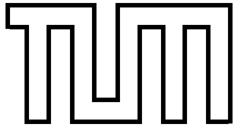
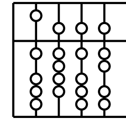


Technische Universität
München
Fakultät für Informatik

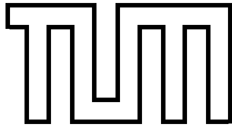


Diplomarbeit

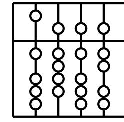
Interaction Management for Ubiquitous Augmented Reality User Interfaces

CAR
Car Augmented Reality

Otmar Hilliges



Technische Universität
München
Fakultät für Informatik



Diplomarbeit

Interaction Management for Ubiquitous Augmented Reality User Interfaces

CAR
Car Augmented Reality

Otmar Hilliges

Aufgabensteller:	Prof. Gudrun Klinker, Ph.D.
Betreuer:	Dipl.-Inf. Christian Sandor
Abgabedatum:	15. Juni 2004

Ich versichere, dass ich diese Diplomarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Juni 2004 Otmar Hilliges

Abstract

One of the major challenges of current computer science research is to provide users with suitable means of interaction with increasingly powerful and complex computer systems. In recent years several concepts in user interface technologies and human computer interaction have been evolved. Among them *augmented*, *mixed* and *virtual reality*, *tangible*, *ubiquitous* and *wearable* user interfaces. All these technologies are, more and more, converging into a new user interface paradigm which we call *Ubiquitous Augmented Reality*.

Ubiquitous Augmented Reality user interfaces incorporate a wide variety of concepts such as multi-modal, multi-user and multi-device aspects. Also these include new input and output devices. In contradiction to classic 2D user interfaces, there has no standardization taken place for ubiquitous augmented reality user interfaces' input and output devices nor for the interaction techniques utilized in such user interfaces.

This thesis presents a method that handles interaction management for ubiquitous augmented reality user interfaces, consisting of flexible integration of I/O devices at runtime and information-flow control. The presented solution allows to assemble user interfaces very quickly and to change the behavior of them at runtime. This enables researchers to experiment and identify appropriate interaction techniques, metaphors and idioms.

The presented component for interaction management has been prototypical implemented and tested within the project CAR. That has been conducted at the augmented reality research group of the Technische Universität München. The project CAR is part of a interdisciplinary research project that aims at the development of user interfaces in automobiles of the near future (five to ten years). Its main goal is to provide a collaboration platform for researches of different disciplines to discuss and develop new concepts for human computer interaction in automobile environments.

Zusammenfassung

Es ist eine der grössten Herrausforderungen in der Informatik, Benutzern einfache und adequate Möglichkeiten zur Interaktion mit immer komplexeren Computersystemen zur Verfügung zu stellen. In den letzten Jahren wurden viele neue Technologien für Benutzerschnittstellen entwickelt. Unter anderem *augmented*, *mixed* und *virtual reality*, *tangible*, *ubiquitous* und *wearable* Benutzerschnittstellen.
All diese Konzepte nähern sich immer mehr aneinander an um ein neues Paradigma zu formen. Dieses nennen wir *Ubiquitous Augmented Reality*.

Ubiquitous Augmented Reality Benutzerschnittstellen beinhalten eine Vielzahl an technischen Konzepten, darunter Multi-Modalität, Mehrbenutzersysteme und Systeme mit mehreren Geräten. Darüber hinaus beinhalten sie oft neue Ein- und Ausgabe Geräte. Im Gegensatz zu klassischen zwei dimensionalen Benutzeroberflächen hat noch keine Standardisierung von Ein- und Ausgabe Geräten, sowie von Interaktions Techniken für ubiquitous augmented reality Benutzerschnittstellen stattgefunden.

Diese Arbeit präsentiert eine Methode für Interaktions Mangement für ubiquitous augmented reality Benutzerschnittstellen. Diese Methode besteht aus der flexiblen Integration von Ein- und Ausgabe Geräten zur Laufzeit und der Kontrolle von Informations Strömen. Die präsentierte Lösung ermöglicht es, Benutzerschnittstellen sehr schnell zu entwickeln und später ihr Verhalten zur Laufzeit anzupassen.Dieses erlaubt Forschern, Experimente mit Interaktions Techniken, Metaphern und Idiomen auszuführen, um die richtigen Konzepte zu identifizieren.

