally) entail information which can be used to derive (in part) a workflow specification.

# Do We Practise What We Preach in Formulating Our Design and Development Methods?

Paula Kotzé<sup>1</sup> and Karen Renaud<sup>2</sup>

<sup>1</sup> Meraka Institute (CSIR) and School of Computing (UNISA),  
P O Box 395, Pretoria, 0001, South Africa

<sup>2</sup> Department of Computing Science, University of Glasgow,  
Lilybank Gardens 17, Glasgow, G12 8RZ, United Kingdom  
paula.kotze@meraka.org.za, karen@dcs.gla.ac.uk

**Abstract.** It is important, for our credibility as user interface designers and educators, that we practice what we preach. Many system designers and programmers remain sceptical about the need for user-centred design. To win them over, we need to be absolutely clear about what they need to do. We, as a community, propose many different methods to support naïve designers so that they will design and implement user-centred systems. One of the most popular methods is HCI design patterns – captured and formulated by experts for the sole purpose of transferring knowledge to novices. In this paper we investigate the usability of these patterns, using both theoretical and experimental analysis, and conclude that they are *not* usable. Hence, unfortunately, we have to conclude that we don't practice what we preach. We conclude the paper by making some suggestions about how we can address this situation.

**Keywords:** Design patterns, usability, learnability, memorability, efficiency, errors, satisfaction.

## 1 Introduction

In human-computer interaction we advocate that human factors must be considered during the planning, design, development, implementation and evaluation stages of interactive systems. In software engineering there is a growing awareness of human factor issues, although few of the effects of this awareness are evident in actual systems development processes and delivered system interfaces. A myriad of tools, techniques, methods, etc. are being advocated for use by designers and developers to support them in developing systems that cater for the human factor issues in interactive systems. A cursory scan of any of the prominent textbooks used in the teaching of HCI will reveal many of these techniques. Examples of these are lifecycle models, such as the Star lifecycle model by Hartson and Hix [24], the Usability Engineering Lifecycle by Mayhew [39], and the Simple Lifecycle Model of Preece *et al.* [46]. There are also design rules, such as principles to support usability [15, 44], standards [26, 27], guidelines [38, 52], golden rules [51] and heuristics [43], and HCI design patterns [18, 55, 57].

Most of these techniques and tools attempt to address the needs of the *user* of the interactive system. There is also another angle to be considered: that of the *designer* and/or developer of the interactive system. The above-mentioned methods claim to facilitate the design of usable systems, but the question we are asking is whether these methods themselves are usable? This paper focuses on this key question: do the guidelines and principles we promote for facilitating the design of usable products apply to the very methods we advocate for the development of such usable products? Furthermore, do these methods adhere to the usability principles advocated by usability experts such as Nielsen [43]?

Using both a theoretical and experimental analysis, this paper will examine the use, by designers, of one of the most popular methods and one that has received a lot of attention in recent years: *HCI design patterns*. We will analyse design patterns from the perspective of the most widely accepted usability metrics with special attention being paid to the most relevant of these: learnability and memorability.

Section 2 introduces and discusses patterns. Section 3 describes widely accepted usability metrics. Sections 4 to 7 consider patterns from the perspective of each of these metrics in turn. Section 8 wraps up by considering how the usability of patterns can be improved. Section 9 concludes.

## 2 HCI Design Patterns

A design pattern can be defined as ‘a piece of literature that describes a design problem and a general solution for the problem in a particular context’ [10:2]. Designers have striven towards the elusive goal of reuse for many years now, but it only became widely achievable with the advent of the object-oriented paradigm [20] and the patterns that emerged from repeated use of successful object-orientation. The use of design patterns in HCI was a natural progression from the use of patterns in other domains and was discussed at a number of workshops in the late 1990s (for example at CHI '97, INTERACT '99, and HCI '00) [14]. An influential book by Gamma, Helm, Johnson and Vlissides [20], based on the Alexandrian format, also played a role in promoting the acceptance and use of design patterns in the field of HCI [4].

An object-oriented SE design pattern can be considered a ‘solution to a general design problem in the form of a set of interacting classes that have to be customized to create a specific design’ [48:225]. The definition of an HCI design pattern has a somewhat different perspective – as a proven solution for a common user interface or usability problem that occurs in a specific context of work [14].

HCI design patterns are assigned to different categories, including task representation, dialogue, navigation, information, status representation, layout, device aspects and physical interaction, user-profile, and overall system architecture [14]. A comprehensive list of HCI design patterns is available from Tidwell’s collection [55], Sally Fincher’s Pattern Form Gallery [18] and Van Welie’s collection [57], amongst others.

Over the last few years, the idea of anti-patterns has gained favour in SE design pattern research [36, 58]. Anti-patterns capture poor or sub-optimal design or software development practices, and many also explain why such practices appear attractive to a novice and why they turn out to be a bad solution [6, 9]. The basic rationale in

publishing anti-patterns is to identify recurring design flaws for the purpose of preventing other people from making the same mistakes. An anti-pattern is therefore a pattern that ‘describes a commonly occurring solution to a problem that generates decidedly negative consequences’ [6:7].

HCI patterns and pattern languages are characterised by a number of features, that, it is claimed, distinguish them from rules and guidelines [2, 14, 16]:

- They capture design practise and represent knowledge about successful solutions (in the case of patterns) or unsuccessful solutions (in the case of anti-patterns).
- They encapsulate the essential common properties of good design, but do not tell the designer exactly how to do something, but rather *when* to do something and *why*.
- They represent design knowledge at varying levels, encompassing a range of issues from social issues through to widget design.
- They are not neutral but represent values within their rationale, e.g. they can express values about what is humane in interface design.
- As the concept of pattern languages is generative in nature, they can provide support in the development of complete designs.
- Patterns appear to be an effort to introduce an HCI-wide ‘lingua franca’. They are, in general, claimed to be intuitive and comprehensible and it is claimed that they can therefore be used as a communication medium between various stakeholders. If this claim is true, then HCI patterns should be *accessible and understandable by end-users*. The end-users, in our context, are the designers of user interfaces.

Having given a brief overview of patterns, we now consider their usability in supporting the design process in the following sections.

### 3 Usability

The ISO 9241 Standard [26] defines usability as the effectiveness, efficiency and satisfaction experienced by a user in achieving specified goals in a specific environment. These three aspects are in line with the five attributes that contribute to usability as identified by Nielsen [42]:

1. *Learnability*: Learnability refers to the promptness with which users start performing their tasks with the system. It pertains to the features allowing novice users to understand how to use the system initially and how to attain a maximal level of performance once the system has been mastered [15]. This aspect is directly related to short-term memory and the skill acquisition process.
2. *Memorability*: Memorability refers to how easy it is to remember how to use a system feature, once learned [46] and the effort required to reuse the system feature after not having used it for some time. This aspect is directly related to long-term memory and skill retention. If something is memorable, it can be recalled with little conscious effort.
3. *Efficiency*: Efficiency refers to the level of productivity, i.e. the resources spent in relation to the accuracy and completeness of the goals achieved [26]. Efficiency therefore refers to the ways in which a system supports users in carrying out their

tasks [46]. The kinds of resources we usually measure are time and monetary cost to the user.

4. *Errors*: Users should be able to use the system with accuracy without making undue errors, and, if errors *are* made, they should be able to recover from them and still achieve their goals with minimal disruption.
5. *Satisfaction*: Satisfaction refers to the comfort and acceptability of the user-system interaction process, as well as the effects on other people affected by its use [26]. This is also related to the cognitive load placed on a user by the system – if the cognitive load is high, users will generally feel dissatisfaction.

The following section will consider learnability and memorability issues, since both are related to memory and therefore cannot be separated. For example, a system cannot be memorable unless it is easily mastered – and it needs to exhibit a high level of learnability to support this.

## 4 Learnability and Memorability

### 4.1 How Do Humans Learn?

To judge anything in terms of learnability and memorability, we must first understand how humans learn and remember things, i.e. how we form mental models and how knowledge transfer takes place.

People store what they know in mental models, which are small-scale psychological representations of real, hypothetical, or imaginary situations [12]. The mind constructs mental models as a result of perception, imagination and knowledge, and the comprehension of discourse [28, 29] in order to be able to anticipate events, to reason and to underlie explanation. It is therefore reasonable to assume that we construct mental models to represent HCI patterns (and anti-patterns) in a problem context.

People ‘learn’ by repeated exposure to concepts using one of two major types of learning: implicit or explicit:

- *Implicit learning*, or unintended learning or tacit (silent) learning [45, 47], can be seen as a passive process where people, when exposed to information, simply acquire knowledge of the information by means of that exposure, i.e. it is unconscious and always active [30, 47, 54]. Invoking implicit knowledge involves the indirect application of the knowledge without the requirement of knowledge declaration [30]. This aspect is thus related to the memorability of a system.
- *Explicit learning*, or intended learning, in contrast, is characterised by people actively seeking out the structure of any information presented to them, i.e. it is intentional and conscious [3, 30, 54]. For example, explicit learning would be involved if a designer is instructed to acquire some target knowledge and then explicitly to apply and state the knowledge acquired in design phase [30]. This aspect is related to the learnability of the system.



An alternative perspective on learning, closer to the process of learning as supported by HCI design patterns, is presented by Gorman [23], who identifies four types of knowledge in technology transfer:

1. *Declarative knowledge (what)* refers to the recall of facts and events. Declarative knowledge is composed of chunks, consisting of a number of slots each of which can hold a value (which can also be another chunk) [33]. In the context of design patterns this is the process of learning about a design pattern – its name, its rationale, its recommended application.
2. *Procedural knowledge (how)* that refers to the skill of knowing how to do something. Procedural knowledge is usually encoded as declarative knowledge first and then translated into procedures (algorithms) [1], but can also be learned by feel or intuition. Procedural knowledge therefore consists of productions, which are condition-action pairs specifying the action to be taken if a particular condition is satisfied [33]. In the context of design patterns this is the process of learning how to use the design pattern.
3. *Judgement knowledge (when)* that involves the ability to recognise when knowledge is applicable to a particular instance, i.e. recognising that a problem is similar to one for which a solution is known and knowing when to apply a particular procedure or solution. Judgement knowledge is therefore structured in a way that facilitates problem solving, and is usually applied by experts in a particular context. Whereas novices would rely more on declarative and to a lesser extent on general or weak heuristics based on procedural knowledge, experts rely more on judgement knowledge. [33]. In the context of design patterns this is the process of learning to recognise situations where the previously learnt pattern should be applied.
4. *Wisdom (why)* knowledge refers to meta-cognitive monitoring which may lead to a new course of action. It is related to judgement knowledge referring to the ability to reflect, question, and come up with new courses of action. It involves an element of moral reasoning. [33]. In the context of design patterns this is the process of understanding the rationale of the pattern, and understanding why it comprises a good and effective design.

This model is confirmed by Miller [41] in his ‘pyramid of competence’. Miller was concerned with the assessment of medical students. He proposes 4 levels of competence:

1. *Knows* – factual knowledge.
2. *Knows how* – ability to apply the knowledge.
3. *Shows how* – ability to identify situations where knowledge can be applied.
4. *Does* – ability to use the skills in everyday medical practice.

Level 1 aligns well with Gorman’s ‘what’ level. Gorman’s ‘how’ level encompasses level 2 of Miller’s pyramid while level 3 accords well with Gorman’s ‘when’ level. Finally Gorman’s ‘why’ level can be thought to be somewhat similar to level 4 – the ‘does’ level (see Fig. 1). Interestingly, both Miller and Gorman communicate the concept of different kinds of knowledge building onto each other, and the acquisition of the knowledge being acquired in a particular sequence over a period of time.

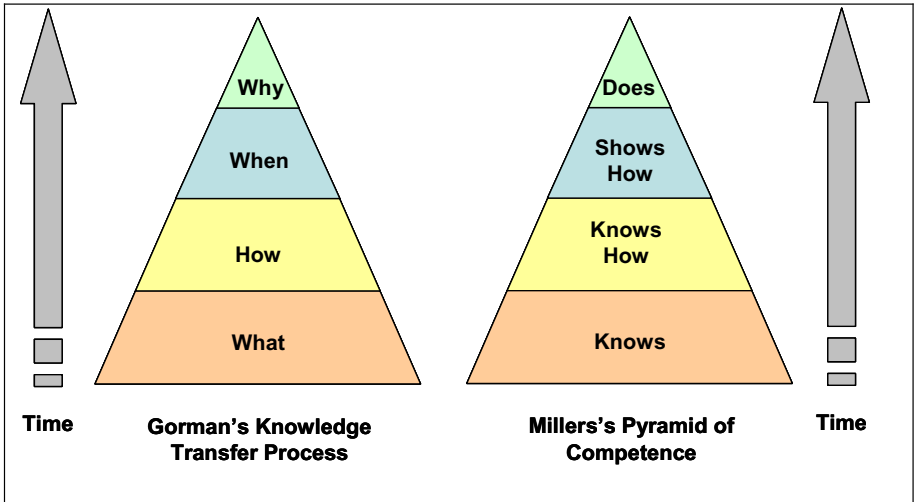
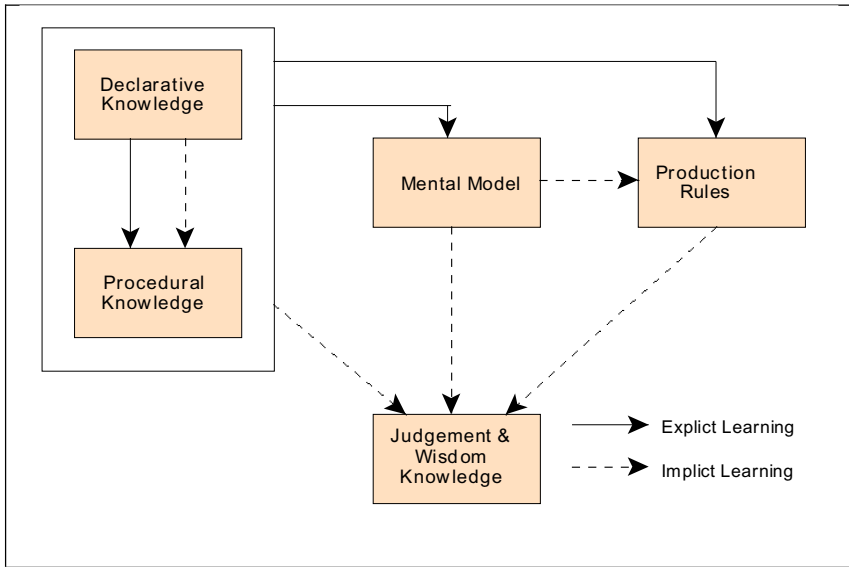


Fig. 1. Gorman and Miller's perspectives on knowledge transfer models

The distinction between declarative and procedural knowledge maps roughly onto the distinction between explicit and implicit knowledge since declarative knowledge is generally accessible (and therefore explicit) while procedural knowledge is generally inaccessible (and therefore implicit). It is, however, not uncommon for implicit learning also to require declarative knowledge, although there is no consensus as to the function or the source of the declarative knowledge [30]. The development of judgement knowledge is also implicit, and occurs over a period of time during the process of applying declarative and procedural knowledge to problems or instances, and whilst experience is gained in the use of this knowledge. Wisdom is tacit knowledge and therefore implicit [23]. Wass *et al.* [59] refer to Miller's pyramid of competence and point out the difficulty of assessing whether a student has reached competence in the top-most level of the pyramid. They argue that, even if the student is able to pass exams testing the first two competencies and is observed treating a patient to test the third level ('shows how'), this still does not guarantee competence at the apex of the pyramid. The implication is that the 'does' competence does not follow automatically from the student having mastered the knowledge this builds on. This appears to imply that the 'does' competence is implicitly mastered, unlike the explicitly studied knowledge it builds on. In this context, Fig. 2 gives a graphical representation of the relationship between implicit and explicit learning and the four knowledge types identified by Gorman [23].

Whether or not implicit or explicit learning is involved, one cannot present a concept only briefly and expect it to be encoded and available for retrieval after any significant interval without any further effort. There has to be an effort made in order to encode the information. If, during the encoding process, the new concept is linked to already-encoded knowledge, the retrieval process becomes easier and more likely at a later stage. Repeated exposure to a concept strengthens the encoding and makes retrieval faster and stronger, i.e. memorability is improved.



**Fig. 2.** The relationship between different types of learning and knowledge types

Fig. 1 aims graphically to depict the knowledge transfer/acquisition process using Gorman’s [23] and Miller’s classifications and their relationship with time. It indicates that time is required to form procedural knowledge based on acquired declarative knowledge, and then judgement and wisdom knowledge built on these. We can therefore realistically use the Gorman model, as presented in this figure, to evaluate the learnability and memorability of HCI design patterns and anti-patterns.

#### 4.2 Knowledge Encapsulated in HCI Design Patterns

A pattern aims to encompass all the different types of knowledge enumerated by Gorman [23]. The procedural and declarative knowledge types can be taught and learnt but the judgement and wisdom knowledge can only be assimilated over time. It therefore is clear that novice designers master the declarative and, to a small extent, the procedural pattern-related knowledge, but that they do not develop judgement knowledge very quickly. This is probably due to the fact that the only way to develop judgement knowledge is by making use of the declarative and procedural knowledge over a period of time. Gorman [23] explains that judgement knowledge is developed gradually over a long period of time, so it is perfectly understandable that novice designers cannot develop this knowledge simply because they have been given a book of design patterns to read. Judgement knowledge is implicit – and is developed in the process of using explicit knowledge repeatedly, in context.

However, given the fact that patterns *are* being used as a knowledge transfer artefacts, let us consider how a novice designer might assimilate the knowledge captured in the pattern.

A novice designer’s receptivity to the pattern creator’s envisaged transfer of pattern-encapsulated knowledge will depend absolutely on how well the pattern is formulated and how strongly it is linked to the problem for which the pattern is the solution. The efficacy of the pattern, therefore, does *not* depend on the technical brilliance of the implemented design, but rather on the quality of the mental model the user constructs as a result of the way in which the pattern is structured and presented. This internalised mental model will be matched against future design problems encountered by the novice, and used if the problem matches the potential solution proffered by the model. If the model is sufficiently well captured, there is a better chance of the learner identifying it and using it. Hence, the efficacy of any design pattern’s knowledge transfer process depends on how well the issues in the pattern are communicated to the learner *at the first encounter*, which is when the pattern is first understood and internalised, and the mental model constructed [56].

The tricky problem in the formulation of effective patterns therefore lies in ensuring that the formulation satisfies the needs of naïve user interface designers. Experts often omit essential details, simply because they assume knowledge of these facts. The efforts of many researchers in the field of HCI design patterns have been aimed at closing this communication gap [17, 50]. When we consider the use of patterns in HCI knowledge transfer, the closing of this gap becomes essential.

Fig. 3 contrasts the knowledge transfer model (as illustrated in Fig. 1) with the general presentation structure of HCI design patterns [55, 57]. We used the Tidwell HCI Pattern Definition format [55] as example format, but other HCI design pattern formats have a similar structure.

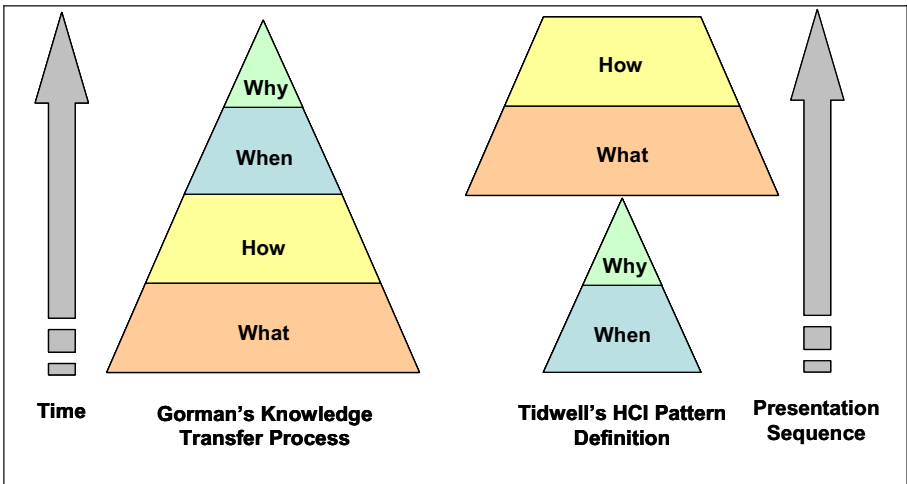


Fig. 3. The pattern presentation sequence vs. the knowledge transfer process

When we study Fig. 3 closely, we uncover what may be the primary reason for the difficulties many naïve designers have with comprehending and using patterns. The order in which information is presented in patterns, and the assumptions of embedded knowledge linked to this imposed order, simply do not align with the knowledge transfer process, which needs to occur in a specific sequence. Patterns typically

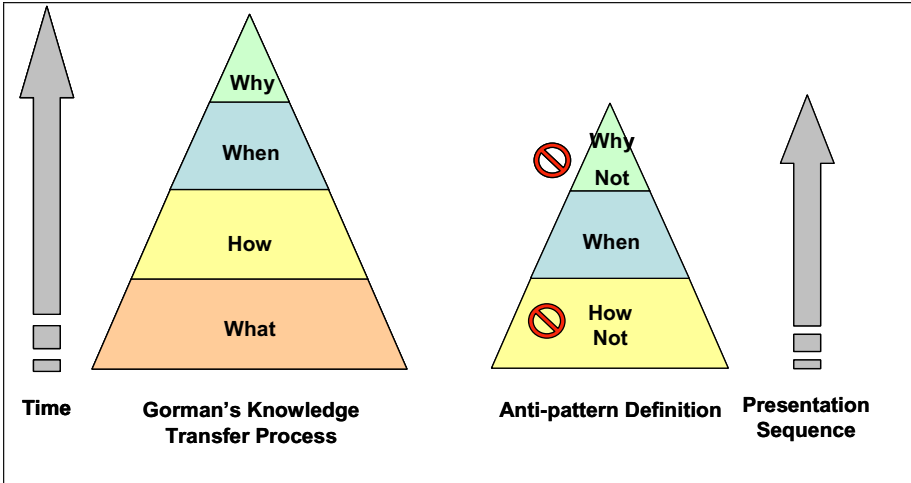


Fig. 4. The anti-pattern presentation sequence vs. the knowledge transfer process

introduce first the ‘when’ and the ‘why’ and this assumes prior mastery of the ‘what’ and the ‘how’. Patterns appear, at first glance, to accord well with human information processing processes because they include information related to all the mental model knowledge representation processes. However, their knowledge presentation structure does not align correctly with the accepted knowledge acquisition process and this could impair their efficacy.

This problem is even more severe when anti-patterns are contemplated, since the cognitive processing of anti-patterns has to deal with negation. An anti-pattern theoretically shows how to do the ‘opposite’ of the required solution or ‘how not to do it’ (not necessarily the opposite of any proper solution).

The negation schema involved with anti-patterns is the schema-plus-tag model [32]. The schema-plus-tag model states that the core supposition of a premise is processed as a cognitive unit, which is then marked with a negative tag [8, 40]. The critical issues are the argument that the core can be disassociated from the negation tag at a later stage (and as result the individual might remember the opposite of the intended meaning), and that the consideration of the core supposition activates associations congruent with the core, but incongruent with the intended meaning of the negation as a whole. The negation of a premise is therefore kept as a ‘mental footnote’ in the designer’s mind, whereas the solution itself is kept as a mental model. These tags sometimes fail to activate and can lead to systematic errors and illusions. For example, if you tell a designer: ‘don’t use red print on a green button’, the designer has to think about the green button with red print on it before storing it with the footnote reminding him/her of the folly of this course of action. According to the schema-plus-tag model we tend to internalise what we focus on, so when the designer thinks of colour schemes for a button s/he may well use a green button with red print because the mental trace to that concept has survived but the footnote has failed to activate.

Fig. 3 demonstrated the inherent defects related to the commonly used design pattern structure. Fig. 4 compares the anti-pattern structures to Gorman’s knowledge transfer process. There are two things to be noted about this comparison:

1. Two ‘not’ tags are used – ‘how not’ and ‘why not’. This invokes the use of the schema-plus-tag negation model, and either or both tags could thus easily go missing.
2. The anti-pattern assumes prior knowledge of the ‘what’, which, in a novice, cannot be assumed (the ‘what’ knowledge is not explained or referenced in an anti-pattern presentation).

The effects of anti-patterns on novice designers, therefore, could be confusing, at least, and detrimental, at worst.

### 4.3 Learnability and Memorability of HCI Design Patterns and Anti-patterns

From the arguments above it seems as if design patterns will indeed exhibit problems when assessed for learnability and memorability. But is this indeed the case?

In researching the practice of teaching in the negative we did a number of experiments with the teaching of patterns and anti-patterns and observed how students learn based on the mode of teaching. The results of these experiments are described in detail in Kotzé, Renaud and Van Biljon [32], but we will highlight our findings here to support our argument that the learnability and memorability of design problems may be suspect.

- The work of a third year group of software engineering students at the University of Glasgow was observed and serves to illustrate the pattern knowledge transfer process. Students were randomly allocated to groups of five to do a project during their third year. The project entailed the design and implementation of a project management system. Students were taught basic software engineering and HCI design patterns and given examples of their use in a graphical user interface. Although a group project, the students were required to write an individual report about what they learnt during the project, including the role of patterns. Only one of the students reported making use of the full complement of patterns (they could use 5 in the exercise). But what is more interesting is that the student’s team members did not report using the same 5 patterns. Even though students had two lectures on patterns, and the lecture notes were also freely available on the module website, only 28% of the students appear to have made use of patterns in their group project. It is possible that students made use of patterns and then did not report it, but this is unlikely because it was an explicitly mentioned topic. The only conclusion we can draw from this is that students had the theoretical knowledge but had difficulty applying it. The discussion on different knowledge levels above offers some explanation for this phenomenon – students master declarative knowledge and, to a lesser extent, procedural knowledge, but they do not develop judgement knowledge.
- Two experiments were conducted on teaching patterns and anti-patterns with third-year Computing Science students at the University of Glasgow: an intra-group study and an inter-group study. The intra-group study found that students had difficulty in applying guidelines stated in the negative, in contrast with guidelines stated in the positive, which resulted in fewer errors. The inter-group study had two groups of students receiving group tutorials separately, either being taught using positive HCI design pattern-like information or anti-pattern like information. Table 1 depicts, as

percentages, the difference between the average scores of the students in the patterns group and those in the anti-patterns group for each of the assessed components. It is clear from the results in this table alone that the students in the patterns group performed significantly better in all of the assessed concepts than did the students in the anti-patterns group. But what is also clear is the extremely low performance even in the group that were taught with patterns, i.e. positively.

**Table 1.** Comparing the marks (as percentages) of students in the anti-pattern group and the pattern group per component

	Use of Colour	Instructions given	Button Design	Error Reporting
Anti-Patterns	39	41	22	46
Patterns	47	54	37	59

The findings of these experiments can be criticised for not focusing on the usability issues directly, and therefore we conducted a survey with another group of 17 third-year Computing Science students at the University of Glasgow focussing specifically on their experiences with patterns. This survey was done within two weeks of their receiving a number of lectures on patterns. When asked *'how easy it was to understand design patterns when first taught'*, 12 of the 17 found it to be difficult, while only 1 thought it was easy. More than half of the students did not understand the rationale behind specific patterns. When asked *'how easy is it to remember patterns that were taught after a week or two'*, the overwhelming response was that it *'was hard'* (only 2 though it was relatively easy). They also had problems in remembering the patterns they were taught the year before. They forgot either the rationale behind the patterns they were taught or the design method it represented, or both.

Evidence from these experiments, and from the theoretical foundations, therefore show that HCI design patterns and anti-patterns could be deficient with respect to learnability and memorability. This leads us to the inescapable conclusion that HCI design patterns and anti-patterns *do not* meet the first two of Nielsen's [43] usability attributes.

In the next three sections we will briefly look at the other three attributes of usability, namely efficiency, errors and satisfaction and consider the extent to which HCI design patterns adhere to these attributes.

## 5 Efficiency

Efficiency refers to the level of productivity, i.e. the resources spent in relation to the accuracy and completeness of the goals achieved [26]. Efficiency also refers to the ways in which a system supports users in carrying out their tasks.

For a pattern language to be efficient in generating solutions it should be generative, allowing users to develop new solutions, and provide a taxonomy enabling the user to easily locate relevant core patterns, to find related or proximal patterns, and to evaluate the problem from different standpoints [19].

The organization of pattern languages in HCI is particularly problematic because of the wide range of different levels that have to be addressed by HCI design patterns, from the broader social context in which an interactive system is used, to the low-level details of interaction itself [14].

Efficiency is therefore related to the completeness of the pattern languages. This is particularly problematic in HCI design patterns, as no coherent pattern language exists. There are a lot of competing voices and individual (and often repeated) efforts [14]. This is often as a result of the demands on researchers to publish and own work. Although pattern language development needs to be a community effort, the competitive pressures within the wider research context can mediate against such a cooperative approach [2].

Unless a collaborative process can be developed in future whereby participants can select and develop the patterns towards a coherent pattern language, HCI design patterns will continue to fail to meet the efficiency usability attribute.

Our experiences [32] suggest that poor knowledge transfer by means of the use of patterns can be attributed directly to the fact that students do not develop judgement knowledge in the short period of time allowed for teaching a concept. Furthermore, we also argued that anti-patterns confused students and did more harm than good.

During our survey amongst the third-year Computing Science students we asked them ‘*how difficult it is to match design problems to the patterns you were taught when you are designing software now?*’ Only 4 of the 17 students found it relatively easy – the other 13 found it very hard. When they were asked whether they ‘*get frustrated when they have to try to find a pattern to match a problem*’, 12 of the 17 expressed dissatisfaction and frustration with matching patterns to problems.

In terms of efficiency and efficacy in knowledge transfer and use, therefore, patterns have yet to prove their worth.

## 6 Errors

When we introduced the concept of patterns in section 2, we referred to two types of patterns, namely *patterns* and *anti-patterns*. There is, however, a third type of pattern, called an *amelioration pattern*. An amelioration anti-pattern tells the reader how to go from a bad solution to a good solution. It defines a migration path (or refactoring) from a negative to a positive solution. It tells you why the bad solution appeared viable in the first place, why it turned out to be bad in conjunction with the desired new outcome or behaviour, and what positive patterns are applicable instead [6]. Amelioration anti-patterns are only required because people fail to locate the correct pattern and then apply the wrong pattern, or, if they do manage to match the correct pattern to the problem, they apply it incorrectly.

The mere existence of amelioration patterns hints at problems with the usability of HCI design patterns. Recovering from a problem should not require the designer to look up a solution from yet another set of HCI design patterns. On the positive side, if an amelioration pattern exists for a specific problem or incorrectly applied solution, it will provide the designer with a ‘way out’ when things go badly wrong or when the designer does not know how to correct an obvious mistake. At present there are, unfortunately, only a small number of amelioration HCI design patterns in existence.



In terms of errors, once again HCI design patterns do *not* prove to be the silver bullet of design – confirming Fred Brooks' [5] prediction that design, being inherently complex and difficult, will never be eased by one particular innovation or tool.

## 7 Satisfaction

Cognitive load is high when designers are working on a project within limited time constraints, and this has been proved to be counter-productive for the interpretation of false or negated information [21, 22], or detailed information requiring the designers to choose between various option (e.g. choosing the correct HCI design pattern for a specific interaction design).

For seasoned designers who have developed judgement and wisdom knowledge this should not be a problem, but for novice designers who are still attaining and developing such knowledge, it might lead to a high degree of dissatisfaction if they cannot easily identify a suitable design pattern. Furthermore it is likely that they simply will not understand how to use it or why it should be used.

Although all but 3 of the students in our survey saw the point of learning patterns, the majority of them (12 of the 17) found patterns to be obscure.

Dearden and Finlay [14] argue that one of the most obvious weaknesses of HCI designs patterns is the lack of substantive evidence as to the benefit of using them in actual design practice. Considerable attention has focused on generating patterns and developing various individual pattern languages, rather than on their use in practice. Significant effort is now required to examine the use of these languages in actual design (e.g. via empirical and observational studies) and in education to demonstrate what, if any, benefits might be gained from a patterns-led approach. We argue that satisfaction levels will stay low until these benefits have been proven.

## 8 Improving Pattern Usability

From the arguments above we have to conclude that HCI design patterns do not meet any of the basic usability principles or attributes. Our investigations have also convinced us that patterns are neither efficient nor efficacious in transferring expert HCI design knowledge to naïve designers.

Should we give up on patterns altogether? Not at all! We should simply be more realistic and circumspect about their use.

We can compare the process of learning how to design systems with language acquisition, albeit on a very superficial level. People learn a new language starting by mimicking particular words. Only once they have accumulated a fair number of commonly used words, and built up a bare framework of the language, and used it for some time, can they start to understand more intricate formalisms such as sentences, tenses and grammar.

Perhaps we can learn a lot from the way schools have changed how they teach in the last 40 odd years. Crystal [13] provides some interesting insights into the changing modes of language instruction. Before the 1960s children were taught grammar – given sentences to analyse in terms of grammatical constructs. Those of us

who experienced this approach often remember it with a sense of repugnance. Grammar was reduced to a set of rules but the meaning and richness of the language was never experienced or understood. Between the 1960s and the mid 1990s children were taught no grammar at all. This too was found to be unsatisfactory because one needs an understanding of grammar to understand the immense creative power of language. A comprehensive study of grammar also helps us to master second and third languages. Consequently, in the late 1990s the approach changed once more, to reintroduce grammar into the curriculum. Only now, a different, more effective paradigm was applied – discovery-based learning. Grammar was no longer merely prescriptive, but was introduced to help students to understand meanings and effects of different constructs in communicating and language. The paradigm was: discovery first, definitions of terms last.

The fact is that we learn in a stepwise fashion, learning rudimentary skills (declarative and procedural) first, then we learn by doing and by watching others more skilled than ourselves (moving towards judgement and implicit procedural skills) and then, only once we have mastered the basics and used them over a period of time, can we be said to have the basic skills to start looking at formalisations such as patterns (once we have the judgement knowledge.)

Someone learning to design interfaces will learn information about basic widgets, and accumulate an understanding of basic HCI principles in a discovery-based way. Only once they are fully conversant with the basic building blocks of the interface can they start thinking about formalisms such as using basic concepts in conjunction with each other to create more complicated artefacts that are, nevertheless, usable. Only once they have spent some time engaged in this process will they be ready for the pattern formalisms and for understanding patterns which bring all the different concepts together in a structured way.

Since we've argued that patterns are contra-indicated for naïve designers, what should we do to direct them and prevent them from making errors? *We should provide them with rules and guidelines, which are easily understood and applied.* We should provide them with a mentor – a seasoned designer to guide their discovery process.

This is not an arbitrary recommendation. There is empirical evidence that guidelines may be easier to use and more effective than patterns [11, 60]. There is little evidence that interfaces produced by using HCI design patterns are better than interfaces designed using guidelines [11]. Koukouletsos, Babak and Dearden [31] also found that patterns, being longer in text and more difficult to assimilate, are harder for novice designers to comprehend. Novice designers need to undertake an extra mental process when contemplating the use of a pattern. Patterns need to be analysed and well understood to be efficacious. Guidelines do not suffer from these problems. The University of California studies in the early 1990's on teaching with or without patterns also confirm this [7, 25, 34, 35, 37, 49]. The group's overall finding was that patterns need rich connections to examples and multiple links to context of use if they were to be effective in teaching. If patterns are too narrow or inflexible, novices have difficulty abstracting from them and would rarely use them. It is generally accepted that the way in which expert programmers work has a great deal more to do with large 'libraries' (patterns) they have built up over time of stereotypical solutions to

problems, as well as strategies for coordinating and composing them, than the mere syntax and semantics of language constructs [53]. If novice students are to mature into expert programmers, they should be taught explicitly about building up these libraries and developing strategies for activating them.

We therefore argue that HCI design patterns should be recorded by experienced designers but should not be inflicted on naïve designers – rather they should be available for use by seasoned designers, those who have attained a particular proficiency in the language of design – much as colloquialisms are understood only by people who have attained a high level of proficiency in a particular language. In the same way, patterns can only really be comprehended and correctly applied by people who have attained a high level of proficiency in the language of design.

## 9 Conclusion

It is clear that HCI design patterns are basically unusable by their currently targeted audience, since they do not exhibit the basic characteristics of usability, as defined by Nielsen [42].

If HCI design patterns were to be representative of the design and development methods promoted for the design of interactive systems then the answer to our question ‘*do we practise what we preach in formulating our design and development methods*’ should be in the negative: and the obvious conclusion should be that we unfortunately do *not* practice what we preach.

Unfortunately this does not apply to HCI design patterns only – the same might be said about design patterns in general, as was exhibited in the University of California studies. As educators and mentors, we should consider these findings carefully and we should be more careful about recommending a technique that we, as experts, find helpful, in the mistaken belief that it will be equally helpful to novices.

## References

1. Anderson, J.R.: Rules of the Mind. Lawrence Erlbaum Associates, Hillsdale (1993)
2. Bayle, E., Bellamy, R., Casaday, G., Erickson, T., Fincher, S., Grinter, B., Gross, B., Lehder, D., Marmolin, H., Moore, B., Potts, C., Skousen, G., Thomas, J.: Putting it all together: Towards a pattern language for interaction. SIGCHI Bulletin 30(1), 17–33 (1998)
3. Berry, D.C.: How Implicit is Implicit Learning? Oxford University Press, Oxford (1997)
4. Borchers, J.A.: Teaching HCI Design Patterns: Experience from Two University Courses. In: Patterns in Practice: A Workshop for UI Designers (at CHI 2002 International Conference on Human Factors of Computing Systems). City (2002)
5. Brooks, F.P.: The Mythical Man Month and Other Essays on Software Engineering. Addison Wesley, Reading (1995)
6. Brown, W.J., Malveau, R.C., McCormick III, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc., New York (1998)
7. Clancy, M.J., Linn, M.C.: Patterns and pedagogy. ACM SIGCSE Bulletin (3/99), 37–42 (1999)

8. Clark, H.H., Chase, W.G.: On the process of comparing sentences against pictures. *Cognitive Psychology* 3, 472–517 (1972)
9. Cockburn, A., Baruz, A., Englund, A., Hanes, P.B., Brown, C., Siska, C., Olson, D., Xexo, G., Lowe, I., Chapman, J., Coplien, J.O., Holloway, J., Brown, K., Eichin, M., Phillips, R., Jeffries, R., Gordon, S., McCormick III, H.W.: Antipattern (2005) [cited 2005-12-12], <http://c2.com/cgi/wiki?AntiPattern>
10. Coplien, J.O.: *Software Patterns*. SIGS Books & Multimedia, New York (1996)
11. Cowley, N.L.O., Wesson, J.L.: An experiment to measure the usefulness of patterns in the interaction design process. In: Costabile, M.F., Paternó, F. (eds.) *INTERACT 2005*. LNCS, vol. 3585, pp. 1142–1145. Springer, Heidelberg (2005)
12. Craik, K.: *The Nature of Explanation*. Cambridge University Press, Cambridge (1943)
13. Crystal, D.: *How Language Works*. Penguin, London (2005)
14. Dearden, A., Finlay, J.: Pattern Languages in HCI: A critical review. *Human-Computer Interaction* 21(1), 49–102 (2006)
15. Dix, A., Finlay, J., Abowd, G.D., Beale, R.: *Human-computer Interaction*. Pearson Education Limited, Harlow (2004)
16. Dix, A., Finlay, J., Abowd, G.D., Beale, R.: *Human-Computer Interaction*, 3rd edn. Pearson Education Ltd., Harlow (2004)
17. Faridul, I.: *Investigating XML as Language for HCI Patterns Representation*. Concordia University, City (2003)
18. Fincher, S.: *The Pattern Gallery* (2000) [cited 2005-12-12], <http://www.cs.kent.ac.uk/people/staff/saf/patterns/gallery.html>
19. Fincher, S., Windsor, P.: Why patterns are not enough: some suggestions concerning an organising principle for patterns of UI design. In: *CHI 2000 Workshop on Pattern Languages for Interaction Design: Building Momentum* (2000)
20. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (1995)
21. Gilbert, D.T., Krull, D.S., Malone, P.S.: Unbelieving the unbelievable: some problems in the rejection of false information. *Journal of Personality and Social Psychology* 59, 601–613 (1990)
22. Gilbert, D.T., Tafarodi, R.W., Malone, P.S.: You cannot believe everything you read. *Journal of Personality and Social Psychology* 65, 221–233 (1993)
23. Gorman, M.E.: Types of knowledge and their roles in technology transfer. *Journal of Technology Transfer* 27(3), 219–231 (2002)
24. Hartson, H.R., Hix, D.: Toward empirically derived methodologies and tools for human-computer interface development. *International Journal of Man-Machine Studies* 31, 477–494 (1989)
25. Hoadley, C.M., Linn, M.C., Mann, L.M., Clancy, M.J. (eds.): When, why, and how do novice programmers reuse code? In: Gray, W.D., Boehm-Davis, D. (eds.) *Empirical Studies of Programmers*, vol. 6. Ablex, Norwood (1996)
26. International Organization for Standardization: ISO9241: Ergonomic requirements for office work with visual display terminals (VDTs) (1997) [cited 2006-12-01], <http://www.iso.org/iso/en/iso9000-14000/index.html>
27. International Organization for Standardization: ISO14915: Software ergonomics for multimedia user interfaces (2002) [cited 2006-12-01], <http://www.iso.org/iso/en/iso9000-14000/index.html>
28. Johnson-Laird, P.N.: *Mental Models*. In: Posner, M.J. (ed.) *Foundations of Cognitive Science*, pp. 469–499. MIT Press, Cambridge (1989)

29. Johnson-Laird, P.N., Girotto, V., Legrenzi, P.: *Mental Models: A Gentle Guide for Outsiders* (1998), <http://www.si.umich.edu/ICOS/gentleintro.html>
30. Kirkhart, M.W.: The nature of declarative and nondeclarative knowledge for implicit and explicit learning. *The Journal of General Psychology* 128(4), 447–461 (2001)
31. Kotzé, P., Renaud, K., Koukoultsos, K., Khazaei, B., Dearden, A.: Patterns, anti-patterns and guidelines: Effective aids to teaching HCI principles? In: Hvannberg, E.T., Read, J.C., Bannon, L., Kotzé, P., Wong, W. (eds.) *Inventivity: Teaching theory, design and innovation in HCI - Proceedings of HCIED2006-1 (First Joint BCS / IFIP WG 13.1 / ICS /EU CONVIVIO HCI Educators Workshop*, pp. 115–120. University of Limerick, Limerick (2006)
32. Kotzé, P., Renaud, K., Van Biljon, J.: Don't do this - Pitfalls in using anti-patterns in teaching human-computer interaction principles. *Computer & Education* (2006), doi:10.1016/j.compedu.2006.10.003
33. Lebiere, C., Wallach, D., Taatgen, N.: Implicit and explicit learning in ACT-R. In: Ritter, F., Young, R. (eds.) *Proceedings of the Second Conference on Cognitive Modelling (ECCM 1998)*, pp. 183–189 (1998)
34. Linn, M.C.: How can hypermedia tools help teach programming? *Learning and Instruction* 2, 119–139 (1992)
35. Linn, M.C., Clancy, M.J.: The case for case studies of programming problems. *Communications of the ACM* 35(3), 121–132 (1992)
36. Mahernoff, M.J., Johnston, L.J.: Principles for a usability-oriented pattern language. In: *Proceedings of the Australasian Computer Human Interaction Conference*, Adelaide, pp. 132–139 (1998)
37. Mann, L.M.: *The Implications of Functional and Structural Knowledge Representations for Novice Programmers*. In: Graduate Group in Science and Mathematics Education. University of California, City (1991)
38. Mayhew, D.J.: *Principles and Guidelines in Software and User Interface Design*. Prentice Hall, Englewood Cliffs (1992)
39. Mayhew, D.J.: *The Usability Engineering Lifecycle*. Morgan Kaufmann, San Francisco (1999)
40. Mayo, R., Schul, Y., Burnstein, E.: I am not guilty vs I am innocent: Successful negation may depend on the schema used for its encoding. *Journal of Experimental Psychology* 40, 433–449 (2003)
41. Miller, G.E.: The assessment of clinical skills/competence/performance. *Acad. Med.* 65, 563–567 (1990)
42. Nielsen, J.: *Usability Engineering*. Academic Press, Boston (1993)
43. Nielsen, J.: Heuristic evaluation. In: Nielsen, J., Mack, R.L. (eds.) *Usability Inspection Methods*. John Wiley & Sons, New York (1994)
44. Norman, D.: *The Design of Everyday Things*. MIT Press, London (1998)
45. Polanyi, M.: *Personal Knowledge - Towards a Post-Critical Philosophy*. Routledge and Kegan Paul, London (1958)
46. Preece, J., Rogers, Y., Sharp, H.: *Interaction Design: Beyond Human-computer Interaction*. John Wiley & Sons, Inc., New York (2002)
47. Reber, A.: *Implicit learning and tacit knowledge*. Oxford University Press, Oxford (1993)
48. Schach, S.R.: *Object-oriented and Classical Software Engineering*, 6th edn. McGraw Hill Higher Education, New York (2005)
49. Schank, P.K., Linn, M.C., Clancy, M.J.: Supporting Pascal programming with an on-line template library and case studies. *International Journal of Man-machine Studies* 38, 1031–1048 (1993)