Die präsentierte Komponente für Interaktions Management wurde im Rahmen des Projektes CAR, welches von der Forschungsgruppe Augmented Reality an der Technischen Universität München durchgeführt wurde, prototypisch implementiert und getestet. Das Projekt CAR ist Teil eines interdisziplinären Forschungsprojektes, mit der Zielsetzung der Erforschung von Konzepten für Benutzerschnittstellen für die Automobile der nächsten Jahre (fünf bis zehn Jahre). Das Hauptziel des Projektes CAR ist es, eine Kollaborations Platform für Forscher unterschiedlicher Disziplinen zur Verfügung zu stellen, damit diese neue Konzepte für die Mensch-Maschine Kommunikation in Automobilen diskutieren und entwickeln können.

Preface

About this Work This thesis was written as *Diplomarbeit* (similar to *Master's Thesis*) at the *Technische Universität München*, augmented reality research group at the *Chair for Applied Software Engineering*. During the last six months (November 2003 through May 2004).

My work has been embedded in a group effort of eight students at the *Technische Universität München*. The project CAR was part of a interdisciplinary project aimed at the development of user interfaces for next generation's automobiles. Incorporating researchers from computer science, human factor engineering, mechanical engineering and psychology.
The goal of project CAR is the development of a collaboration platform for all those researchers. Within this environment traffic conditions can be simulated using virtual cities and toy cars. In addition the corresponding user interfaces inside the involved cars are displayed. These user interfaces then can be modified at runtime according to the ongoing discussion about certain aspects of the user interface. Therefore a very flexible user interface infrastructure has to be developed. Further tools to modify the running user interface are needed. The focus of my work is the interaction management for such user interfaces, including input handling and modification of the output devices.

Structure of this Document This thesis addresses various aspects of my work and therefore parts of it might be of varying interest for different audiences. Here I would like to give a short overview of the single parts of my thesis.

Augmented Reality Researchers and other Computer Scientists should read Chapters 1 and 2 to get an idea what issues are discussed in this thesis and how it is related with their own work. Chapter 3 and 4 describe the problems, the resulting requirements and proposed solutions from a high level point of view. Chapter 6 concludes with a discussion of what has been achieved and also gives information about the shortcomings and things that remain to be done.

Future Developers might read Chapters 3, 4 to understand the discussed concepts and issues. To get insights into the functionality and implementation Chapter 5 is the right choice. Finally Chapter 6 provides ideas for future work.

General Readers might not be familiar with Ubiquitous Augmented Reality and therefore should read Chapter 1 for a general description of the problem domain. Chapter 2 describes the project CAR in more detail and can provide an general idea what my work is about. The Chapters 3 and 4 define what interaction management and runtime prototyping are and what problems they carry implicitly. Those Chapters already are of technical nature but might be understandable for the interested reader.

The DWARF Team might already be familiar with Chapters 1 and 2. But Chapters 3 and 4 should be very interesting because there all concepts are explained and discussed. Chapter 5 gives insight into the implementation of the component and its application. Finally Chapter 6 could provide further ideas on research topics.

Acknowledgements This thesis would not have been possible without the help of several people.
I would like to thank the supervisors of this thesis. Therefore, I would like to thank Gudrun Klinker and Christian Sandor for advising this work, supporting me with great ideas, and making this work possible.
I would like to thank all people involved in the augmented reality group for their technical support, especially on the DWARF system and its components, namely Martin Wagner, Asa MacWilliams, Christian Sandor, and Martin Bauer.
Further I would like to thank all students involved in the CAR project for all their efforts to let CAR become a success. Namely Fabian Sturm, Vinko Novak, Nicolas Dörfler, Maximilian Schwinger, Korbinian Schwinger, Peter Hallama and Markus Geipel.
Also I would like to thank all my friends for all the encouragement but also for dispersal, and for reminding me from time to time that there is a life out there.
Special thanks go out to Alex Olwal who helped me with extended proof reading of this document. And last but not least I want to thank my whole family for all their support, patience, and encouragement.

Contents

1 Introduction **8**
1.1 User Interface Paradigms . . . 8
1.2 Ubiquitous Augmented Reality . . . 9

2 Thesis Context **11**
2.1 DWARF . . . 11
2.2 The DWARF User Interface Architecture . . . 12
2.2.1 Layering and Device Abstraction . . . 12
2.2.2 Lightweight and Stateless I/O Components . . . 14
2.2.3 Set of Reusable I/O Components . . . 14
2.3 Problem Statement . . . 15
2.4 Goals . . . 16
2.5 Setup . . . 17
2.6 Scenario . . . 19
2.7 Subsystems . . . 20
2.7.1 User Interface Controller . . . 21
2.7.2 Attentive User Interface . . . 22
2.7.3 View Management . . . 22
2.7.4 Continuous Integration . . . 23

3 Interaction Management **24**
3.1 Multiple Users . . . 25
3.2 Multiple Devices . . . 27
3.3 Requirements Analysis . . . 29
3.3.1 Actors . . . 29

3.3.2 Scenarios . . . 30
3.3.3 Use Cases . . . 33
3.3.4 Functional Requirements . . . 36
3.3.5 Non Functional Requirements . . . 37
3.3.6 Pseudo Requirements . . . 38
3.4 Related Work . . . 38
3.4.1 Multi-modal Integration . . . 38
3.4.2 Tangible User Interfaces . . . 39
3.4.3 Semantic Interpretation . . . 40
3.5 Formal models . . . 40
3.5.1 Finite Automata . . . 40
3.5.2 Petri Nets . . . 42
3.5.3 Petri Net Frameworks . . . 43
3.6 Proposed Solution . . . 44

4 Interactive Runtime Development Environment 45
4.1 Scenarios . . . 45
4.2 Use Cases . . . 46
4.3 Requirements . . . 49
4.3.1 Functional Requirements . . . 49
4.3.2 Non Functional Requirements . . . 50
4.4 Related Work . . . 50

5 Implementation 52
5.1 Petri Nets for Interaction Management . . . 52
5.2 The Petri Net Kernel . . . 56
5.3 Interactive Runtime Development . . . 58
5.3.1 Net Structure Modification . . . 60
5.3.2 Dynamic Code Modification . . . 61
5.3.3 Connectivity Management . . . 63
5.4 Implementation Details . . . 65
5.4.1 Communication and Event Processing . . . 69
5.4.2 Net Manipulation . . . 70
5.4.3 The Graphical User Interface . . . 71

6 Conclusion **72**
6.1 Lessons Learned 73
6.1.1 Social Lessons 73
6.1.2 Technical Lessons 74
6.2 Future Work 74
6.2.1 The Runtime Development User Interface 74
6.2.2 Programming by Example 76
6.2.3 Authoring Within Augmented Reality 76
6.2.4 System Feedback 77
6.2.5 Extensions for the DWARF UI Architecture 77

7 Abbrevations **79**

Bibliography **80**

List of Figures

2.1 Functional decomposition of DWARF specific user interface components . 13
2.2 The setup of the CAR collaboration platform. 17
2.3 a) The table showing the virtual city and the tracked car. b) The driver's cockpit with the front shield simulation on a movable display. 18
2.4 The CAR subsystem decomposition: components in blue have been newly developed or rewritten, components in orange have been modified, components in grey have been reused as is. 21

3.1 Single user system vs. ubiquitous computing environments. 25
3.2 Co-allocated working vs. Collaborative working. 26
3.3 Various input devices that are readily available to DWARF programmers. 27
3.4 Different output components for Augmented Reality applications. 27
3.5 Integration of pointing gestures and speech commands. 28
3.6 Controlling a 3D and a 2D view simultanously. 28
3.7 HMD calibration with a pointing device 31
3.8 From left to right: Herding sheep with a tangible leader sheep. Scooping sheep with a tracked iPaq. Coloring sheep by moving it through the display fixed color bars. 33
3.9 Attentive user interface prototype within project CAR and its AR visualization. 34

5.1 from left to right: Petri nets modeling Sequential Execution, Concurrency, and Synchronization 53
5.2 The taxonomy for DWARF input events. 54
5.3 On the left, a Petri net that is executed within the JFern simulator, the currently active transition is highlighted in red. On the right, the same Petri net and how it is connected with the I/O components. 56

5.4 Point-and-Speech interaction in SHEEP. The tangible pointing device is used to indicate the position of a new sheep, the user then utters "insert" and a new sheep gets created. On the bottom one can see the Petri net in different stages. 57
5.5 Sequence of images for Scoop-and-Drop interaction with a virtual sheep and an iPAQ. In the lower right corner, the corresponding Petri net is shown. 58
5.6 The DWARF UIC showing a very simple Petri net and the net structure modification tab. 59
5.7 Adding a new ability to the DWARF UIC and the new connection shown in red. 60
5.8 a) A simple Petri net modeling the insertion of a sheep with voice and gesture commands. b) A new place and transition are inserted to the net to control the sheep's draw-style. 60
5.9 On the left, a Petri net containing a sub net. On the right, the sub net in a separate pop up window executing two transitions separately. 61
5.10 On top-left the CID, on top right the AR visualization of user's gaze. On the bottom the UIC we used to modify the AUI at runtime. 64
5.11 Schematic DWARF user interface incorporating different connection possibilities. 66
5.12 The layering of the DWARF UIC. 67
5.13 The packages of the DWARF UIC component. 68

6.1 Mock-up for a new version of the DWARF UIC. 75

Chapter 1

Introduction

One of the major challenges of current computer science research is, to provide users with suitable means of interaction with increasingly powerful and complex computer systems, which are composed out of many inherently dependent processes. For example, control rooms of industrial plants, surgery preparation rooms, airplane cockpits, and consoles of modern cars are typically equipped with many different physical or electronic input and output devices.