50. Seffah, A.: Learning the ropes: human-centered design skills and patterns for software engineers education. *Interactions* 10(5), 36–45 (2003)
51. Shneiderman, B.: *Designing the User Interface*. Addison-Wesley, New York (1998)
52. Smith, S.L., Mosier, J.N.: Guidelines for designing user interface software. Mitre Corporation Report, MTR-9420 (1984)
53. Soloway, E.: Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM* 29(9), 850–858 (1986)
54. Taatgen, N.A.: Learning without limits: from problem solving towards a unified theory of learning (1999) [cited 2005-06-05], <http://www.ub.rug.nl/eldoc/dis/ppsw/n.a.taatgen/>
55. Tidwell, J.: *Designing Interfaces: Patterns for Effective Interaction Design* (2005) [cited 2006-07-01], <http://designinginterfaces.com/>
56. Van Biljon, J., Kotzé, P., Renaud, K., McGee, M., Seffah, A.: The use of anti-patterns in human computer interaction: wise or ill-advised? In: Marsden, G., Kotzé, P., Adesina-Ojo, A. (eds.) *Fulfilling the promise of ICT, SAICSIT (ACM Conference Proceedings Series)*, Pretoria, pp. 176–185 (2004)
57. Van Welie, M.: *Patterns in Interaction Design* (2006) [cited 2006-07-01], <http://www.welie.com/>
58. Van Welie, M., Van Der Veer, G.C.: Pattern languages in interaction design: structure and organization. In: Rauterberg, M., Menozzi, M., Wesson, J. (eds.) *Human-computer interaction, INTERACT- 2003*, pp. 527–543. IOS Press, Amsterdam (2003)
59. Wass, V., Van Der Vleuten, C., Shatzer, J., Jones, R.: Assessment of Clinical Competence. *The Lancet* 357, 945–949 (2001)
60. Wesson, J., Cowley, N.L.O.: Designing with patterns: Possibilities and pitfalls. In: *2nd Workshop on Software and Usability Cross-Pollination: The Role of Usability Patterns* (2003) [cited 2005-12-23], <http://wwwswt.informatik.uni-rostock.de/deutsch/Interact/05WessonCowley.pdf>

## Questions

**Michael Harrison and Janet Wesson:**

*Question: About the Experiment: How do you measure the quality of the resulting product?*

Answer: We checked the product for obvious mistakes such as violation of guidelines, mismatch of colours, etc...

**Michael Harrison:**

*Question: How did you train the students in the use of patterns?*

Answer: Students were introduced to patterns, as is general practice, during lectures. To ensure that they understood how to apply and use the patterns, students were instructed to use them in a design exercise.

**Laurence Nigay:**

*Question: Were the inspected patterns specific to HCI?*

Answer: Yes, but the discovered flaws and limitations may also be applicable to patterns in other domains (e.g. software design).

*Question: Why is it a problem to use patterns from different languages?*

Answer: Different collections may contain patterns for the same problem, but with different (even contradicting) solutions. This can be explained by the fact that the solution stated in the pattern is bound to the overall context of the language.

***Kirstin Kohler:***

*Question: What makes you think that the problem is the way patterns are written and not the way you teach them to students?*

Answer: The teaching method appears to be representative of the methods used by most Universities to teach patterns. Those practitioners who do not attend classes usually attempt to learn patterns from a textbook. Transferring knowledge by means of patterns is the real issue, which is the core of what we are saying. People can learn a pattern as a kind of recipe to be followed, but matching that pattern to a problem is something which cannot be taught - it only develops with experience.

# Engaging Patterns: Challenges and Means Shown by an Example

Sabine Niebuhr, Kirstin Kohler, and Christian Graf

Fraunhofer Institute for Experimental Software Engineering  
Fraunhofer-Platz 1  
67663 Kaiserslautern, Germany  
{kohler,niebuhr,grafc}@iese.fraunhofer.de

**Abstract.** This paper presents first results of a research project whose goal is to develop a pattern language that enhances business software by motivating and engaging elements. The goal of the pattern language is to turn the soft and vague term of “emotions in user interaction design” into constructive design guidance. The patterns are especially tailored for joy-of-use in business applications. The main contribution of this paper is the description of quality characteristics for this pattern language. They are illustrated by references to existing pattern descriptions and elaborating their deficiencies. This paper shows how these weaknesses were addressed in the pattern language.

**ACM Classification Keywords:** D.2.1 Requirements/Specifications, D.2.2 Design Tools and Techniques, H.5.2 User Interfaces.

## 1 Introduction

Using patterns (originally introduced in architecture [1, 2]) for developing software is well established [3] and still up-to-date [4]: Why reinvent the wheel if solutions for a problem are already known and approved? Many pattern languages exist for nearly every developing step – e.g., for designing the interaction and the user interface [5-7], or for the software implementation [3]. But for a software developer, applying patterns is not as simple as one might assume.

Let us imagine a software developer who wants to design a user interface. He has found some interaction patterns on the Web and hopes they will help him. Trying to apply these patterns he first has to find an appropriate pattern. This is a big problem to overcome, since matching a specific design problem to the problem descriptions in existing patterns is a question of interpretation. After the software developer is convinced that the pattern he has identified matches his problem, he tries to understand the author’s recommendations – how does the author think this problem can be solved? The software developer might not see the correlation between the problem statement and the solution described in the pattern: the problem statement matches his problem, but the solution does not make any sense to him. After interpreting the recommendation our software designer applies the pattern in the way he thinks it would be the author’s intention.



Let's assume that the software developer has another problem and therefore searches for a fitting pattern a second time: he finds two patterns that are nearly the same – so which one does he have to apply? How do they relate to each other? Does one specialize the other? Do they have different conditions when to apply? The software developer might not be a very patient person, so he stops searching for another pattern and tries his best on his own – without any guidance or implementation advice. What went wrong? – The pattern descriptions were not concrete enough. – The developer did not find the right pattern to apply. – The developer did not understand the pattern idea.

These are just three problems. For us as authors of patterns, this means: Do not repeat existing defects. Identify the developer's problems with patterns and fix them!

In our project, we try to identify patterns to enhance business software by elements that motivate and engage. In writing down these patterns, several challenges had to be mastered– from setting up a pattern language with all of its elements and relations up to the internal validation of the patterns and the problem of making it discoverable. When we performed a search in the literature, we mostly found solutions to the syntax problems of a pattern language - how to build up relations and designed meaningful elements – but no answers to our semantic questions, for example, how we can formulate our patterns in an understandable way. We found the Pattern Language Meta Language (PLML) [8], and we found approaches that name our challenges – e.g., Meta Patterns [9], patterns for writing patterns – but we did not find any real solution for our problems (we will discuss this in chapters 3 and 4).

In this paper, we demonstrate our challenges and how we mastered them with an example pattern. Our contribution consists of defining quality characteristics for pattern languages that base on our challenges and approaches to master them.

We will first describe our project context to give you an idea of our work: writing engaging patterns. Then we will describe the challenges that came up while writing these patterns, which led us to quality elements. Since we think it would be easier to understand how we mastered the challenges by reading examples, we introduce an excerpt of our pattern language, which is still work in progress. Finally we present our approaches for mastering the described challenges and what we will be doing next.

## 2 Project Context

The work presented here is part of a three-year research project funded by the German federal government entitled 'FUN' (acronym for “**fun-of-use in Geschäfts**anwendungen”).<sup>1</sup> In the project, three industrial partners and Fraunhofer IESE deal with the topic of “fun-of-use for business applications”. One goal of the project is to develop a pattern library that captures fun-of-use interaction pattern. The research work is closely related to the needs of the industrial partners in order to ensure the usefulness of the results for industry.

The challenges given for the pattern library are motivated by our project context: As part of the project, a call center software has to be redesigned in order to improve users engagement with the software. The software helps agents to solve incoming support calls from people complaining about trouble they have with the product. The

---

<sup>1</sup> You can find more detailed information about the project at <http://www.fun-of-use.de>

work of the agents is kind of frustrating and monotone, which results in a loss of motivation. As a consequence, agents are inefficient, make more mistakes, and take fewer calls.

In the first step of the project, we were looking for existing interaction patterns, that might help us to solve the problems of the call center agents as described above. We found two promising candidates: the status display [5] (listed in Table 1) and the high score list [10]. While searching for patterns and applying them to the software described above, we start doubting that a “software engineer” would have been successful in doing this. We are experienced user interface designers/usability specialists well familiar with the concept of “interaction patterns”. Would a software engineer have found the high score list or status display pattern and would he/she have been able to derive an adequate solution for the software from the description? We turned this impression into a challenge for our project. We investigated effort in extracting “quality requirements” for the pattern library. These quality requirements or challenges will be elaborated in the next section.

### 3 Quality Challenges for Pattern Languages

We set up a list of characteristics that we believe are required to support software engineers in creating “engaging” user interfaces. To provide valuable support, our pattern collection has to assist the engineer during the following steps:

**Step A - Pattern Discovery:** The engineer has to find a pattern to the given user interaction problem. The library is intended for software engineers respectively requirements engineers, who design the user interaction as part of the requirements phase. We assume that they follow a task-oriented approach, which means the requirements for the system to be developed are stated as “tasks”. In addition, “non-functional requirements” or business goals are part of the requirements.

**Step B - Pattern Application:** During this step, the software engineer has to apply the solution given by the pattern, which is often still on a quite abstract level, to a concrete interaction realization.

Looking at these two steps in more detail, we identified a set of four quality requirements for our pattern language. These requirements consider quality needs stated by other authors [11, 12], but extend and combine them to address all the problems we investigated. We will explain them by expanding the problems we faced in our project.

#### 3.1 Problem Fit

The pattern language has to guide the user from the problem to the solution; the pattern should be stated in a way that the user can match his problem and project context to the pattern description. This might, on the one hand be a problem of the entire pattern language; the way the pattern are linked or put into hierarchies might not be useful for the engineer. And/or it might be a problem of the individual pattern itself – the pattern description does not give a clue to the real world problem.

The problem in our case was described by the information given in the use case description of the requirements document and the “undesired” behavior of the agent “losing motivation” (which is derived from the business goal “improve agents’ job satisfaction”). The existing description of the “status display” does not give any idea that it might improve the agents’ motivation.

### 3.2 Understandability

This challenge belongs to steps A and B. The wording and notation of the pattern description has to be understandable for the engineer; otherwise, he will neither be able to identify nor to apply the pattern. What does this mean more concretely?

The reader should interpret the words that describe our pattern in such a way, that he understands the idea behind it and the intention we as authors had in writing this pattern. This means that we have to write unambiguously, so that the reader will not misinterpret the content, and we have to write completely and without contradictions, in order to avoid different interpretations. Here the challenge is: How can we ensure this?

Understandability is also closely related to readability. So another aspect is a syntactical aspect, which supports the readability and understandability of our patterns: the elements that describe them. Therefore, we searched in literature and found PLML [8], on which many people worked for gaining a uniformed, standardized Pattern Language. This is a very helpful aspect indeed: The reader gets patterns formulated in the same pattern language, so he knows where to find the context, the problem and the solution. But this approach is not really finished: Many people are still working on this language. However, although definitions for elements and how to fill these elements exist, they are not sufficiently defined, leaving out which kind of content can be found in the “context” element and which in the “problem” element. What would solve this problem?

### 3.3 Correctness

We want to describe patterns that will motivate or engage users. How can one ensure that the desired effect of a user’s engagement or motivation really takes place? Is there any theoretical background that guarantees that the given solution (such as the status display) encourages users to continue their task? Does showing status information really influence the users’ motivation? Todd et al. [11] talk about the “internal validity” of an individual pattern. We define it as the relationship between the description of the problem and the solution: The solution must solve the problem in the given context.

For a lot of described patterns, the way the patterns are phrased makes this step trivial. For example, the problem of the status display is expressed as “How can the artifact best show the state information to the user”, the solution says “Choose well-designed displays for the information to be shown...”. The topic of our pattern language covers emotional effects (like motivation, engagement, fun) and therefore makes it more important to either empirically prove the evidence between “Problem” and “Solution” of a pattern or relate it to one or more psychological theories.

### 3.4 Concretization

Assuming he had found the problem, the task of the developer would be to transfer the pattern description, which is quite abstract, to a concrete solution for the call center software. How can one ensure that this concretization still solves the problem? There are often minor differences in design that make a big difference in the desired effect.

Assuming that while detailing out the user interface for our call center someone had the idea of putting in a kind of “ranking” that shows the performance of each agent compared to the others in terms of “time to fix a support call”. At first glance, one can assume that this kind of ranking would lead to competition between the agents and keeps them motivated. And on the abstract level of a pattern, this assumption might be right in terms of Correctness. Unfortunately, this solution destroys the social relationship between agents and enforces the “Galley Slave Model” [13]. As a consequence dissatisfaction and turn-over of agents increase. As stated before, the intention of the user interface redesign was to increase agents’ satisfaction. Another problem with concretization is that a software engineer reading the “status display” as it is might not even have an idea, what range of freedom he has in bringing it to a concrete solution – showing a “progress bar” is not the only way of representing “status”, as we will show in the next section.

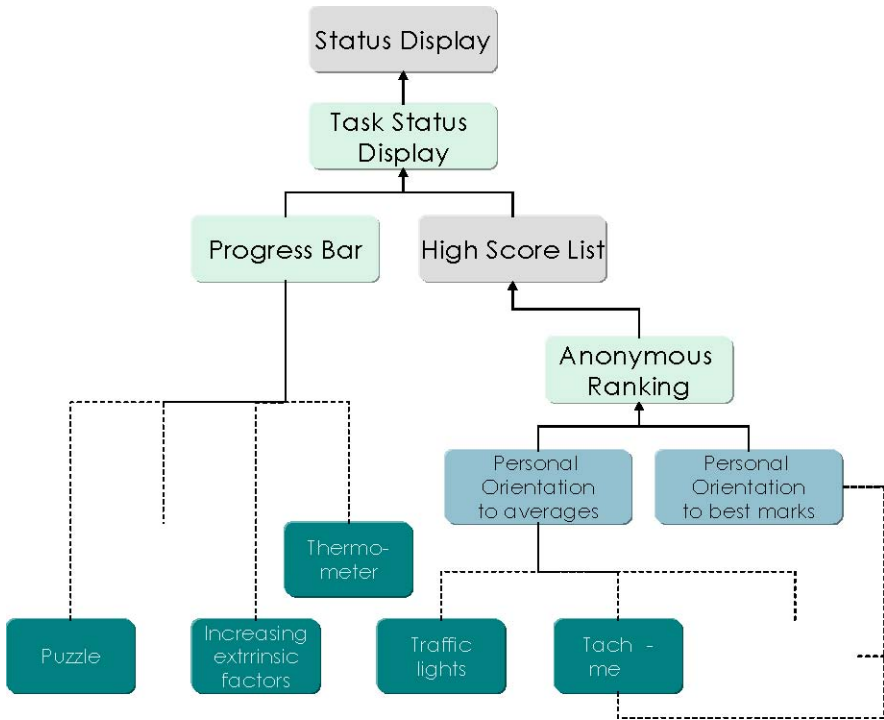
The first two quality requirements (Problem Fit and Understandability) address step 1, “find a pattern”, whereas step 2 is related to the quality requirements Understandability, Correctness, and Concretization.

## 4 Engaging Patterns

We would not have been able to concretize these problems if we did not have the idea of writing down patterns to support developers in designing and implementing user interfaces containing motivating elements – elements that help users stay concentrated on their work tasks. For detecting patterns that engage we looked into existing pattern languages as well as into the literature for e-learning and game design. Especially in these disciplines, much time has been spent on developing applications that capture the user, because these applications depend on the user keep on using them voluntarily. We now try to apply this knowledge to business application design.

Some of our engaging patterns can be specialized from the existing usability pattern “Status Display” (see Table 1), established by Jennifer Tidwell in [5]. An overview of the patterns that could be specialized from Tidwell’s “Status Display” is given in Figure 1. “Status Display” and “High Score List” are patterns described in the literature, “Task Status Display”, “Progress Bar”, and “Anonymous Ranking” cover patterns specialized by us, boxes building the leaves of this tree are examples for concrete implementations.

The pattern “Task Status Display” proposes a solution for showing any kind of information concerning the user’s task. The pattern “Progress Bar” as a specialized “Status Display” shows this information in relation to a specific goal. The pattern “High Score List” (this comes out of game design) shows information concerning the work task (for example, performance data) as a specialized status display in relation to other performance data. This data can show performance of other people, statistical values, or values that should be achieved.



**Fig. 1.** Hierarchy of Status Display patterns with examples of concrete implementations

A specialized “High Score List” is an “Anonymous Ranking”. Normally, in high score lists, names mark the presented information. This could cause some group effects or discouraging effects, so in some applications, names should not be mentioned. The idea of this pattern can be specialized in a personal ranking – a personal orientation from which the user gets information about his personal performance data related to an average value or related to personal or group-wide best marks.



**Fig. 2.** Different solutions for the “Progress Bar” to display the task status: a) as traffic light, b) as card stack c) as a puzzle

To give a better idea of how these patterns can be implemented, we display some concrete examples: In the first example, an “Anonymous Ranking” is implemented as a traffic light (see Figure 2a). A “Progress Bar” could be implemented as increasing or decreasing volume, for example as a card stack (see Figure 2b). The picture originates from an application where the user has to fill in an address database. Every time he enters an address, the set of cards in the picture is reduced by one card. Another example is the idea of a puzzle, like the example in Figure 2c), which originates from a computer configuration tool. The puzzle completes a little more every time a user adds one part to a computer. In some companies the employees receive certain incentives – extrinsic motivating values – which can be visualized by a progress bar (see Figure 3).



**Fig. 3.** The Progress Bar displays the status plus the rewards that can be expected when reaching certain degrees of completion

In the following, you will read more about approaches we found to master the challenges encountered while writing down these patterns.

#### 4.1 Problem Fit

To guide the engineer from his “real world” problem to the pattern solution, our pattern library followed two strategies:

- The hierarchy of patterns (given by the relationship between them) within the pattern language and
- The pattern description of individual patterns.

The hierarchy of patterns guides the engineer from more general patterns to more specific patterns. This helps to “narrow down” the appropriate patterns by matching them to the various context/problem fields of more specific patterns. Figure 1 illustrates this hierarchy for an excerpt of our pattern language.

The second strategy to improve “problem fit” covers the pattern description of individual patterns. By giving the descriptions of single pattern elements a more specific semantic pattern can be integrated into a task-oriented requirements approach. This facilitates the “detection” of the appropriate pattern in a natural way. The engineer matches the requirements given by the project to the problem and context section of the pattern descriptions. This means in more detail:

- Individual pattern state the non-functional requirement they contribute to.
- The engineer should be able to match these non-functional requirements to the business goals that characterize his project.
- The context of a pattern contains fields characterizing the user type, the task, the environment, and all the elements that belong to a contextual design. By specifying the context as “completely” as possible, we try to prevent the engineer from applying a pattern that does not fit the “real world” problem.

## 4.2 Understandability

The first question is: How can we formulate patterns unambiguously? Meszaros and Doble propose to find out who the audience is and to focus on it with wording and notation [9]. This is a helpful approach, but it is not sufficient for solving our problem: We have software developers who (hopefully) will implement our patterns as well as psychologists or graphic designers. By describing several interactions through the use of UML activity diagrams, the software developer gets an exact idea of how to solve the problem, whereas the graphic designer just reads some strange symbols. Thus, for usability aspects we have a broad audience. To ensure that every reader will understand our ideas behind the patterns, we will have to use natural language, which is often ambiguous or badly structured.

The solution of the “Status Display” pattern starts with a sentence in natural language: “*Choose well-designed displays for the information to be shown*”. What is meant by “well-designed” and which information should be displayed? This example was just the first sentence of the pattern’s solution.

In software engineering, the same problem of a broad audience exists at the beginning of a software project: Requirements for this project have to be defined and written down in a way that guarantees understandability for the software developer as well as for the customer. And this customer might be a dentist or a mechanic, with totally different knowledge and background. Rupp and Götz [14] dealt with this topic in requirements engineering and identified three main problems of natural language used for defining requirements: distortion, generalization, and deletion.

A whole process described as a single event in the textual description leads to distortion and misinterpretation. The problem of generalization can be described as trying to derive a more general description based on your experience while neglecting exceptions. Deletion often occurs when information expected to be well-known by everyone is left out. Therefore, Rupp and Götz propose rules to detect these problems and delete them. One way to keep it simple from the beginning is to use some structured sentences, a pattern for building sentences, which aids readability. Now we propose to use these rules and structured methods that exist for writing down requirements to write down the content in our patterns unambiguously, completely, and without contradictions.

Coming back to the “Status Display” example, we would formulate the solution a little bit more concretely (see “Task Status Display” in Table 3): “Display the task’s state information. [...]. Display the information the user needs at a glance.”

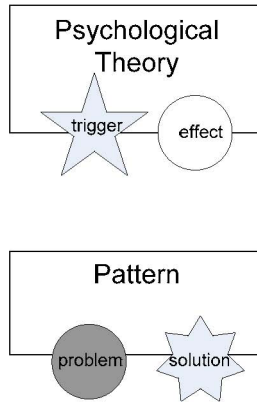
Another helpful thing to prevent misinterpretation is to keep the vocabulary constant and simple. Sure, normally it is good style to call the user “user” the first time, “driver” the second time, and else third time something to avoid repeating the words too often. But the reader may ask, whether there are three different users. So why don’t we call our user – if he is a driver – a driver every time we talk about him? It does not sound very nice, but it increases readability. This is why the sentences in our pattern descriptions always look the same: “Display...Display...Display...” instead of “show... paint... draw...display...”

Let us now proceed from the vocabulary aspect to the syntactical aspect that assists readability and understandability of our patterns: The elements described in PLML [8] should be defined more exactly. They should be differentiated to make clear

which content can be found in a specific element - especially the element “context” and “problem”. For finding a pattern, both elements have to be read, but the first look should be focused on the problem. This semantic lack in document based pattern is a reason to push ontology based infrastructures for patterns (e.g., BORE [15]).

### 4.3 Correctness

We want to ensure that our patterns are correct, meaning the solution described as part of the pattern solves the problem given in the problem field. We try to achieve this quality characteristic by rationalizing the pattern with psychological theories. Most of these theories describe relationships between triggers and effects. We conducted a literature survey as part of our project, scanning theories that describe triggers for positive emotional reactions like motivation, creativity, and fun. The triggers specified by such theories have to be related to the “solution” part of the patterns. If pattern solutions are design examples for such triggers, they might lead to the desired effect specified in the theory. For our engaging patterns this means: If the pattern covers a theory which is validated, we know that a software system which includes this pattern is more engaging than without. As a consequence of the effect, the problem stated in the pattern is solved. Figure 5 illustrates this in an abstract way. Effect and problem are related (indicated by circles but in different colors, because the problem is the “negation” of the effect) and the trigger and solution are associated (indicated by the star),



**Fig. 4.** Relationship between psychological theories and pattern description

To guarantee correctness in the case of the (task) status display, we will consult two different theories that back this approach with psychological reasoning.

Herzberg’s two-factor theory proposes that after having compensated for all the un motivating factors at the workplace (like uncomfortable workspace, bad relationship with the boss etc.) a person will be in an equilibrium, a neutral state [16]. Beginning in that state, one might try to gain satisfaction through ‘motivators’ while at work (this is the desired “effect”). Some of these motivators are: performance, being responsible, pay, or promotion (these are the “triggers”).



**Table 1.** The pattern “status display” as found in [5]

Name	Status Display
Context	The artifact must display state information that is likely to change over time, especially if that state information represents many variables.
Problem	How can the artifact best show the state information to the user?
Forces	<ul style="list-style-type: none"> <li>– The user wants one place where he knows he can find this state information.</li> <li>– The information about it should be organized well enough so that the user can find what the needs at a glance, and can interpret it appropriately.</li> <li>– It needs to be unobtrusive if the information is not critically important, but...</li> <li>– It does need to be obtrusive if something important happens.</li> </ul>
Solution	Choose well-designed displays for the information to be shown. Put them together in a way that emphasizes the important things, deemphasizes the trivial, doesn't hide or obscure anything, and prevents confusing one piece of information with another. Never rearrange it, unless the user does it himself. Call attention to important information with bright color, blinking or motion, sound, or all three -but use a technique appropriate for the actual importance of the situation to the user
Resulting Context	If there is a large set of homogeneous information, use High-density Information Display and the patterns that support it (Hierarchical Set, Tabular Set, Chart or Graph); if you have a value that is binary or is one of a small set of possible values, use Choice from a Small Set. Visually group together discrete items that form a logical group (Small Groups of Related Things), and do this at several levels if you have to. For example, date and time are usually found in the same place. Tiled Working Surfaces often works well with a Status Display, since it hides nothing -- the user does not need to do any window manipulation to see what they need to see. (You might even let the users rearrange the Status Display to suit their needs, using Personal Object Space.) If you don't have the space to describe what each of the displayed variables are (e.g., Background Posture), or if your users are generally experts who don't need to be told (e.g., Sovereign Posture), then use Short Description to tell the users what they are.

The second supporting theory is the goal setting theory [17]. The central statements of this highly recognized and empirically proven theory are as follows:

- Setting goals that are difficult to achieve leads to higher performance than the setting of easy goals.
- Setting specific goals leads to higher performance than the setting of vague, unspecified or no goals.

**Table 2.** The “Status Display” pattern explicated for one task

Name	Task Status Display
Context	The user wants to fulfill a task. The artifact must display state information that is likely to change over time, especially if that state information represents many variables.
Problem	The user needs an orientation on how far he has come with his task.
Forces	– The user wants to see the task’s state information. – The state information should display information the user needs at a glance. – The state information should be appropriately interpretable. – If the information is not critically, the state information should be too unobtrusive. – If the information is critically, the state information should be obtrusive. – Information is critically, if something important happens.
Solution	– Display the task’s state information. – Always display the information in the same place. – Display information the user needs at a glance. – Display the state information in an appropriately interpretable way. – If the information is not critical, display the state information unobtrusively. – If the information is critical, display the state information obtrusively.
Rational	Herzberg’s two-factor theory [16]; Goal setting theory (Schmidt & Kleinberg 1999)
Resulting Context	The user gets orientation on how far he has come with his task. The user is able to estimate his task status.

Both statements have been supported widely by other researchers and are known to have high external validity, i.e., findings can be transferred to diverse settings, like groups and single persons, different task types, and different cultures [17, 18]. The most important factor in this respect is the complexity of the task. The completion of an easy task can be more successfully supported by goal setting than that of a difficult task. This results from different effects. One is that complex tasks need more efforts and take longer so that the effect of the single effort is not directly visible as performance.

Complementing the goal setting, giving feedback is recognized as an important factor [19]. Feedback transfers information back to the user, so that he knows what he has achieved and how he might possibly adjust his actions. Feedback can motivate because the person notices that earlier set goals have been achieved and this tendency will hopefully last. This results in ongoing or even increase motivation.

Applying either goal setting or feedback might not necessarily result in any performance increase. The maximum effect is reached when combining compulsory goals and related feedback [20].

**Table 3.** The “Progress Bar” pattern [23]

Name	Progress Bar
Context	The user is working on a <b>task</b> . The user knows the task’s <b>goal</b> . An employee has to achieve different goals at <b>work</b> . The work has <b>one or more defined goals</b> . The work can be <b>dreary</b> or <b>long lasting</b> . An employee has to fulfill different <b>tasks</b> at work. The task has one ore more defined goals. The task can be dreary or long lasting.
Problem	The user loses sight of the goal. The user needs to be <b>reminded what the goal is</b> about.
Forces	See forces from the pattern “Status Display”. Additionally: – The displayed information should contain the goal. – The displayed information should contain the distance to the goal. – The displayed information should contain the scale of the movement into a direction. – The displayed information should contain the starting point. – The displayed information should contain the distance to the starting point. – The information should contain if the user draws nearer to the goal.
Solution	See the solution from pattern “Status Display”. Additional: – Display the <b>task</b> . – Display <b>goal</b> . – Display the <b>starting point</b> . – Display the <b>distance from the starting point</b> . – Display the <b>distance to the goal</b> . – Display the scale of the movement into a direction ( <b>step width</b> ).

#### 4.4 Concretization

The challenge of concretization is addressed by two contributions:

The problem is on a higher level of abstraction than the solution description. This means the solution summarizes design decisions and is therefore closer to the final solution than the given problem. We show a large variety of different concretizations for a given pattern. As one possible concretization for the “progress pattern”, we have several very different examples as shown in Figures 2-5. This should open the engineer’s thinking to further creative concretizations of the same problem. At the same time, it already provides such a wide range that it might be easy to simply pick one of the solutions

- By working out a variety of different concretizations, we were able to state the commonalities between the variants more clearly. This helped us to make the description of the solution more precise. For the solution part of the “progress pattern” is very precise in listing the user interface elements that have to be defined. It lists elements like “task”, “goal”, “starting point” etc. All these are variables the engineers has to define through concrete values when developing a concrete user interface solution. The likelihood that a engineer derives a solution from this description, which is not a correct concretization of the “progress bar”, is very small.

- While building the pattern collection we order pattern in a hierarchical manner from more abstract “task levels” down to detailed “user interface levels”. Beside the problem of concretization this facilitates the linkage from the requirements phase (which is task or use case oriented) to the concrete user interface design solution. With this approach we built on concepts introduced by Mahemoff and Johnston [21] and the PSA-Framework [22].
- Display the **direction** of the movement (if the user draws nearer to the goal) Resulting Context

The user won't lose track of the goal.

The user can see if he draws nearer to this goal.

The user can see how far he is away from the goal.

The user can see how far he is away from the starting point.

The user is able to estimate his work progress from this data.

The user is able to estimate the remaining time.

## 5 Next Steps

After having identified promising approaches from other disciplines that have proven to engage users, we will conduct empirical studies that investigate how well these ideas were transformed into effective means for motivating in the particular context – into high quality patterns that work.

With each specific implementation of an idea, we will undergo a thorough validation process. The process will consist of two phases: First, we are going to check in a laboratory setting if the result of the particular implementation of a pattern satisfies the “intended outcome” section of the pattern description. If the result is as intended the pattern can be viewed as valid (for this context). Second, the pattern will be tested in a field study with a group of real users. These users will be from the target audience of the enhanced application and will be trained to work with a basic version of the application. Thus we want to avoid effects of curiosity or learning effects that might distort or spoil the result of the analysis. In the field study, we want to learn if the application can transfer its motivational nature to the target audience. It will show whether the realizations of patterns are understood and up to what level of abstraction (as some patterns are very basic - e.g. the status pattern - others are more high-level).

With the results from the first evaluation, we are planning to try out other patterns originating from the games context or e-learning context. We expect that not all ideas from those specific contexts will be beneficial in the target domain. As a result, a pattern language with multiple relations like “contributes to”, “is supported by” or “is suspended by” will evolve for the domain of information services.

Having learned about patterns in one domain it will be challenging to look for possible transfer into other domains in the same way as interaction patterns [7, 24] can be found in different domains like the Web [25, 26] or mobile devices [27]. That question will be a topic of future research.

One practical aspect of our research – current and upcoming – is the process integration of the present and future patterns into the daily work of software engineers. We strive for a beneficial, yet easy, handling of patterns in the context of use. To support developers, we have started the development of a plug-in for the Eclipse Framework

([www.eclipse.org](http://www.eclipse.org)). As an open source platform with a thriving community, it is highly suitable for an effort such as deploying and actively developing a pattern library. Let developers and users of software be engaged by patterns that engage!

## Acknowledgements

This work is supported by the German Federal Ministry of Education and Research (BMBF) within the project FUN (Grant: 01 IS E06 A). For more information see the project website <http://www.fun-of-use.de>. We wish to acknowledge the contribution of our project partner a3 systems GmbH ([www.a3systems.com](http://www.a3systems.com)). Some of the patterns presented were conceptually designed with them. They also contributed the pattern implementation in a real-world business application.

## References

- [1] Alexander, C., Ishikawa, S., Silverstein, M.: *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Oxford (1977)
- [2] Alexander, C.: *The Timeless Way of Building*. Oxford University Press, Oxford (1979)
- [3] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, Boston (1994)
- [4] Gamma, E.: *Design patterns: ten years later*. In: *Software pioneers: contributions to software engineering*, pp. 688–700. Springer, New York (2002)
- [5] Tidwell, J.: *COMMON GROUND: A Pattern Language for Human-Computer Interface Design*, vol. 2006 (1999)
- [6] van Welie, M.: *The Amsterdam Collection of Patterns in User Interface Design*, vol. 2006 (1999)
- [7] Borchers, J.: *A Pattern Approach to Interaction Design*. John Wiley & Sons, Ltd., Chichester (2001)
- [8] Fincher, S.: *CHI 2003 Workshop Report - Perspective on HCI Patterns: Concepts and tools (introducing PLML)*. *Interfaces* 56, 27–28 (2003)
- [9] Meszaros, G., Doble, J.: *Metapatterns: A pattern language for pattern writing*. In: *The 3rd Pattern Languages of Programming conference*, Monticello, Illinois (1996)
- [10] Björk, S., Holopainen, J.: *Patterns in Game Design*. River Media, Charles (2004)
- [11] Todd, E., Kemp, E., Phillips, C.: *What makes a good user interface pattern language?* In: *Proceedings of the fifth conference on Australasian user interface*, Dunedin, New Zealand, vol. 28 (2004)
- [12] Cunningham, W.: *Tips for writing Pattern Languages* (1994)
- [13] Kjellerup, N.: *The Galley Slave Model*. In: Kjellerup, N. (ed.) *Call Centre Know How Essays: Productivity, Measurements & Benchmarks*, vol. 2006, Resource International Pty Ltd., Ashgrove (2005)
- [14] Rupp, C., Goetz, R.: *Linguistic Methods of Requirements-Engineering (NLP)*. In: *Proceedings of the European Software Process Improvement Conference (EuroSPI)*, Denmark (2000)
- [15] Henninger, S., Ashokkumar, P.: *An Ontology-Based Infrastructure for Usability Design Patterns. Semantic Web Enabled Software Engineering (SWESE)*, 41–55 (2005)
- [16] Herzberg, F.: *The motivation of work*. John Wiley & Sons, Chichester (1959)

- [17] Schmidt, K.-H., Kleinbeck, U.: Funktionsgrundlagen der Leistungswirkungen von Zielen bei der Arbeit. In: Jerusalem, M., Pekrun, R. (eds.) *Emotion, Motivation und Leistung*, pp. 291–304. Hogrefe, Göttingen (1999)
- [18] Latham, G.P., Lee, T.W.: Goal setting. In: Locke, E.A. (ed.) *Generalizing from laboratory to field settings*, pp. 101–117. Lexington Books, Lexington (1986)
- [19] Schmidt, K.-H.: *Motivation, Handlungskontrolle und Leistung in einer Doppelaufgabensituation*. VDI-Verlag, Düsseldorf (1987)
- [20] Locke, E.A., Latham, G.P.: *A theory of goal setting and task performance*. Prentice-Hall, Englewood Cliffs (1990)
- [21] Mahemoff, M.J., Johnston, L.J.: Pattern Languages for Usability: An Investigation of Alternative Approaches. In: *Asia-Pacific Conference on Human Computer Interaction (APCHI) 1998*, Shonan Village, Japan (1998)
- [22] Granlund, Å., Lafrenière, D., Carr, D.A.: A Pattern-Supported Approach to the User Interface Design Process. In: *International Conference on Human-Computer Interaction, HCI International 2001*, New Orleans, USA (2001)
- [23] Niebuhr, S., Graf, C., Kerkow, D.: *Pattern für Fun-of-Use im Kontext einer Service-Center-Anwendung*, Fraunhofer-IESE, Kaiserslautern, Germany, IESE-Report 154.06/D (2006)
- [24] van Welie, M., Trættemberg, H.: Interaction Patterns in User Interfaces. In: *7th. Pattern Languages of Programs Conference*, Allerton Park Monticello, Illinois, USA (2000)
- [25] van Welie, M.: *Patterns in Interaction Design - Web Design Patterns*, vol. 2006 (2006)
- [26] Graham, I.: *A Pattern Language for Web Usability*. Addison-Wesley Longman Publishing Co., Inc., Amsterdam (2003)
- [27] van Welie, M.: *Patterns in Interaction Design - MobileUI Design patterns*, vol. 2006 (2006)

## Questions

### ***Peter Forbrig:***

*Question: Do you think a basic training in HCI and overview knowledge of the patterns in the collection is required to affectively and correctly apply patterns?*

Answer: Yes, I agree.

# Organizing User Interface Patterns for e-Government Applications

Florence Pontico<sup>1</sup>, Marco Winckler<sup>1</sup>, and Quentin Limbourg<sup>2</sup>

<sup>1</sup> LIIHS-IRIT, Université Paul Sabatier, 118 Route de Narbonne  
31069 Toulouse, France

{Florence.Pontico, Marco.Winckler}@irit.fr

<sup>2</sup> SmalS-MvM, Rue du Prince Royal 102

1050 Bruxelles, Belgium

Quentin.Limbourg@smals-mvm.be

**Abstract.** The design of usable interactive systems is a complex task that requires knowledge and expertise on human factors and on software development. Usability guidelines and design patterns may be one way to alleviate the lack of expertise on usability of development teams by providing guidance to solve every designer's problem when designing and developing User Interface. However, the utility of guidelines and design patterns relies on two main issues: a) the quality of the advices provided, and b) the way they are organized allowing fast access to the appropriate solutions. In this paper we discuss the organization of usability guidelines and patterns at the light of an industrial project at SmalS-MvM devoted to the development of e-Government applications in a very large scale. This paper presents not only a proposal of patterns organization but also it describes a set of analysis patterns identified for e-Government applications.

**Keywords:** Usability guidelines organization, design patterns, User Interface design process, e-Government applications.

## 1 Introduction

Nowadays, the problem of designing usable interactive applications has become a major concern because usability is recognized by standardization bodies like ISO [1] as a criterion of quality for software and, not less important, because poor designed application costs money to the company [2]. To study, express and ensure the usability of a User Interface, several disciplines can help every person who is responsible for developing the User Interface, notably participatory design, cognitive psychology, contextual enquiry, and software ergonomics [3]. Several methods issued from these disciplines have proven their positive impact on the usability of User Interface: usability evaluation methods with users [4], manual or automated inspection of the User Interface [5] and ergonomic approach based on guidelines [6].

In the last decades, guidelines have been used to capture and describe ergonomic knowledge. Guidelines are very versatile since they can be employed at several phases of development process. For example, they can be used to help designers to

make the right design decisions and to prevent the designer from making common mistakes but also they can support the evaluation of the final product. The utility of guidelines and design patterns relies on two main issues: a) the quality of the advices provided, and b) the way they are organized allowing fast access to the appropriate solutions. In fact, many guidelines are ambiguous and can be correctly applied only by an expert on User Interface design, which creates a barrier to a wider dissemination of guidelines due to the lack of this kind of expertise in the industry [7]. It is noteworthy that even experts might experience difficulties in selecting and applying guidelines, at least in the format in which they are conflicting with one another because there is a wide gap between the recommendation (e.g., “make the web site consistent”) and its applications [5].

In order to overcome this limitation of guidelines, some authors [8,9] propose to organize ergonomic knowledge under the form of design patterns. Design patterns emerged to cope with repetitive problems occurring in building architecture [10] and this concept has been extended by the Software Engineering community that created its own catalogues of proven solutions to recurrent software design problems [11]. Design patterns focus on the context of a very specific problem at a time and provide a solution that not only includes the ergonomic knowledge but also guides the designers to apply it in a practical way. Most guidelines can be extended to be expressed as patterns and the more recent research and development have preferred to present ergonomic knowledge in the form of User Interface patterns [7,12-15].

User Interface design patterns are easier to apply than guidelines but the number of patterns required to cover every usability problem increases the volume of the catalogue. This problem has already been observed when organizing guidelines [3] but it is even more dramatic in the case of patterns because patterns should be extended and reified for every application domain (e.g. web guidelines, mobile applications, etc) which increases again the volume of the information they provide.

In this paper we present a large case study conducted in the industry, at SmalS-MvM (<http://www.smals-mvm.be/>), where we followed the implantation of User Interface design patterns as a solution to create a usability culture in that company. SmalS-MvM is devoted to the design, deployment and handling of public e-Government applications. The discussions presented in this paper are therefore focused on e-Government domain, even though some of the lessons learned could be generalized to the organization of patterns in general. We performed an ethnographical study, which is fully described in section 2, to identify the needs in terms of access to information concerning ergonomic knowledge for the User Interface. At the light of data and evidences observed in the field, we propose, in section 3, an alternative way for organizing User Interface design patterns. During this study in the field it was possible to identify a set of patterns of User Interfaces for e-Government applications. Some of these User Interface patterns for e-Government applications are presented in section 4. In the section 5, we compare our proposal for organizing User Interface design pattern to the other organization schemas. In section 6 we discuss the lessons learned both on this state of the art of literature and on the case study we led in the field. Lastly, we present our conclusions and future work.



## 2 e-Government UI Analysis: A Study in the Field

SmalS-MvM is a non-profit organization devoted to the design, deployment and handling of public e-Government applications in Belgium. The current method of designing in SmalS-MvM enables the development of useful and usable e-Government applications. The design process is already user-centered, and follows many recommendations from HCI Software Engineering such as user testing and cooperative reflections led on mock-up supports onto final developed application. However, weaknesses appear about communication and reinvestment of design efforts from a project to another. That could be improved by a method of design that would fit these particular e-Government requirements.

### 2.1 Lots of Stakeholders, as Many Jargons and Viewpoints

One of the characteristics of e-Government is the huge number of stakeholders. This makes the design very complex because they all have to eventually cooperate in design and to be satisfied with the application while carrying different interests (interests in the application design and use, as well as political interests in general), and also different jargons, and backgrounds. Actually consider the number of persons involved in the design process:

- **Final users** can be administrative agents (social workers, office clerks, and so on) and/or citizens (individuals, representatives of an association, firm managers and firm manager secretaries). Most of the administrative procedures involving firms are actually conducted by agencies devoted to undertake procedures for the benefice of the firm. One should consider the critical aspect of the e-Government application for final users: the procedure must success because it emerges from a personal need (e.g. *I go to New-York for 2 weeks, I need a tourist Visa*) or from an administrative service need (e.g. *Visa applications have to be submitted to the embassy*), but also because personal and eventually confidential data is handled and stored during the procedure.
- **Clients** are the representatives of the institutions involved: administrative managers, commercials, domain experts and so on. The achievement of the procedure is critical for them as well, because it is intended to satisfy a need (e.g. *Management of housing benefit demands*) but also because a failure can have disastrous consequences on them in terms of corporate image. Proceedings can even be taken against the concerned institutions in some cases.
- **Design team** involves many corporate bodies for a single e-Government application: project manager, usability experts, analysts, content managers, data quality managers, graphic designers, developers, database experts, security experts and so on. They are responsible for the leading of the design process and some of them work directly with the clients (mainly the project manager, analysts and usability experts). The design firm is commercially engaged in the process which makes critical for them also that the final application permits fulfilling administrative procedures successfully and in a usable way.

## 2.2 Difficulties Encountered by the Design Team

SmaS-MvM employs more than 1.000 persons, mainly administrative staff, database managers, developers, architects, analysts, project managers, system and network experts. Some 25 projects are carried on, involving one or several institutions. One of the projects where many applications are developed and handled is the Social Security project. The Social Security portal (<https://www.socialsecurity.be/>) provides some static information and enables the fulfillment of administrative procedures in relation with the Belgian Social Security, most of them being targeted to firms. For example, the Social Risk Declaration is dematerialized on this portal: it enables an employer to declare an employee's inability to accomplish his work (e.g. in case of pregnancy, accident or disease). This way, the employee will receive allowance from the Social Security during the period he is off job. The ethnological study we led was in the context of this Social Security project.

A field observation revealed that the design process in SmaS follows many of the HCI Software Engineering recommendations. User testing is led from the very beginning of the application lifecycle, on mock-up support. User testing is done on implemented application as well. The mock-up is incrementally modified and improved until all design stakeholders agree on it. Then, the actual application (database implementation etc.) is realized and deployed. The firm is still in charge of the application after its deployment as it undertakes the call center management. Traces are kept by the call centre to allow follow-up: if a user is calling for the third time, the operator can be displayed the contents of the user's first two calls.

It appears that this iterative mock-up based process is hard to lead with so many stakeholders (see §2.1). Some weaknesses in communicating to the whole team are already noticed at the very beginning, when analysts have to transform business requirements in a first proposal to the rest of the working team. They seem to lack some expression support in order to define the application without entering into implementation details. This was quoted in a meeting, from an analyst about his work: *"I often let myself be tempted by coding some HTML pages, even if I realize that this way I already suggest design decisions that aren't yet required"*. A lack of expression support is there revealed by this analyst: no tool or notation is provided, and to communicate his analysis, he uses developer's language. To cope with this lack of power of expression about recurrent topics, a UI analysis patterns catalogue is being developed towards analysts.

## 2.3 User-Centered Approach of Making Patterns

A catalogue of UI analysis patterns has to be user-centered itself, just as any application deployed that cares about being actually used. Integrating such a tool for analysts will obviously modify their way of working; however, we have to get inspired by their current design activities to make the integration as smooth and useful as possible. That is the reason why the catalogue of UI analysis patterns is made in cooperation with volunteers belonging to SmaS-MvM (mostly analysts, developers, usability experts and content managers) and who are therefore daily involved in e-Government design projects. They are not UI pattern experts, but they are interested in this initiative and, as final users of such a methodology (if not directly users of the

patterns), they bring relevant comments and evaluation of the patterns in terms of their contents as well as the way to use them.

To constitute this catalogue of UI patterns, we browsed applications designed by SmalS-MvM among the ones already deployed or at advanced acceptance stages. This permitted us to pick up which UI fragments were keeping appearing in these e-Government applications. Good as well as bad examples of UI fragments were picked up in order to get as many arguments as possible for proven solutions, including by giving wrong examples (anti-patterns). Once the list constituted, those recurrent fragments of UI were integrated in UI patterns. As for the content of these patterns, we studied the design process in order to ensure a successful integration in it. Analysts are responsible for the first rough UI proposal after they have studied and treated business requirements. At CHI 2002 Workshop [16], it was suggested that wireframes could be integrated in UI patterns. This fits very well our present case: low-level fidelity UI prototypes are integrated in our UI analysis patterns, so that business requirements can be mapped to those first rough drafts.

### 3 Organizing UI Patterns for e-Government Applications Analysis

UI patterns can be integrated at several stages of an eGovernment design project: to support analysis and specification, to organize the information, to study graphical aspects and even to evaluate the usability of the application [14]. In this work, we focus on analysis that is the transformation of business requirements into a first specification. The specification of interaction at early stages of design is already possible thanks to several notations and formalisms, with various main intentions: supporting communication in the design team for MoLIC [17], formalizing and simulating the navigation model for StateWebCharts [18], organizing and presenting information for WebML [19]. Our own intentions are mainly the following: describe the User Interface (navigation and layout) without ambiguity though avoiding technical details, and intuitively enough so that any design stakeholder can read and at least slightly modify the description. To support these intentions, UI analysis patterns can follow the template presented in the Fig. 1.

<b>TITLE OF THE PATTERN</b>	
<b>DESCRIPTION</b>	Description of the pattern
<b>EXAMPLES</b>	Screen captures of good and bad examples of use of this pattern
<b>CASES OF USE</b>	Cases when this pattern must be applied, when it should, when it shouldn't and when it mustn't be applied (anti-patterns)
<b>LAYOUT</b>	Advices about visual implementation of the pattern
<b>RATIONALE</b>	Reason for the solution, may it be scientific or empirical. When it is a theoretically proven solution, resources (scientific papers, online catalogues or useful design books) are referenced to encourage the analyst to know more about the topic and to help him add or modify patterns if necessary
<b>WIREFRAME</b>	Draft of the user interface. It can have different shapes along the nature of UI Pattern concerned. Some patterns will actually deal with <i>Screen Flow</i> level topics, other ones with <i>Page</i> level, and some other with <i>Basic Components</i>

**Fig. 1.** Template of a User Interface analysis pattern

The description of our UI patterns is rather classical (advices of implementation and rationale around a given UI design problem) until we reach the WIREFRAME attribute. Patterns have to provide solutions to recurrent problems, but it is not enough in this context: the considered solution has to be readable and understandable by every stakeholder. It even has to be a first step, an input from the analysts in the mock-up based iterative process. We therefore integrate a rough draft of the UI in our UI patterns. There are eventually different drafts illustrating several solutions to a same problem, if concrete parameters influence the application of the UI pattern. Different alternatives can be indexed in the pattern, referring to sub-patterns describing with precision each situation in which the considered sub-pattern is to be used (see MULTI-STEP WIZARD pattern Fig. 3). This way, the analyst is able to compare different propositions to choose which one better fits his proper situation. According to the level of granularity of the UI patterns, the WIREFRAME consists in a schematic representation of the layout and disposition of an UI element (ex. a page or a form), or in a rough schema of the navigation. For this former case, we chose StateWebCharts (SWC) [18] navigation modelling formalism which is an extension of StateCharts [20] devoted to navigation modeling on web applications. SWC presents the advantage of being both not ambiguous and easy to read and modify.

UI analysis e-Government patterns can be naturally classified along a hierarchical structure, following a quite traditional way of designing applications: from the general to the details. This structure implies a kind of “progressive disclosure of information” for analysts. However, when presenting them such a structure, some analysts told us about their will to have some other access to UI analysis patterns information: *“On top of that guiding procedure of browsing the UI patterns [from top to bottom], I would like to be able to directly find recommendations on list boxes for example. Couldn’t we have some search engine inside the catalogue?”* This request is of great interest because it outlines the need to provide direct access to patterns, in addition to top-bottom or bottom-top paths. Moreover, patterns should refer to other ones, in order to allow transversal navigation in the pyramid. Works on this topic can be found in the literature, from just considering that some patterns can ‘refer to’ other ones, to more complex networks using Semantic Web concepts for linking them. We will investigate afterwards in this paper (§5) existing methods to organize UI patterns to give directions on how our pyramidal organization can be completed with some relevant direct and transversal accesses.

## 4 Identified UI Analysis Patterns

E-Government is a highly repetitive domain, which makes design patterns relevant to reinvest design knowledge from a project to another. In particular, UI analysis patterns should be associated to the UI analysis recurrent problems listed while browsing existing applications. A support would then be provided to analysts when transforming business requirements (coming from the client) into a first draft of User Interface that will be discussable with the rest of the design team (including the client himself).

#### 4.1 Listing of Recurrent Fragments of e-Government User Interface

The best way to list relevant UI patterns is to browse existing applications, listing manually what keeps occurring. For this activity, we browsed and studied a set of some 25 applications designed by SmalS, already deployed or in final phases of testing. Some recurrent pieces of interface stood out at three different levels of UI granularity. As a first proposal (see §3), a pyramidal structure is taken to organize patterns (see Fig. 2 below.) At the top of the pyramid, patterns stand that help structuring the application in terms of *Screen Flow*, giving directions on how to structure the procedure achievement. Underneath, interface patterns directions are given at the *Page Level*: layout of a wizard step, form fields grouping and displaying, position of the state of advancement of the procedure and so on. Lower again are *Basic Components* recommendations such as advices on how to signalize that a form field is mandatory. The basis of this pyramid is actually a set of ergonomic recommendations and even more: the “Golden Rules” recommended in HCI design whatever the application support may be [21,22].

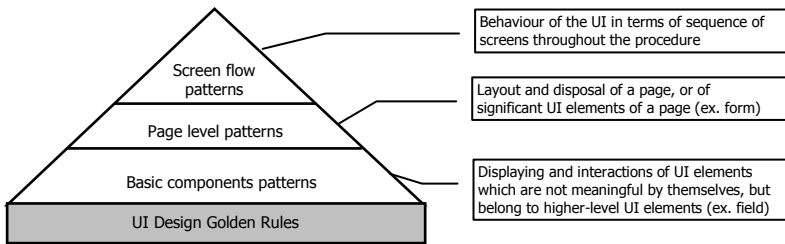


Fig. 2. Pyramidal structure of e-Government UI analysis patterns

**Screen Flow Level.** Few sequences of pages actually occur in e-Government applications. Several of them can appear and be combined in a single application.

- *Consult and Modify Data* screen flow consists in consulting and modifying one or more items from a displayed list (e.g. management of employees’ information for an employer);
- *File Management* support several activities in parallel (e.g. application for social workers to report endangered people and follow their ongoing files);
- *Hub and Spoke* flow, from a dashboard page, allows the access to a procedure just as if the user entered a funnel. At the end of the funnel, the user is led back to the first page (e.g. application for a firm employer to declare information on each employee towards the Social Security);
- *Integration in a Portal* flow is about referring and allowing access to an application from a portal (e.g. any application related to Social Security portal);
- *Multi-Step Wizard* flow consists in a strongly guided sequence of pages to achieve a single procedure (e.g. individual citizen’s declaration of incomes);
- *Role Management* flow occurs as soon as the application interface depends on the role of the user (e.g. website providing offers and demands of jobs provide different functionalities to bidders and to demanders).

**Page Level.** Several fragments keep occurring as well at the page level. Here are some of them, possibly combined just as Screen Flow level UI fragments.

- *Acknowledgement of Receipt* page is to be displayed and proposed for printing each time a procedure has been successfully accomplished (e.g. after a Social Risk Declaration, the employees appear with the associated declarations, as well as an identifier of the web session, so that if there are some modifications to do, the declaration is easy to find);
- *Advancement Box* appears on each page of a multi-step wizard procedure to show the user his current position, what steps have been done, and which ones are to be done (e.g. during the declaration of incomes, such a box will display the sub categories of incomes to declare, and where the user is currently arrived);
- *Clear Entry Points* page supports the displaying of a few choices, each of them leading to a different part of the application, or to make the user fill in the first step of a procedure (e.g. “I want to: declare my incomes / modify my declaration / follow-up the treatment of my declaration”);
- *Filter a List* page shows how the filtering can be done and other eventual functionalities directly available on the items (e.g. for a social worker, filter should be provided on the list of cases, according to the name of the person concerned, the name of the agent who initiated the case, the date of creation, or the state of advancement of treatment of the case);
- *Overview* page is displayed at the end of the procedure and, if validated both by the user and the system, it leads to the Acknowledgement of Receipt (e.g. a summary of the Social Risks declared during the web session is displayed to the employer, so that he can check the information filled in before validating the procedure);
- *Wizard Step* has to provide the form corresponding to the current step, and some information on the state of advancement of the procedure (e.g. inheritance incomes declaration is one of the wizard steps of the incomes declaration).

**Basic Components.** Many fragments of the interface in terms of basic components of a page can be found in existing e-Government applications. At this level, the fragments could be applied to some close domains, such as e-Commerce for example.

- *Conditional Activation of Fields* is appreciated to deactivate the filling of a non-relevant field (e.g. “Name of the spouse?” should be deactivated in the case of a single person);
- *Download Link* have to provide information about the type of file to be downloaded, its weight and so on (e.g. proxy form, PDF format, 37 ko);
- *Mandatory Fields* have to be signaled by an asterisk just after the label (e.g. last name or social security number);
- *Non Textual Objects* such as images or video have to provide alternative text for those who can’t display them, for example blind people (e.g. “logo Social Security” as an alternative text for the picture);
- *Pre Formatted Form Fields* occur when the user has to fill a formatted field, above all when the data is intended to be automatically treated afterwards (e.g. date of birth or bank account identifier);
- *Typography* has to be taken care of, and standardized among the applications of a same portal (e.g. font size must be 11pt).

## 4.2 Examples of User Interface Analysis Patterns

Here are three of the UI analysis patterns that are to appear in SmalS-MvM catalogue, each one belonging to the different levels of granularity listed in section 4.1. For lack of space and for the sake of readability, bad and good examples screen captures are not displayed here. For the same reason, UI patterns are flattened: they are usually displayed as a set of tabs, with a tab for each attribute (DESCRIPTION, EXAMPLES, etc.). The first UI analysis pattern extracted from our catalogue is named “MULTI-STEP WIZARD”. This is a Screen Flow level UI pattern as it describes the way a multi-step procedure should be structured among several screens when some guidance is required. This UI pattern corresponds to a very recurrent UI topic as it appears in 80% of the applications we reviewed. Three alternatives of screen flows are proposed in the WIREFRAME attribute, corresponding to different ways to let the user correct the data he filled in when he reaches the overview page. Each one of these three alternatives corresponds to a sub pattern of the MULTI-STEP WIZARD pattern (Fig. 3), as each one has to be applied in different contexts and situations.

### MULTI-STEP WIZARD

**DESCRIPTION** The goal of the procedure is reached through the accomplishment of a sequence of activities. This sequence of activities is guided by the sequence of screens but also by the navigation proposed which is limited to “next step” and previous step” eventually “cancel all”).

**EXAMPLES** Good Declaration of a foreign employee to the Social Security

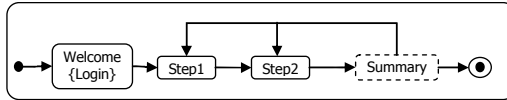
**CASES OF USE** Must be used when the user is a novice  
Shouldn't be used when the user is very likely to interrupt his task before the achievement of the procedure

**LAYOUT** 1) Distinguish procedure steps (ex. Step 2) and auxiliary pages (ex. OVERVIEW page)  
2) See WIZARD STEP pattern for the layout of each step  
3) Give the procedure a clear title, whose formulation is user-centred and contains a verb corresponding to the goal of the procedure.

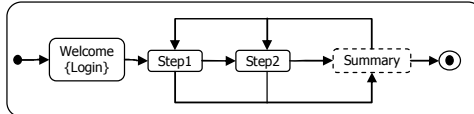
**RATIONALE** 1) [http://www.designofsites.com/about\\_the\\_book/patternh1.pdf](http://www.designofsites.com/about_the_book/patternh1.pdf)  
2) [http://harbinger.sims.berkeley.edu/ui\\_designpatterns/webpatterns2/webpatterns/pattern.php?id=7](http://harbinger.sims.berkeley.edu/ui_designpatterns/webpatterns2/webpatterns/pattern.php?id=7)

**WIREFRAME** Several implementations are possible, just around the way provided for the edition of the overview page. See MULTI-STEP WIZARD sub patterns to identify which one fits to your situation.

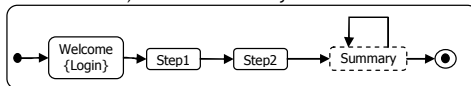
SUB PATTERN 1) Strong guidance wizard



SUB PATTERN 2) Supple guidance wizard



SUB PATTERN 3) Editable summary

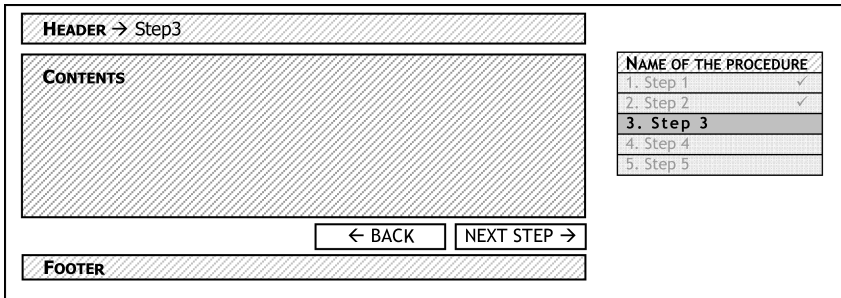


**Fig. 3.** Example of UI analysis pattern at the Screen Flow level: “MULTI-STEP WIZARD”

Hereafter, Fig. 4 presents the **ADVANCEMENT BOX** UI pattern. It belongs to the Page Level as it concerns the layout, disposal and behavior of a UI fragment which has a sense by itself in the application. This pattern appears (or should appear) in as many applications as the **MULTI-STEP WIZARD** pattern we saw above, which means very often in e-Government applications.

### ADVANCEMENT BOX

- DESCRIPTION** Display the user its current position in the procedure: where he is, what he has done successfully or not), what is left to be done.
- EXAMPLES** Good Declaration of a foreign employee (box on the right of the screen)  
Bad Declaration of socially endangered persons (no advancement box)
- CASES OF USE** Must be used in multi-step wizard procedures holding three or more steps
- LAYOUT**
  - 1) Use 2 shades of 1 colour for the background of the box places, the deeper one signaling the current step, the lighter for the steps done or to be done.
  - 2) Use three icons to show the state of a step (e.g. ✓ x -)
  - 3) Don't use checkboxes to indicate (completed) steps as this can give a false impression users can click on them.
  - 4) Give each step a number
  - 5) Put the box in the right-hand part of the screen, just as any non critical information that can be missed by people holding a low screen resolution.
- RATIONALE** Van Welie "Purchase Process" pattern is close to this one, with a line instead of a box  
<http://www.welie.com/patterns/showPattern.php?patternID=purchase-process>
- WIREFRAME** The advancement box appears on the right-hand side



**Fig. 4.** Example of UI analysis pattern at the Page level: “ADVANCEMENT BOX”

Our last example is presented above in Fig. 5 and is called “**MANDATORY FIELD**”. This Basic Component UI pattern could appear in any web application holding forms and caring for usability. This pattern is useful because, if most of the applications investigated do signalize the mandatory fields, many of them don’t place correctly the asterisk just after the label which is yet better for the readability of the form. In other terms, this is the kind of UI patterns that carries usability principles which are basics ones but often missing. It outlines as well that our UI analysis patterns catalogue strongly suggests a uniform solution to analysts. Other ways to distinguish mandatory fields could actually have been suggested (ex: *use red to label mandatory fields* or just some *distinguish mandatory fields from optional ones* advice) but our purpose here is to provide directly applicable and unified solutions to analysts, towards uniformed e-Government applications, at least for the applications belonging to the same portal, such as in our case with the Social Security portal.



<b>MANDATORY FIELD</b>	
DESCRIPTION	Warn the user about the fields required to pursue the procedure
EXAMPLES	<u>Good</u> Forms of the social workers' application supporting cases management <u>Bad</u> Forms of the incomes declaration (bad disposition of the asterisk)
CASES OF USE	<u>Must</u> be used as soon as there are mandatory AND optional fields in a form.
LAYOUT	1) Use an asterisk, just after the label of the concerned field 2) Write an obvious legend 3) Insert an asterisk (character) or an image
RATIONALE	<a href="http://www.welie.com/patterns/showPattern.php?patternID=forms">http://www.welie.com/patterns/showPattern.php?patternID=forms</a> .
WIREFRAME	The third field is mandatory in this example

The diagram shows a rectangular form with a hatched background. At the top left, it says "FORM TITLE". Below this, there are three rows of text and input fields. The first row is "Label of optional field" followed by an empty rectangular input box. The second row is "Label of optional field" followed by another empty rectangular input box. The third row is "Label of mandatory field\*" followed by a third empty rectangular input box. Below these three rows, there is a legend: "\* Fields marked with an asterisk are mandatory". At the bottom right of the form, there is a button labeled "OK".

Fig. 5. Example of UI analysis pattern at the Basic Component level: “MANDATORY FIELD”

## 5 Related Work

One of the major issues for the use of User Interface patterns in the practice is the proper organization of patterns in accessible catalogues providing fast access to the appropriate solutions. Fincher [8] claims that patterns must be organized in such a way that they are easy to locate, they are grouped when appearing in common cases, they provide different viewpoints, and they permit to generate new solutions from the ones proposed. The most famous collections of UI patterns provide some intrinsic classification that is a proposal of some categories supposed to be useful for an efficient browsing of the collection. These catalogues might concern User Interfaces in general [12,24] or be focused on a particular application domain such as web applications [25], e-Commerce [26,27], and mobile applications [28]. Specialized catalogues are created by selecting already known patterns and, based on experience on considered field, adapting known patterns and identifying new ones. As far as we know, there is not yet a catalogue for e-Government applications. This might be explained by the emergence of e-Government and as such, some time is needed for the community to identify successful solutions that could be clearly stated as *patterns*.

### 5.1 Currently Available UI Patterns Catalogues and Inner Organization

Hereafter we present a short summary of most representative UI patterns catalogues found in the literature. We focus in particular on the way the patterns are organized in the catalogue rather than their content.

The **Van Welie's catalogue** [29] is a large catalogue which is organized in subsets according to the application domain: ex. Web-based applications, mobile applications and GUI design in general (which is at a higher level of implementation detail for the design phase we consider here that is early UI specification). In this catalogue, patterns are basically centred on the user's intentions. Examples of categories and patterns in categories: *SITE TYPES* (ex. *artist site, portal, etc.*), *USER EXPERIENCES* (ex. *fun, shopping, etc.*), *E-COMMERCE* (ex. *shopping cart, store locator, etc.*), etc.

The **Yahoo! Design Patterns Library** [13] follows a goal-oriented approach. Reflections on how authors came to this classification are available online [30]. The outlined goals actually include user’s goals and designer’s goals, considering that the User Interface has to satisfy both of them. This way, user’s intentions and needs can be satisfied – for example: *USER NEEDS TO: NAVIGATE* (ex. of patterns: *breadcrumbs, tabs, etc.*), *EXPLORE DATA* (ex. *calendar picker, pagination, etc.*)... – as well as designer’s technical constraints – for example: *APPLICATION NEEDS TO: CALL ATTENTION* (ex. *help by dynamic tool tip, transition with an animation, etc.*), *GROUP RELATED ITEMS* (ex. *scrolling list, tree, etc.*), etc.

The **Coram’s catalogue** introduced Experiences [31] as a new UI pattern language in order to cope with high-level UI design problems. These patterns are grouped by focus and belong to a network which is presented in Fig. 6. From “Interaction style” meta-pattern, patterns are grouped and linked in four categories, corresponding to how the user is intended to interact with the application.

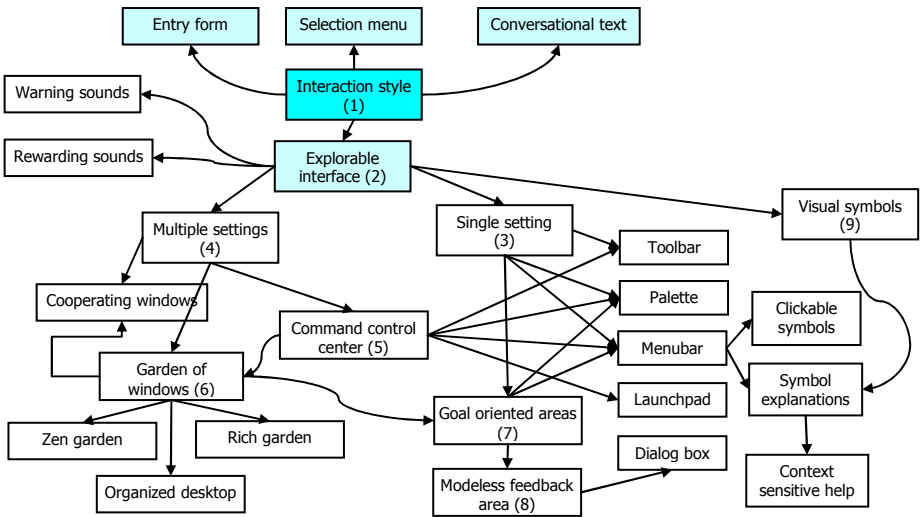


Fig. 6. Partial view of the map of the Experiences UI patterns (from [31])

The **Laakso’s catalogue** [24] covers several kinds of applications, including web but not only. Most is done about visualisation that is about how information and/or data are organized (ex. *DATA VIEWS* category contains these patterns: *overview beside detail, fisheye, etc.*) even though some categories are devoted to displaying of information (ex. *TIME: calendar strip, schedule*); command interactions are included as well (ex. *SAVE AND UNDO: auto-save, object-specific undo, etc.*).

The **Tidwell’s catalogue** [12] is a collection of generic UI design patterns that can be used to deal with web applications, mobile applications or any other kind of interfaces. The patterns are very generic and cover multiple levels of the User Interface design. Some of the categories are entirely devoted the description of interactions with users (ex. category *GETTING INPUT FROM USERS*, contains the following patterns: *forgiving format, dropdown chooser, etc.*). Other examples of patterns

categories: *ORGANIZING THE CONTENT* (ex. of patterns: two-panel selector, wizard, etc.), *SHOWING COMPLEX DATA* (ex. overview plus detail, cascading lists, etc.), etc.

The **Van Duyne's catalogue** [23] is designer-oriented (e.g. "helping customers complete tasks") but the catalogue aims to follow a "customer-oriented approach". This calling emphasizes the help that is given about functional and procedural aspects of the web application, such as "buying products" or "search for a similar product". At the beginning, there is some progressive in-depth display of the patterns (site genre, then navigation framework, then homepage), but it is lost afterwards, in favour of more general advices. Example of categories in this catalogue: *SITE GENRES* (ex. of patterns: personal e-Commerce, self-service government, etc.), *CREATING A NAVIGATION FRAMEWORK* (ex. alphabetical organization, popularity-based organization, etc.), *CREATING A POWERFUL HOMEPAGE* (ex.: homepage portal, up-front proposition), etc.

The **Montero's catalogue** [25] aims to guide design towards usable web applications. Its specificity is that patterns in this catalogue are grouped along three levels of abstraction: *WEB SITE*, *WEB PAGE* and *ORNAMENTATION*, based on Alexander's first works about architecture patterns [10]. Moreover, a network weaves patterns throughout categories, around common ergonomic advises for web design, as it is shown in the Fig. 7 below.

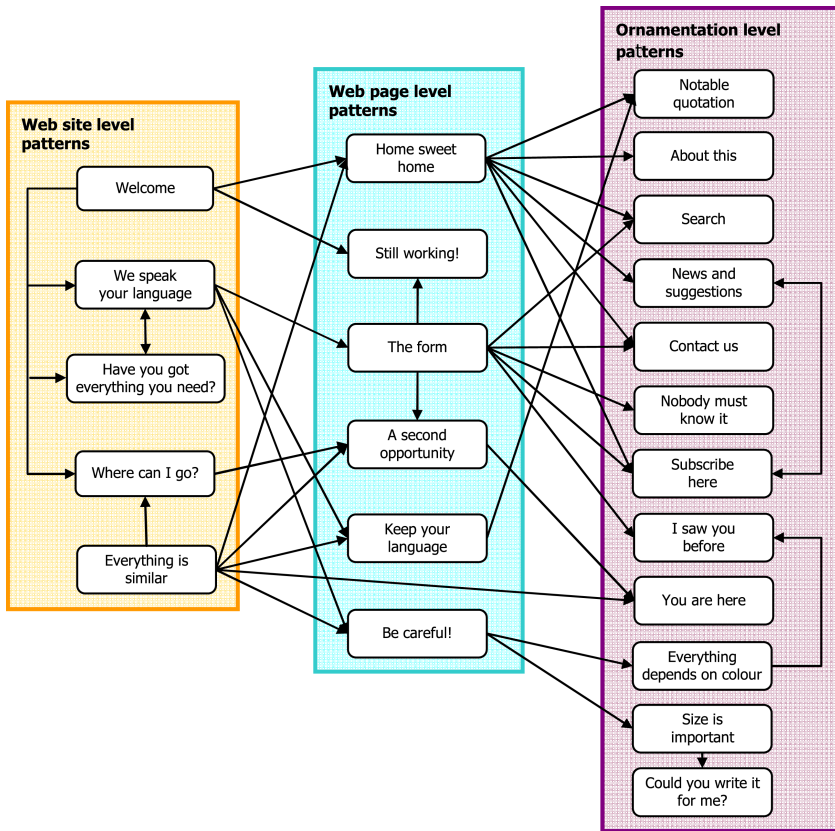


Fig. 7. Montero's proposed pattern language (from [25])

In addition to these catalogues, some authors provide alternative methods for structuring their catalogues. The rationale behind this alternative organization is that patterns could therefore be “composed of” or “derived from” another one or other ones. Two classification proposals following such an approach are noteworthy: the **Object-Oriented organization** proposed by Van Welie et al. [7], and the proposal of Henninger et al. [15] using a **Semantic Web** approach.

Van Welie et al. [7] investigated the possibility of structuring web UI patterns in a hierarchical way featuring an Object-Oriented organization. This way of structuring with a top-down approach is actually similar to what had been proposed at the very beginning of the design patterns history [10]. Web design patterns can therefore belong to different levels that are:

- **POSTURE**: reason for existence of the application (ex. *e-Commerce, Personal site*), coming from the business goals (ex. *Customer satisfaction, Selling products*);
- **EXPERIENCE**: high level goal for which the user comes to the website, beyond functional tasks and goals (ex. *Playing, Shopping, Browsing, Sharing thoughts*);
- **TASK**: solutions to small user problems which are part of a higher level “Experience”; the solution is given in terms of a set of interactions (ex. *Product comparison, Identify*);
- **ACTION**: pieces of interaction that are at the lowest level of interest for UI patterns; they are meaningful only if they are related to a task pattern (ex. *Pushbutton*).

Moreover, some precisions are given that imitate Object Oriented modeling, in order to distinguish different relationships of connecting patterns: aggregation, specialization and association.

The approach suggested by Henninger et al. [15] aims to make more pro-active the representation of sets of design patterns in general (i.e. not especially UI patterns, but we only observe the structure principle here, which could be applied to UI patterns). The authors presuppose that weaving design patterns thanks to Semantic Web methods would provide a more usable and navigable set of patterns for designers. As Semantic Web is developed to cope with a more efficient and supple access to information whose volume is increasing, this method may effectively be of interest for design patterns information. A tool is associated to this framework, supporting the edition of ontology to weave the design patterns: BORE (Building an Organizational Repository of Experiences) [32]. The main goal of this section is rather discuss strategies for guidelines rather than the content of patterns themselves.

## 5.2 Classifying UI Catalogues Organizations

Based on our experience in the field at SmalS-MvM, we have identified some suitable requirements for organizing UI patterns. Hereafter we present a list of these requirements, which were inspired from Fincher’s criteria, to evaluate the organization of currently available UI patterns catalogues:

- **Hierarchical/Pyramidal** access has to be provided as a “progressive disclosure of information” which is a natural way of thinking design.
- **Cross References on UI Elements** appearing in different patterns. For example, if a pattern contains a list box, references should be available to other patterns in which list boxes appear as well.

- **Siblings grouping.** Patterns which often are of interest in common cases should be put together and therefore create similar families' patterns that may be applied to similar applications.
- **Viewpoints comparison.** In some cases, several patterns can be applied. This criterion is about the way the designer is supported in this choice.
- **Evolution and scaling.** Can the list of existing patterns be augmented? Is scaling possible? In other terms, this criterion tells if the investigated organization of patterns would bear an important volume of data.

The Table 1 provides a comparison of patterns catalogues found in the literature according to the criteria aforementioned.

**Table 1.** Evaluation of reviewed organizing UI patterns principles

	Pyramidal structure	Cross-ref on UI elements	Siblings grouping	Viewpoints comparison	Evolution and scaling
Coram	✓	✗	✓	✓	✗
Henninger	✗	✓	✓	✓	✓
Laakso	✗	✗	✗	✗	✗
Montero	✓	✗	✗	✗	✗
Van Duyne	✓	✗	✓	✓	~
Van Welie catalogue	✓	✗	✓	✓	✗
Van Welie oo organization	✓	✗	✓	✓	~
Yahoo!	~	✗	✓	✓	✗

Legend: ✓ supported, ✗ not supported, ~ cumbersome

## 6 Lessons Learned

Integrating some new artifact as a support to an existing activity is a sensitive process. The way of leading the activity will anyway have to be adapted to this new artifact, whatever its quality will be. For a supple adaptation, the authors of the artifact have to consider how users currently carry activities, and, as much as possible, to confront the project of artifact to their opinion. If not, the artifact is very likely to be rejected (in the case of a commercial product for example) or diverted by its users towards a way that better fits their habits and needs (in the case of a support to work for example). Observation and user testing are therefore wise ways to design useful and usable products. To follow this HCI basic principle, we had to learn more about analysts' activities both from current web design methodologies [33-35] and from analysts' observations and reactions to the UI patterns proposed. In parallel, to feed our reflection on UI patterns, we reviewed the literature and studied other works' experiences and conclusions. This section is a summary of the lessons learned both from the ethnological study and the UI patterns literature browsing.

**Need for e-Government patterns.** The browsing of existing e-Government applications revealed that patterns are relevant for e-Government which is a highly repetitive context, with many recurrent fragments appearing (see §4.1 for the particular case of UI patterns). Moreover, the strong rationale included (by definition) in UI patterns would help coping with some decisions that may be hard to take when stakeholders hold divergent interests. Patterns help bringing people's opinions back together for the benefice of the application, which is very useful in e-Government where so many stakeholders are involved (see §2.1).

**Need for e-Government specific UI patterns.** UI patterns have to be close to their application domain. In particular at the highest level, specific UI fragments occur as we consider a defined domain. E-Commerce UI patterns are proposed in several studies such as Van Welie's catalogue [29], which can somehow but not entirely help building e-Government applications, in spite of their common points [27]. For example, an e-Commerce Page Level UI pattern includes incentives to buy additional products whereas in e-Government, the purpose is to provide clear and formal information to fulfill the goal, not to give rise to new wishes.

**User-centred UI patterns and catalogue.** Integrating a design support must be done with respect to current activity. The UI patterns catalogue must therefore be user-centred. Observations and meetings with design team members, as well as investigations on theoretical design practices are done all along the making of this catalogue (see §2.3). However, some rigorous user testing has to be carried out as soon as the catalogue is complete enough to be operational.

**E-Government UI patterns content.** The usability of the UI patterns proposed first depends on their content (see §3). Bad and good examples have to appear to support and illustrate the rationale included in the patterns. Both static and dynamic aspects of the application have to be described in a non ambiguous though "easy to read and modify" way. The static behaviour mainly refers to the layout of the pages and UI elements (such as forms). UI patterns on static topics are accompanied by wireframes to be an efficient support for communication among stakeholders. For the same reason of readability and non ambiguousness, StateWebCharts formalism ensures the representation of the dynamic aspects (navigation among the application).

**E-Government UI patterns organization.** UI patterns have to be displayed in a way that suggests their actual use. The investigations we made in the field revealed that analysts not only need a progressive disclosure of information, but also some transversal access to the UI patterns information (§3). UI patterns organization has therefore to be efficient concerning easy location of patterns, cross references on UI elements or on context of application, comparison and grouping of patterns applicable in close situations, and finally a possible increasing of the number of UI patterns while keeping the benefits of the organization. Existing methods of UI patterns organization don't fit these requirements, globally failing in providing relevant cross references among patterns, and in supporting an evolution that would lead to a huge number of patterns (§5.2). However, Semantic Web principles appear to be the more relevant among the organization principles investigated.

## 7 Conclusion and Future Work

As e-Government influence keeps increasing, more and more IT firms are eager to invest their efforts into this complex domain. The important number of stakeholders involved in such projects makes e-Government design a hard activity to lead. They are critical systems for the institutions involved as well as for final users. To ensure that the goals of these final users will be satisfied thanks to a usable application, UI patterns are a solution. We studied contents for e-Government UI patterns as well as an organization for a user-centered displaying of UI patterns to analysts. This study was based on an ethnological study as well as on literature. These investigations prompted us to find a relevant organizing UI patterns as a critical topic for UI patterns usability and acceptance in the design team. The UI analysis patterns catalogue contents and organization are strongly related to the activity observed in the field and also to the particular tasks fulfilled in the investigated domain. Users and their supposed tasks are well-known in this mature e-Government domain, and that was the basis of the catalogue building. This is actually a limit of our work which, to be extended to other domains, would necessitate the same kind of investigation in the field and inventory of recurrent patterns. However, this methodology employed to build UI analysis patterns could be reinvested for other domains, in particular the lessons learned about the development of a user-centered organization of patterns and their integration in a design process. Our future work envisages the building of an ontological mapping of the concepts appearing in UI patterns. Inspired by Semantic Web principles, this could support as many navigation links among UI patterns as there are links among UI patterns concepts. Moreover, by nature, this kind of structure would support the enlargement of the existing UI patterns catalogue. The necessary support to consult and edit patterns has to be considered as well. This possibility may be given as well for the user of the catalogue to make his customized organization. UI patterns are a relevant support for e-Government design because it copes with recurrent design questions with a strong rationale and first proposals towards a usable application.

**Acknowledgments.** This was possible thanks to SmalS-MvM and to COST294-MAUSE (<http://www.cost294.org/>).

## References

1. ISO/WD 9241, Ergonomic requirements for office work with visual displays units International Standard Organization (1992)
2. Mayhew, D.J., Bias, R.G.: Cost-justifying usability. Morgan Kaufmann, San Francisco (1994)
3. Vanderdonckt, J.: Development milestones towards a tool for working with guidelines. *Interacting with Computers* 12(2), 81–118 (1999)
4. Hix, D., Hartson, R.: *Developing user interfaces: ensuring usability through product and process*. John Wiley & Sons, New York (1993)
5. Ivory, M.Y.: *Automated web site evaluation: researchers' and practitioners perspectives*. Kluwer Academic Publishers, Dordrecht (2003)

6. Bastien, C., Scapin, D.L.: Evaluating a user interface with ergonomic criteria. *International Journal of Human-Computer Interaction* 7, 105–121 (1995)
7. Van Welie, M., Van der Veer, G.C.: Pattern languages in interaction design: structure and organization. In: *Interact 2003*, Zürich, Switzerland (2003)
8. Fincher, S., Windsor, P.: Why patterns are not enough: some suggestions concerning an organizing principle for patterns of UI design. In: *Workshop Pattern languages for interaction design: building momentum (at CHI 2000)*, The Hague, The Netherlands (2000)
9. Javahery, H., Seffah, A.: A model for usability pattern-oriented design. In: *TAMODIA*, Bucharest, Romania, pp. 104–110 (2002)
10. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: *A pattern language*. Oxford University Press, New York (1977)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns, elements of reusable object-oriented software*. Addison Wesley, Reading (1995)
12. Tidwell, J.: *Designing interfaces*. O'Reilly, Sebastopol (2005)
13. Yahoo! Design pattern library (2006),  
<http://developer.yahoo.com/ypatterns/>
14. García, F.J., Lozano, M.D., Montero, F., Gallud, J.A., González, P., Lorenzo, C.: A Controlled Experiment for Measuring the Usability of Webapps Using Patterns. In: *ICEIS*, Miami, USA (2005)
15. Henninger, S., Ashokkumar, P.: An ontology-based infrastructure for usability patterns. In: *Workshop on Semantic web enabled software engineering (at ISWC)*, Galway, Ireland (2005)
16. Van Welie, M.: Patterns for designers? In: *Patterns Workshop (at CHI)*, Minneapolis, USA (2002)
17. Greco de Paula, M., Santana da Silva, B., Diniz Junqueira Barbosa, S.: Using an interaction model as a resource for communication in design. In: *CHI*, Portland, USA (2005)
18. Winckler, M., Palanque, P.: StateWebCharts: A formal description technique dedicated to navigation modelling of Web applications. In: Jorge, J.A., Jardim Nunes, N., Falcão e Cunha, J. (eds.) *DSV-IS 2003*. LNCS, vol. 2844, pp. 61–76. Springer, Heidelberg (2003)
19. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a Modeling Language for Designing Web Sites. In: *WWW9 Conference* (2000)
20. Harel, D.: Statecharts: A visual Formalism for complex systems. *Science of Computer Programming* 8(3), 231–274 (1987)
21. Shneiderman, B.: *Designing the user interface. Strategies for effective human-computer interaction*. Addison-Wesley, Reading (1998)
22. Constantine, L.L., Lockwood, L.A.D.: *Software for use*. ACM Press/Addison-Wesley Publishing Co., New York (1999)
23. Van Dyne, D.K., Landay, J.A., Hong, J.I.: *The design of sites*. Addison-Wesley Professional, Reading (2002)
24. Laakso, S.A.: *User interface design patterns* (2003),  
<http://www.cs.helsinki.fi/u/salaakso/patterns/>
25. Montero, F., Lozano, M., González, P., Ramos, I.: Designing websites by using patterns. In: *SugarLoafPLOP*, Itaipava, Brasil, pp. 209–224 (2002)
26. Rossi, G., Lyardet, F., Schwabe, D.: *Patterns for e-Commerce applications*, EuroPLOP, Irsee, Germany (2000)
27. Wimmer, M.A.: *Knowledge Management in e-Government – e-Commerce vs. e-Government* (2001),  
<http://falcon.ifs.uni-linz.ac.at/research/ceepus.zip>



28. Van Welie, M.: MobileUI design patterns (2007), <http://www.welie.com/patterns/mobile/>
29. Van Welie, M.: Web design patterns (2007), <http://www.welie.com/patterns/>
30. Malone, E., Leacock, M., Wheeler, C.: Implementing a Pattern Library in the Real World: A Yahoo! Case Study ASIS&T, Montréal, Canada (2005)
31. Coram, T., Lee, J.: Experiences – A pattern language for user interface design (2002), <http://www.maplefish.com/todd/papers/Experiences.html>
32. Henninger, S., Ivaturi, A., Nuli, K., Thirunavukkaras, A.: Supporting adaptable methodologies to meet evolving project needs. In: Joint conference on XP universe and Agile universe, Chicago, Illinois, USA, pp. 33–44 (2002)
33. Ivory, M.Y., Hearst, M.A.: Improving web site design. IEEE Internet Computing 6, 56–63 (2002)
34. Nogier, J.-F.: De l'ergonomie du logiciel au design des sites Web. Dunod, Paris (2002)
35. Nielsen, J.: Alertbox (2006), <http://www.useit.com/alertbox/>

## Questions

### **Phil Gray:**

*Question: Did you include “Forces” in your pattern description?*

Answer: Yes, it is within the RATIONALE section of the pattern. Whenever possible, references to academic papers or to other pattern catalogues are given. This way, the user of our catalogue of patterns can know more, extend his study and eventually the catalogue of patterns.

*Question: Did you define relationships between patterns that, for example, point to patterns that propose alternative solutions for similar problems?*

Answer: Yes, links to related patterns are included in the pattern description. A global map of links between patterns is being developed, to allow navigation among the patterns which allows in particular the comparison of patterns between them. This mapping is an ontological mapping of the concepts appearing in the patterns.

### **Ann Blandford:**

*Question: What makes your patterns specific to e-government applications?*

Answer: The patterns we present here have been discovered while working for an e-Government enterprise. Hence they are applicable for, but not limited to, the domain of e-Government applications. Some of the patterns are also applicable in a broader context, for example the ones describing the behaviour of UI elements (cf. the MANDATORY FIELD pattern exposed in Fig. 5). By the way, a notion of standardization is included in our patterns: this is acceptable for e-Government, for the sake of UI coherence across different applications, to let the user adapt to the UI one time for all the times he will visit a governmental website. Due to marketing reasons, it would be very difficult to create uniform user interfaces in other domains such as e-Commerce.

# Including Heterogeneous Web Accessibility Guidelines in the Development Process

Myriam Arrue, Markel Vigo, and Julio Abascal

University of the Basque Country, Informatika Fakultatea,  
Manuel Lardizabal 1, E-20018, Donostia, Spain  
{myriam,markel,julio}@si.ehu.es

**Abstract.** The use of web applications has extremely increased in the last few years. However, some groups of users may experience difficulties when accessing them. Many different sets of accessibility guidelines have been developed in order to improve the quality of web interfaces. Some of them are of general purpose whereas others are specific for user, application or access device characteristics. The existing amount of heterogeneous accessibility guidelines makes it difficult to find, select and handle them in the development process. This paper proposes a flexible framework which facilitates and promotes the web accessibility awareness during all the development process. The basis of this framework is the Unified Guidelines Language (UGL), a uniform guidelines specification language developed as a result of a comprehensive study of different sets of guidelines. The main components of the framework are the guidelines management tool and the flexible evaluation module. Therefore, sharing, extending and searching for adequate accessibility guidelines as well as evaluating web accessibility according to different sets of guidelines become simpler tasks.

## 1 Introduction

In recent years, the usage of web applications has considerably extended since their usefulness has been proved in a vast variety of contexts meeting diverse needs. Companies show a growing tendency to introduce web applications in their management processes [1]. The previous business standalone applications are evolving into light web applications which have proven to be more manageable and easier to centralize. The former simple static websites have turned into unmanageable large sites which can be used for performing diverse activities. Therefore, web applications have become more complex and nowadays they integrate different technologies. According to Murugesan and Ginige [2] currently web applications can be classified in different categories depending on their functionality: informational, interactive, transactional, workflow oriented, collaborative work environments and online communities or marketplaces.

Consequently, web applications development has changed from merely being a hypertext based interface design process to a much more complex task which involves different activities such as planning, system architecture design, evaluation, quality assessment, system performance evaluation, maintenance, updates management, etc. The development of high quality web applications requires knowledge from a wide

range of disciplines such as information engineering, indexing systems, information recovery, user interface design, human-computer interaction, graphical design, etc.

Designing an appropriate user interface for these applications is probably one of the most demanding task since end-users' abilities and specific characteristics are often unknown. Under some circumstances, web applications should be designed based on "Universal Access" paradigm. This concept is turning into something extremely significant for the current Information Society as it ensures access to the information in the World Wide Web by anyone, anywhere, and at any time [3] and fosters no discrimination. Consequently, Universal Access should be an essential quality target [4] in web applications development process.

A number of initiatives have been taken in order to support the Universal Access paradigm including the promulgation, in some countries, of laws against electronic exclusion. One of the most proactive initiatives is the Web Accessibility Initiative (WAI) [<http://www.w3.org/WAI/>] that was set up by the World Wide Web Consortium [<http://www.w3.org/>]. It has published the well-known Web Content Accessibility Guidelines (WCAG) [5] which is the most universally accepted and established set of guidelines for developing and evaluating web content accessibility. It is considered that the fulfilment of these guidelines ensures that the developed web application is accessible to some extent by all people.

Even though all these efforts are extremely useful for developing accessible web applications and have extended the awareness of accessibility among web developers community, they have proven not to be sufficient in order to achieve the Universal Access. Therefore, some groups of users are still experiencing accessibility problems when interacting with the majority of existing web applications.

This situation has lead to the development of large amount of web accessibility guidelines in recent years. These guidelines aim to improve users' experience when using services in the World Wide Web. Nowadays, in addition to general purpose guidelines such as WCAG, other sets of guidelines related to specific application type (e-learning, e-commerce, etc.), specific users' characteristics (elderly, children, deaf, etc.) and accessing devices (mobile devices, etc.) can be found. Some sets of guidelines can be built combining the mentioned guidelines, e.g.: guidelines for e-learning applications for children.

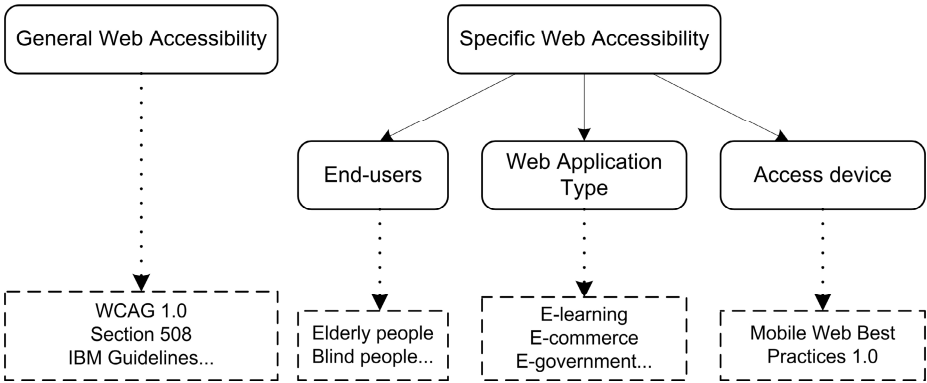
According to Mariage et al. [6], current accessibility sets of guidelines are defined based on different formats, they may include different contents and are defined in different level of detail. Guidelines range from specific rules to common sense statements. Thus, existing accessibility guidelines can be classified in different groups depending on their level of detail. In this sense, Figure 1 depicts the different types of existing web accessibility sets of guidelines.

Consequently, web developers should analyse the existing accessibility knowledge in order to select the most adequate guidelines, techniques and methods for their developments. In this sense, web developers usually have to deal with diverse complex tasks [7]:

- Search for the sets of guidelines which are significant for the current development.
- Select the most adequate sets of guidelines.
- Verify the coherence of the selected sets of guidelines.

- Analyse the applicability of the selected guidelines in the current development.
- Develop directly applicable design rules from the selected guidelines.
- Plan and perform frequent accessibility evaluations based on the selected sets of guidelines during the development process.

**Web Accessibility Knowledge**



**Fig. 1.** A taxonomy for web accessibility sets of guidelines

Due to the diversity of formats and structures used for defining accessibility guidelines, finding, selecting, applying and evaluating these guidelines are tedious tasks for practitioners. There are several automatic tools which assist developers evaluating the accessibility of web pages but most of them are based on general purpose sets of guidelines. Therefore, they are not flexible enough to evaluate guidelines for specific application type, user type or access device.

This paper proposes a framework for flexible web accessibility development. It will assist web developers to evaluate web interfaces according to the selected sets of guidelines. In addition, it will be useful during all the development process since it will provide several functionalities for guidelines definition, edition, searching and sharing. The basis for the development of such a framework is to define a unified definition language for accessibility guidelines so different formats and contents can be accommodated. In this sense, a comprehensive analysis of diverse sets of guidelines has been carried out and the results are outlined in Section 3. The rest of the paper is structured as follows: Section 2 is dedicated to present the related work and Section 4 describes the implementation of the evaluation logic and reporting process of the developed framework. Finally, the reached conclusions are discussed in Section 5.

**1.2 The Role of Accessibility Evaluation**

Evaluating accessibility is an essential stage in the development of accessible web applications. This process will confirm if the selected guidelines have been fulfilled.

Diverse accessibility evaluations have to be performed in order to detect any possible barrier and repair it. In this sense, two different scenarios are considered: proactive and reactive evaluation. The former, concerns to the accessibility aware design and the later relates to the final application accessibility checking. Both scenarios require evaluations, including the proactive one as suggested in [8]. Performing comprehensive evaluations implies combining diverse kind of evaluations:

- Automatic evaluation with tools: this is a preliminary test stage aiming to remove the first and most "evident" obstacles. "Evident" means those errors automatically testable with the help of tools. According to Lang [9], this evaluation method presents diverse advantages in terms of costs and efficiency as the automatic evaluation tools report detected errors in a short period of time. Ivory provides a comprehensive description of different automatic evaluation methods and tools in [10]. The aim of this evaluation is to clear up the content so that forthcoming evaluations with experts and users take less time in order to focus on other complex issues. An effective evaluation tool should be able to validate the fulfilment of most of the guidelines. Yet, nowadays it is a far objective since there is not enough research done to evaluate some checkpoints such as WCAG 1.0 14.1 checkpoint: "Use the clearest and simplest language appropriate for a site's content". In addition, most of automatic accessibility evaluation tools only check the conformance with general purpose guidelines such as WCAG 1.0, Section 508 [11], etc. They are not flexible enough to evaluate other sets of guidelines or new versions as the evaluated guidelines are built-in within the source code. Consequently, incorporating new guidelines implies modifying the code of the tool. In this sense, the separation between guidelines and evaluation engine ensures the required flexibility.
- Expert driven manual evaluations: as previously mentioned the evaluation of some guidelines requires human judgement. Web accessibility experts perform evaluations based on heuristics in order to evaluate this kind of guidelines. Main tasks have to be defined and walkthroughs with different browsers, assistive technologies, devices, etc. are carried out. These evaluation methods will allow detecting accessibility barriers when the web application is used under different conditions as explained in [12].
- Evaluations with users: this evaluation type is essential since it allows detecting real accessibility barriers for users with specific characteristics. Selected users should cover the broader range of disabilities if a comprehensive evaluation is required. Users are asked for performing tasks coinciding commonly with the main functionalities of the web application. Evaluations are usually carried out in controlled environments such as specific laboratories where the experts can observe the actions of the users and gather information about the interaction following accepted usability evaluation techniques such as the ones described by Nielsen and Mack [13] and Rubin [14]. However, results obtained from remote evaluations carried out in users' common browsing environment can be also useful as mentioned in [15].

All these evaluations are complementary and necessary. If only automatic evaluation is carried out the fulfilment of several guidelines will not be checked and the required minimum accessibility level is seldom reached. On the other hand, evaluations with users also help finding out usability barriers which accessibility guidelines and therefore automatic accessibility evaluation tools do not consider. The final objective of these evaluations is to repair the detected errors. As justified above, automatic accessibility evaluation is a necessary task indeed.

## 2 Related Work

As previously mentioned, the basis for the development of a framework for flexible web accessibility evaluation is to separate the definition of guidelines and the evaluation logic. This objective is achieved by defining a language for guidelines specification independent of the evaluation engine. Thus, the defined grammar should be flexible enough to define forthcoming versions of existing guidelines, updates and new guidelines sets. In this sense, several approaches can be found in the literature.

In 2004, Abascal et al. [16] proposed the novel approach for automatic accessibility evaluation: separation of guidelines from the evaluation engine. The usefulness of this approach relies on its flexibility and updating efficiency. Adaptation to new guideline versions does not imply re-designing the evaluation engine but guidelines editing. The guidelines specification language is based on XML.

Following this first approaches, in 2005, Vanderdonckt and Bereikdar proposed the Guidelines Definition Language, GDL [17] and recently Leporini et al. the Guidelines Abstraction Language, GAL [18]. All these guideline specification languages make possible adapting quite straightforwardly to new guideline versions or novel guidelines.

However, these guidelines specification languages are mostly based on general purpose accessibility sets of guidelines. Consequently some specific purpose guidelines may not be defined since previous study of specific accessibility sets of guidelines and their definition in those languages is not provided. In addition, the developed definition languages are quite complex and appropriate tools for defining, editing, sharing and searching for accessibility information are needed. A new framework should be developed in the basis of a comprehensive study of different sets of guidelines and with the aim of assisting web developers during all the development process.

As far as evaluation logic is concerned, there is a growing tendency towards using XML querying languages. These languages are very powerful due to their expressiveness and flexibility. Takata et al. [19] proposed a pseudo-XQuery language for accessibility evaluation purposes and XPath/XQuery sentences are defined to check WCAG guidelines in [20]. We have adopted this technology in our new approach since it allows us to design complex queries. As a result, lots of source code lines are saved.

## 3 Uniform Accessibility Guidelines Definition

We did not predict in 2004 the new amount of guidelines sets appeared, referring to specific user groups, environments or accessing devices. Therefore, a study of existing

sets of guidelines has been carried out and a process for guidelines format standardization has been performed. As a result, it has been defined a new guidelines definition language: Unified Guidelines Language, UGL. The strength of our approach relies on the flexibility of the grammar since it has been defined after studying different sets of guidelines. It is flexible enough to define the guideline sets analysed in this paper and it also allows validating documents according to other criteria. The following table, Table 1, shows some sets of guidelines analysed and their classification regarding the taxonomy presented previously.

**Table 1.** Information about the analysed sets of web accessibility guidelines

<b>Name</b>	<b>Type</b>	<b>Description</b>
WCAG 1.0 [5]	General Web Accessibility	Web Content Accessibility Guidelines 1.0
Section 508 [11]	General Web Accessibility	Section 508 of the Rehabilitation Act
IBM [21]	General Web Accessibility	IBM Accessibility Center: Developer Guidelines for Web Accessibility
CPB/WGBH [22]	Specific Application Type	Making Educational Software and Web Sites Accessible
WDGOP [23]	Specific Users' Characteristics	Research-Derived Web Design Guidelines for Older People
MWBP [24]	Specific Access Device	Mobile Web Best Practices

The developed language should be comprehensive enough to specify different information type: general information about the sets of guidelines, guidelines and methods or techniques and specific information for evaluation purposes such as evaluation procedures or test cases. In addition, the objective is to design a language which could be easily understood by web developers and accessibility experts, so that they are encouraged to specify new guidelines or new interpretations, incorporate them into the framework and share them with other users. The following sections present the fields included in the structure for each type of information.

### 3.1 General Information

This information type refers to general information about the set of guidelines and methods or techniques which will not be processed by the accessibility evaluation tool.

- **Guideline set information:** this type of information is necessary for defining the general information about the set of guidelines. For instance, the classification of the set of guidelines according to the previously presented taxonomy.
- **General guideline information:** the necessary information for specifying each design guideline is specified, such as title, description and so on.
- **Methods or techniques information:** this information is necessary for training purposes so any web designer could find methods, techniques or examples of how to conform to the accessibility guidelines. This information is useful through all the web applications development phase.

General information about guidelines and sets of guidelines can be easily obtained from guidelines documents whereas specification of methods, techniques or examples requires some interpretation depending on the level of detail of guidelines. For instance, among the selected sets of guidelines the WDGOP are not defined in low level of detail. Therefore they require more effort to be interpreted and to define the methods or techniques.

### 3.2 Information for Evaluation Purposes

This information type refers to the necessary evaluation procedures for each guideline. Incorporating this information into the language schema will ensure that automatic accessibility evaluations will be possible for guidelines defined in this format.

However, not all web accessibility guidelines can be automatically evaluated. Therefore, they can be specified only with general information. For instance, the following guideline: “Use the clearest and simplest language appropriate for a site’s content” can not be validated by automatic tools since it requires human judgment. There is another type of guidelines that can not be automatically evaluated but can be triggered by tools. For instance, one of these guidelines is: “Organize documents so they may be read without style sheets. For example, when an HTML document is rendered without associated style sheets, it must still be possible to read the document”. An automatic tool can detect that a web page is associated with a style sheet but up to date it is not possible to automatically validate if the web page is well organized. Since this type of issues can be triggered by the content, they are known as semi-automatic test cases. An automatic evaluation tool will produce a *warning* if a semi-automatic test case is detected. On the other hand, an *error* will be produced if an automatic test case (a test case which can be evaluated automatically) is not fulfilled.

These automatic and semi-automatic test cases have to be defined in the language in order to ensure that the automatic evaluation process will be effectively performed. For this reason, different fields and values for defining test cases have to be incorporated into the language. The evaluation procedures for the guidelines contained in the different sets of guidelines have been analysed. This process has detected all the different semi-automatic and automatic test cases. Some of these test cases are simply validated analysing one HTML element such as IMG, TABLE, FRAME etc. whereas other type of test cases require analysing HTML elements and their attributes such as TYPE attribute of INPUT element, ALT attribute of IMG element, TITLE attribute of A element, etc. In addition, there are some complex test cases that require analysing one HTML element, its attributes and other associated HTML elements, for instance, a INPUT element with a value in its ID attribute requires the existence of a LABEL element with the same value in its FOR attribute.

All the different types of automatic and semi-automatic test cases defined in the analysed sets of guidelines have been compiled and are described in the following tables, Table 2, Table 3 and Table 4.



**Table 2.** This table shows the automatic (A) and semi-automatic (SA) test cases requiring only the analysis of HTML elements

No.	Test Case Name	Description	Example	Type
1	Deprecated	The HTML element is deprecated.	<i>WCAG 1.0 Checkpoint 11.2 FONT</i>	A
2	Compulsory	The element is compulsory.	<i>WCAG 1.0 Checkpoint 3.2 DOCTYPE</i>	A
3	Text Required	A string is required between the open and close tags of the element.	<i>Section 508 Checkpoint (a) &lt;APPLET&gt;Text&lt;/APPLET&gt;</i>	A
4	Avoid	It is recommended to avoid using the HTML element.	<i>WDGOP Checkpoint 9.1 MARQUEE</i>	A
5	Warning Produced	Using the HTML element may cause accessibility problems and have to be tested manually.	<i>Section 508 Checkpoint (m) OBJECT</i>	SA
6	Element Needed	Another HTML element is required.	<i>IBM Checkpoint 9 FRAMESET NOFRAMES</i>	A

**Table 3.** This table shows the automatic (A) and semi-automatic (SA) test cases requiring the analysis of HTML elements as well as their attributes

No.	Test Case Name	Description	Example	Type
7	Compulsory	The attribute is compulsory.	<i>WCAG 1.0 Checkpoint 1.1 IMG ALT</i>	A
8	Compulsory Not Empty	The attribute is compulsory and it must have some value.	<i>IBM Checkpoint 9 FRAME TITLE</i>	A
9	Recommended	This attribute is recommended.	<i>CPB/WGBH Checkpoint 1.1 IMG LONGDESC</i>	A
10	Warning Produced	Using the attribute may cause accessibility problems and have to be tested manually.	<i>IBM Checkpoint 5 TABLE ONCLICK</i>	SA
11	Attribute Needed	Another attribute is required.	<i>IBM Checkpoint 5 SELECT ONBLUR ONFOCUS</i>	A
12	Error Produced	Use of this attribute must be avoided.	<i>WDGOP Checkpoint 1.3 INPUT ONDBLCLICK</i>	A
13	Determined Value	The value of the attribute has to be one of some specifically defined.	<i>WCAG 1.0 Checkpoint 4.3 HTML LANG= en, es, fr...</i>	A
14	Determined Part of Value	The value of the attribute must contain a determined value.	<i>WCAG 1.0 Checkpoint 3.4 TABLE WIDTH =%, em</i>	A
15	Avoid Value	Avoid a specified value for an attribute.	<i>WCAG 1.0 Checkpoint 7.4 META HTTP-EQUIV=refresh</i>	A
16	Value Warning	A value of an attribute may cause accessibility problems and have to be tested manually.	<i>CPB/WGBH Checkpoint 2.2 A HREF=.wav</i>	SA
17	Value Attribute Requires Not Empty	A specific value of an attribute requires another attribute which must have some value.	<i>IBM Checkpoint 1 INPUT TYPE=img ALT</i>	A

**Table 4.** This table shows the automatic (A) and semi-automatic (SA) test cases requiring the analysis of associated HTML elements and their attributes

No.	Test Case Name	Description	Example	Type
18	Attribute requires an Element Determined Value	which contains an <i>CPB/WGBH Checkpoint 1</i> with specific attribute requires the <i>1.1</i> existence of another element with determined value.	<i>IMG LONGDESC</i> <A...>D</A>	A
19	Nested Element Not Empty Attribute	An element nested inside another HTML element requires an attribute which must have some value.	<i>IBM Checkpoint 1</i> <A...> <IMG TITLE=value> </A>	A
20	Elements Needed for Specific Attribute	An attribute requires the existence of a minimum number of occurrences of elements.	<i>IBM Checkpoint 2</i> <i>IMG ISMAP</i> <i>A element occurrences ≥ 2</i>	A
21	Attribute Value requires Element with Attribute Value	An attribute value requires the existence of an element with determined attribute value.	<i>Section 508 Checkpoint</i> <i>INPUT id=value</i> <i>LABEL for=value</i>	A

### 3.3 Unified Guidelines Language, UGL

We have considered all test cases' characteristics in order to develop a common language to frame them. As a result, Unified Guidelines Language (UGL) is the resultant language which is defined according to a grammar defined in a XML-Schema. This language provides the necessary mechanisms for defining test cases for any mark-up language since it allows performing the following operations with the content within the opening and closing of a determined tag and with attribute values:

- Boolean operations
- Logical operations
- Dictionary queries for comparisons with large sets of words. E.g. checking the validity of the document language: en-us, en-gb, fr, eu, es...
- Counting

It is necessary to specify the relationships between different elements (labels and attributes) in the (X)HTML document. In addition, evaluation scope within the document can be set.

- Analyse HTML elements
- Analyse attributes within HTML elements
- Analyse associated elements of attributes and labels. There are infinite combinations since our schema is defined recursively. Therefore, it is possible to specify the following relationship: one label with a determined attribute requires a determined label with a determined attribute; one attribute requires a label (which is not its parent) with a determined attribute which at the same time requires another label and so on. Some relationships are unnecessary and useless but can be used in some contexts. However they are useful to demonstrate the flexibility of the language and future versions of guidelines could take advantage of them.

Both XML-Schema and its graphical representation are rather large and it is out of the objective of this paper to explain thoroughly all the features of UGL. If further information is required in this sense, both schema and its picture can be found in our project's website<sup>1</sup>.

However, relationships between different guidelines sets and its evaluation procedures can be modelled in a static diagram so that the readership could get a general idea. Figure 2 models the XML-Schema of UGL.

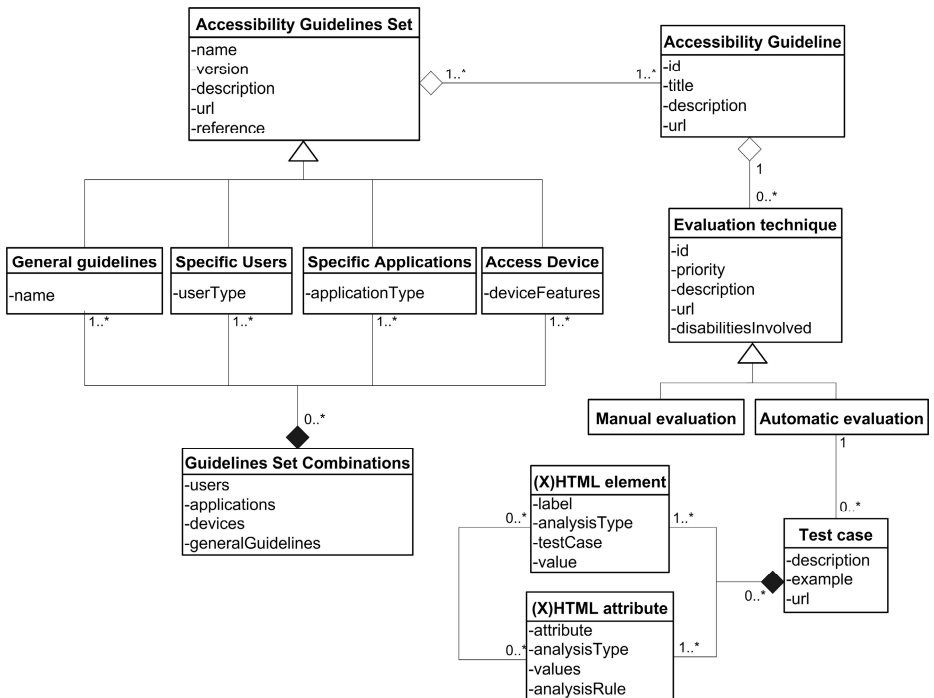


Fig. 2. A model of the relationships among entities in XML-Schema of UGL

### 3.4 Web Interface for Guidelines Management

Expert users may prefer to directly specify guidelines in UGL and upload them to the framework but novel users may get confused due to the complexity of the definition language. Therefore, a web application which guides the user specifying guidelines has been developed. Since it is accessible from the browser it has some advantages over other approaches such as the ones proposed by Mariage et al. [25] and Leporini et al. [26]. Both aim at abstracting the interaction with accessibility guidelines with graphical interfaces. Unfortunately, both are standalone applications which have some drawbacks compared with a web application.

<sup>1</sup> XML-Schema of UGL: <http://sipt07.si.ehu.es/evalaccess3/ugl.xsd>. Its representation: <http://sipt07.si.ehu.es/evalaccess3/ugl.png>

Managing guidelines with a web application makes possible to have a centralized repository of guidelines. Hence, all users that sign up in the system are able to access and make evaluations with them, as well as search for specific guidelines. In addition, guidelines creators can set permissions to guideline sets such as *shared* and *shared but not editable*. The interaction is via XHTML forms and the browser is the interface between the user and the system which is accessible for everybody. Consequently, no plug-ins or software installations are required. As a result, this guidelines management interface leads to bridge the gap between developers and researchers since it is useful for knowledge sharing in this area.

The guidelines management interface is integrated in the evaluation framework proposed in this paper. Users are capable to search for guidelines and creating personal sets in order to perform automatic evaluations with them. Figure 3 shows a screenshot of the guidelines management application.

**Fig. 3.** Interface for guidelines management

Guidelines are stored in a relational data base. As soon as the guideline creation/edition process is concluded they are transformed into UGL. This transformation is automatically performed and the resulting UGL document is stored in a XML native data base afterwards.

## 4 Evaluating and Reporting

The final objective of the framework is to evaluate web pages against the guideline sets stored in the guidelines repository. Thus, the management interface integrates

into the whole guidelines evaluation framework and makes possible evaluating desired guidelines sets according to the requirements for a given development. However, in order to avoid searching, selecting repeatedly guidelines every time the user logs in the system, preferences regarding guideline sets will be saved in user's profile and there will be no need to repeat the process again. Therefore, unless new guidelines are required or existing ones changed, user's preferences are stored for forthcoming accesses. In order to explain the definition, evaluation and reporting stage, the evaluation of two test cases is going to be described step by step in the following subsections.

#### 4.1 Test Cases Definition

Test case number 17 states: "a specific value of an attribute requires another attribute which must have some value". This test case includes examples defined in IBM Checkpoint 1 and their corresponding specification in UGL.

- Example 1: *INPUT type="img" → ALT*. If value of type attribute in element input is "img" an alternative description is required. The processing information for this test case is specified in UGL as follows:

```
<label>INPUT</label>
<analysis_type>check attribute</analysis_type>
<related_attribute>
  <atb>TYPE</atb>
  <analysis_type>value</analysis_type>
  <analysis_type>check attribute</analysis_type>
  <content test = "=">img</content>
  <related_attribute>
    <atb>ALT</atb>
    <analysis_type>compulsory</analysis_type>
  </related_attribute>
</related_attribute>
```

- Example 2: *INPUT name="go" → ALT*. If value of name attribute in element input is "go" an alternative description is required. The processing information for this test case is specified in UGL as follows:

```
<label>INPUT</label>
<analysis_type>check attribute</analysis_type>
<related_attribute>
  <atb>NAME</atb>
  <analysis_type>value</analysis_type>
  <analysis_type>check attribute</analysis_type>
  <content test = "=">go</content>
  <related_attribute>
    <atb>ALT</atb>
    <analysis_type>compulsory</analysis_type>
  </related_attribute>
</related_attribute>
```

Test case 19 states that "An element nested inside another HTML element requires an attribute which must have some value". Its UGL representation:

```
<label>A</label>
<analysis_type>check element</analysis_type>
<related_element scope="inside">
  <label>IMG</label>
  <analysis_type>check attribute</analysis_type>
  <related_attribute>
    <atb>TITLE</atb>
    <analysis_type>compulsory</analysis_type>
    <analysis_type>value</analysis_type>
    <content test="not empty"></content>
  </related_attribute>
</related_element>
```

Fields in bold are the ones *editable* in each test case. In other words, they are the unique fields that when changing their value, the previously stated description still maintains its meaning. They are the fields that would play the role of variables in each test case as explained in the next section.

## 4.2 Evaluation

As mentioned in Section 2, existing novel approaches for Web documents evaluation published by Takata et al. [18] and Luque et al. [19] are the basis for our research. XQuery is a powerful query language for gathering information from XML documents quite straightforwardly. In our previous work [15], DOM and SAX technologies were used to navigate through the XML tree and the implementation required a big amount of source code compared with XQuery. Therefore, we have implemented a XQuery sentence for each test case.

Obviously, it is necessary to transform the original HTML document into XML when it comes to the evaluation of non XHTML files. JTidy<sup>2</sup> and Neko<sup>3</sup> parsers are commonly used for this task in Java environments.

All types of test cases defined in Table 2, Table 3 and Table 4 are linked to a XQuery template. This relationship is implicitly declared in a field of every test case in the UGL document. The templates contain gaps such as element name, attribute name, attribute value etc. which are filled out in a mapping process from UGL to XQuery sentences. These gaps are the previously mentioned *editable* fields and are mapped as soon as UGL guidelines have been built. Once XQuery sentences are ready, evaluation of web pages is performed by applying XQuery sentences to the web page in (X)HTML. Figure 4 depicts the template for test case 17 and shows how values in UGL test cases are mapped there. Figure 5 shows a more complex query.

Guidelines in UGL are useful for guidelines definition by experts. In this case, the expert can directly access and edit the UGL document without using the web interface. It is faster but it requires knowledge of the UGL language. Guidelines in UGL are also necessary in order to show their content in the Web interface while guidelines editing or extending. It takes less effort transforming a mark-up language

<sup>2</sup> JTidy HTML parser. Available at <http://jtidy.sourceforge.net/>

<sup>3</sup> CyberNeko HTML Parser 0.9.5. Available at <http://people.apache.org/~andyc/neko/doc/html/>

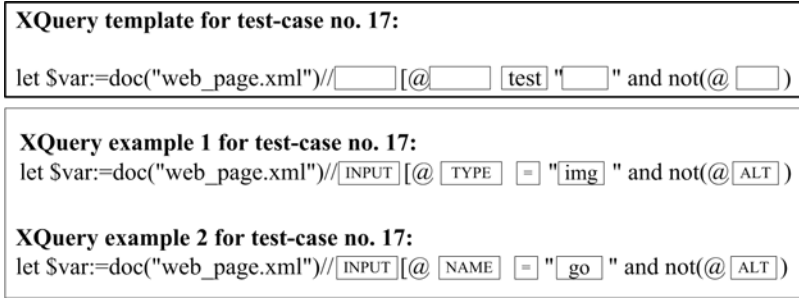


Fig. 4. XQuery template and sentences derived from test case no. 17 in UGL

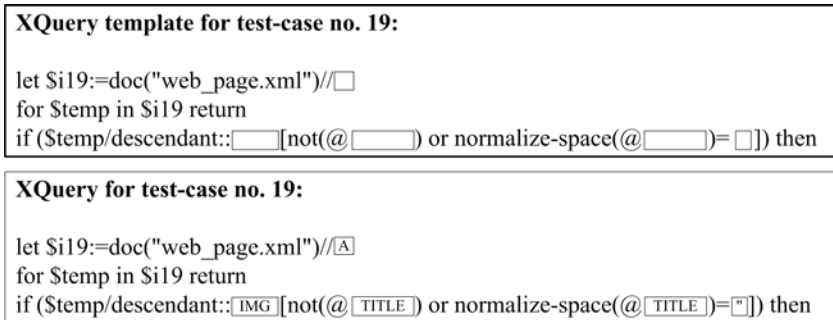


Fig. 5. XQuery template and sentence derived from test case no. 19 in UGL

such as UGL than XQuery for web publishing. In addition, since the guidelines management interface allows the user searching for guidelines, we take advantage of the facilities of the XML data base as data in relational data base data are spread in different tables and requires complex queries. Therefore, XQuery is used for evaluation purposes and UGL for guidelines definition, web publishing and guidelines search.

### 4.3 Reporting

The developed XQuery sentences also include useful information for detected errors reporting and reparation purposes such as the line in the (X)HTML document where the error has occurred and which element and attribute have provoked it. This information and general information stored in UGL guidelines are put together in XML reports. This information is highlighted in the following example.

#### XQuery sentence

```
let $var:=doc("web_page.xml")/INPUT[ @type='img' and not( @alt)
for $temp in $var
return
<test_case no="17" type="error">
<label>{ $temp/@line, $temp/name() }</label>
<attribute>type</attribute>
</test_case>
```

**UGL guideline**

```

<checkpoints id="1">
<priority>1</priority>
<evaluation_type>auto</evaluation_type>
<description>Provide alternative content for visual content</description>
<url>http://www-306.ibm.com/able/guidelines/web/webimages.html</url>
  <techniques id="1">
    <code>HTML</code>
    <description>Provide alternative content to images</description>
    <disabilities>blind</disabilities>
  </url>http://www-306.ibm.com/able/guidelines/web/webimages.html#techniques</url>

```

**Final report**

```

<checkpoint id="1">
<test_case no="17" type="error">
  <description>Provide alternative content for visual content</description>
  <url>http://www-306.ibm.com/able/guidelines/web/webimages.html</url>
  <techniques id="1">
    <description>Provide alternative content to images</description>
    <url>http://www-306.ibm.com/able/guidelines/web/webimages.html#techniques</url>
    <priority>1</priority>
    <label line="35">INPUT</label>
    <attribute>alt</attribute>
  </techniques>
</test_case>
</checkpoint>

```

Nowadays accessibility evaluation tools reports do not have a uniform reporting format. EARL [27] is a RDF-based language supported by the W3C which aims at being the standard language for general reporting. Standardization of the reporting format in web accessibility evaluation area is really useful since it will make possible automatically comparing the same evaluation made by different tools, keeping track of web accessibility evolution, etc. When a stable version of EARL is finally released the transformation of our evaluation report will be quite straightforward as it is XML-based.

**5 Conclusions**

The proposed framework assists web developers in developing accessible web applications. It is useful and reliable throughout the development process as different functionalities have been included. In this sense, web developers can edit, update, search for guidelines, include new accessibility guidelines as well as select guidelines for performing automatic accessibility evaluations. Consequently, it is flexible enough to facilitate the development of web applications according to diverse sets of guidelines.



In addition, all the functionalities included in the framework would allow creating a comprehensive repository of accessibility guidelines which could be shared among developers community. A web interface has been also developed for facilitating the access to the functionalities developed in order to assist developers with diverse level of experience.

The basis of the proposed framework is the UGL, Unified Guidelines Language. This guidelines specification language has been developed based on a comprehensive study of different types of accessibility guidelines. As a result, it integrates the necessary elements for defining a wide range of test cases. Moreover, the components integrated in this language will make possible to specify most of future versions of the existing sets of guidelines.

As far as the evaluation task is concerned, novel approaches based on XML querying technology such as XPath/XQuery are presented as well as the transformation mechanism from UGL to XQuery sentences. The use of these technologies provides a flexible evaluation module which can be easily extended in order to incorporate new features. The flexible reporting of detected errors has been also considered and will be easily updated for accommodating future standard reporting languages such as EARL.

## Acknowledgements

Work of Markel Vigo is funded by the Department of Education, Universities and Research of Basque Government.

## References

1. Hoffman, D., Grivel, E., Battle, L.: Designing software architectures to facilitate accessible Web applications. *IBM Systems Journal* 44(3), 467–483 (2005)
2. Murugesan, S., Ginige, A.: *Web Engineering: Introduction and Perspectives*. In: Suh, W. (ed.) *Web Engineering: Perspectives and Techniques*. Idea Group (2005)
3. Stephanidis, C., Savidis, A.: *Universal Access in the Information Society: Methods, Tools, and Interaction Technologies*. *Universal Access in the Information Society* 1(1), 40–55 (2001)
4. Savidis, A., Stephanidis, C.: *Unified user interface development: the software engineering of universally accessible interactions*. *Universal Access in the Information Society* 3(3–4), 165–193 (2004)
5. Chisholm, W., Vanderheiden, G., Jacobs, I. (eds.): *Web Content Accessibility Guidelines 1.0* (May 5, 1999), <http://www.w3.org/TR/WAI-WEBCONTENT/>.
6. Mariage, C., Vanderdonckt, J., Pribeanu, C.: *State of the Art of Web Usability Guidelines*, ch. 41. In: *The Handbook of Human Factors in Web Design*. Lawrence Erlbaum, Mahwah (2005)
7. Abascal, J., Nicolle, C.: *Why Inclusive Design Guidelines*, ch. 1. In: Abascal, J., Nicolle, C. (eds.) *Inclusive Design Guidelines for HCI*. Taylor & Francis, Abington (2001)
8. Luque, V., Delgado, C., Gaedke, M., Nussbaumer, M.: *Web Composition with WCAG in mind*. In: *Proceedings of the 2005 International Cross-Disciplinary Workshop on Web Accessibility (W4A)*, pp. 38–45 (2005)

9. Lang, T.: Comparing website accessibility evaluation methods and learnings from usability evaluation methods (2003), [http://www.peakusability.com.au/pdf/website\\_accessibility.pdf](http://www.peakusability.com.au/pdf/website_accessibility.pdf)
10. Ivory, M.Y.: Automated Web Site Evaluation: Researchers' and Practitioners' Perspectives. Kluwer Academic Publishers, Dordrecht (2003)
11. Center for IT Accommodation (CITA) U.S. Section 508 Guidelines, <http://www.section508.gov>
12. Brajnik, G.: Web Accessibility Testing: When the Method Is the Culprit. In: Miesenberger, K., Klaus, J., Zagler, W.L., Karshmer, A.I. (eds.) ICCHP 2006. LNCS, vol. 4061, pp. 234–241. Springer, Heidelberg (2006)
13. Nielsen, J., Mack, R.: Usability Inspection Methods. John Wiley & Sons, New York (1994)
14. Rubin, J.: Handbook of Usability Testing. John Wiley & Sons, New York (1994)
15. Petrie, H., Hamilton, F., King, N., Pavan, P.: Remote usability evaluations with disabled people. In: Proceedings of the SIGCHI conference on Human Factors in computing systems (CHI 2006), pp. 1133–1141 (2006)
16. Abascal, J., Arrue, M., Fajardo, I., Garay, N., Tomás, J.: The use of guidelines to automatically verify Web accessibility. *Universal Access in the Information Society* 3(1), 71–79 (2004)
17. Vanderdonckt, J., Bereikdar, A.: Automated Web Evaluation by Guideline Review. *Journal of Web Engineering* 4(2), 102–117 (2005)
18. Leporini, B., Paternò, F., Scordia, A.: Flexible tool support for accessibility evaluation. *Interacting with Computers* 18(5), 869–890 (2006)
19. Takata, Y., Nakamura, T., Seki, H.: Accessibility Verification of WWW Documents by an Automatic Guideline Verification Tool. In: Proceedings of the 37th Hawaii International Conference on System Sciences (2004)
20. Luque, V., Delgado, C., Gaedke, M., Nussbaumer, M.: Proceedings of the 14th international conference on World Wide Web, WWW 2005, pp. 1146–1147 (2005)
21. IBM Accessibility Center: Developer guidelines for Web Accessibility, <http://www-306.ibm.com/able/guidelines/web/accessweb.html>
22. Freed, G., Rothberg, M., Wlodkowski, T.: Making Educational Software and Web Sites (2003), <http://ncam.wgbh.org/cdrom/guideline/>
23. Kurniawan, S., Zaphiris, P.: Research-derived web design guidelines for older people. In: Proceedings of the ACM SIGACCESS Conference on Computers and Accessibility (ASSETS 2005), pp. 129–135 (2005)
24. Rabin, J., McConchie, C. (eds.): Mobile Web Best Practices (W3C Candidate Recommendation) (June 27, 2006), <http://www.w3.org/TR/mobile-bp/>
25. Mariage, C., Vanderdonckt: Creating Contextualised Usability Guides for Web Sites Design and Evaluation. In: Jacob, R., et al. (eds.) Proceedings of the 5th International Conference on Computer-Aided Design of User Interfaces, CADUI 2004, pp. 147–158 (2004)
26. Leporini, B., Paternò, F., Scordia, A.: An Environment for Defining and Handling Guidelines for the Web. In: Miesenberger, K., Klaus, J., Zagler, W.L., Karshmer, A.I. (eds.) ICCHP 2006. LNCS, vol. 4061, pp. 176–183. Springer, Heidelberg (2006)
27. Abou-Zahra, S., McConchie, C.: Evaluation and Report Language (EARL) 1.0 Schema (Working draft) (September 27, 2006), <http://www.w3.org/TR/EARL10/>

## Questions

***Fabio Paterno:***

*Question: How did you calculate the line number where the error occurred?*

Answer: This is done by the parser.

# Author Index

- Abascal, Julio 620  
Arrue, Myriam 620
- Back, Jonathan 18  
Bandelloni, Renata 285  
Barboni, Eric 321  
Barralon, Nicolas 537  
Blandford, Ann 18, 53, 227  
Bock, Carsten 158  
Bouchet, Jullien 36  
Brüning, Jens 175
- Calvary, Gaëlle 140  
Campos, José Creissac 193  
Chalin, Patrice 71  
Chatty, Stéphane 356  
Clerckx, Tim 89, 447  
Coninx, Karin 89, 447  
Connell, Iain 53  
Conversy, Stéphane 321  
Coutaz, Joëlle 140, 537  
Curzon, Paul 18, 227
- Deissenboeck, Florian 106  
Dewan, Prasun 393  
Dittmar, Anke 175  
Dix, Alan 210  
Dubois, Emmanuel 465
- Favre, Jean-Marie 140  
Fish, Andrew 413  
Forbrig, Peter 175  
Friday, Adrian 210
- González, Pascual 374  
Gow, Jeremy 501  
Graf, Christian 586  
Graham, T.C. Nicholas 339  
Gray, Philip 465  
Green, Thomas R.G. 53
- Harrison, Michael D. 193, 243  
Hertzum, Morten 483
- Kadner, Kay 275  
Kerkow, Daniel 555  
Khazaei, Babak 413
- Khendek, Ferhat 71  
Kohler, Kirstin 555, 586  
Kotzé, Paula 567  
Kray, Christian 243
- Leite, Jair 210  
Limbourg, Quentin 601  
López-Jaquero, Víctor 374  
Luyten, Kris 447
- Macías, José A. 303  
Madani, Laya 36  
Markopoulos, Panos 429  
Mommel, Thomas 158  
Mena, Eduardo 1  
Metaxas, Georgios 429  
Mitrović, Nikola 1  
Montero, Francisco 374  
Mueller, Stephan 275
- Navarre, David 321  
Nebe, Karsten 123  
Niebuhr, Sabine 586  
Nigay, Laurence 36
- Oriat, Catherine 36
- Palanque, Philippe 321  
Papatzani, Georgios 227  
Parissis, Ioannis 36  
Paternò, Fabio 285, 303  
Pontico, Florence 601
- Reichart, Daniel 175  
Reiterer, Harald 158  
Renaud, Karen 567  
Roast, Chris 413  
Rose, Tony 53  
Royo, Jose A. 1  
Rukšėnas, Rimvydas 18
- Santoro, Carmen 285  
Sinnig, Daniel 71  
Sottet, Jean-Sébastien 140  
Sun, Zhiyu 243

Thimbleby, Harold 501, 520

Thimbleby, Will 520

Vanderdonckt, Jean 374

Vandervelpen, Chris 89

Vandriessche, Yves 447

van Tonder, Bradley 260

Vermeulen, Jo 447

Vigo, Markel 620

Wagner, Stefan 106

Wesson, Janet 260

Winckler, Marco 601

Winter, Sebastian 106

Wu, James 339

Zhang, Huqiu 243

Zimmermann, Dirk 123