Recent user interface concepts, such as *multimedia*, *multimodal*, *ubiquitous*, *tangible*, or *augmented reality-based* interfaces, each cover different approaches. I believe all of these approaches are necessary to tackle the problems arising with increasingly complex human-computer interaction.

To provide human-computer interaction beyond traditional WIMP-based user interfaces [51], various communications channels are being explored that correspond more naturally to the human visual, auditory and tactile senses.

1.1 User Interface Paradigms

Whenever a user is allowed to use different modalities while interacting with a computer, we speak of *multimodal* interaction. Keyboard input, mouse clicks, but also more advanced input modalities such as speech, gaze, and gestures or input coming from different types of sensors (data gloves, head-mounted devices, haptic devices, sensors attached to body parts, etc.) provide a wide range of input modalities that can appear sequentially or in parallel. The computer needs to understand input coming from different modalities and it needs to be able to integrate these modalities.

Multimedia systems use different communication channels to present the user with content such as sound, haptics, 2D and 3D graphics. Research on *multimedia* based user interfaces focuses on handling the vast amount of data that is required to gather raw input streams and to generate output streams in real-time.

Although multimedia and multimodal based systems have much in common, they cannot be described as one being a subset of the other. Many of today's Internet browsers and e-mail systems provide multimedia based functionality without being multimodal. Multi-modal systems focus more on the synergistic high-level interpretation of a few combined and parallel input tokens.

Ubiquitous [56], *ambient* [10] and *pervasive* [20] interfaces to computers have been proposed by Weiser and others, with the goal of rendering computing power to people in such a pervasive manner that the computer in itself becomes a secondary (virtually invisible) issue. Large-scale environments such as buildings are equipped with networked computers and multi-channel user interfaces, such that users are always surrounded by them. Research in this field focuses on developing proper system layouts for such large-scale computer networks. Which require high data bandwidths and system adaptiveness to changing user demands. Ad-hoc interoperability of services is needed [32] in order to build context-aware smart spaces. Into those, wearable smart appliances can be integrated to provide users with personalized and context-adapted information.

Augmented Reality (AR) focuses on presenting information in three dimensions with respect to the user's current position. Users can thus see, explore and manipulate virtual information as part of their real world environment. In his classical definition of AR, Azuma [2] states three requirements: real-time performance, user registration in three dimensions and a combined presentation of both virtual and real information. Later on this definition was refined by Milgram's taxonomy [38], leading towards the concept of *mixed reality*.

Tangible user interfaces (TUIs) focus on the observation that century-old, very well-designed tools exist, e.g. in craftsmanships, that have been fine-tuned for years towards very specific usage. Based on human spatial and motoric skills, each such tool is directly tailored towards fulfilling a specific task. The purpose of each tool is immediately obvious to a trained craftsman. It represents a unique combination of input and output behavior that is directly suited to the task it has been designed for. The TUI community strives towards providing similarly powerful tangible user interfaces for computers and their interaction with virtual information. Hiroshi Ishii's work with the MIT Tangible Media Group has produced a large number of creative tangible user interfaces, e.g. [27]. A formal model and definition of TUIs is provided in [55].

1.2 Ubiquitous Augmented Reality

In recent years, several user interface techniques have converged. I present steps towards the emerging concept of ubiquitous augmented reality (UAR) which incorporates aspects from all of the concepts introduced above. UAR systems are typically multi-device (multimodal input and multimedia output), multi-user and distributed.

Devices can be carried by the user, which requires support for mobile systems that can be connected to stationary systems dynamically.
Furthermore UAR systems provide 3D computer graphics that are embedded within the real world and thus augmenting it. Scene views are provided via several personal and ubiquitously available displays, accounting for different options in user mobility and privacy. Some views are common to all users (e.g., in the form of projections on a table or wall), others are restricted to subgroups of users (shown on portable display devices) or to a single user (shown on a head-mounted display). The displayed content depends on the current position and orientation of a display in the scene, representing its current viewpoint.

A major challenge for user interface research in UAR is that idioms and metaphors are not yet established nor known, in contrast to classical WIMP user interfaces [51]. To establish the WIMP paradigm with its idioms like *Drag and Drop* or *Point and Click*, a lot of research and usability evaluations have been done. It is clear that for UAR, such experiments and evaluations still have to be carried out.

For experimenting with new concepts, it is necessary to develop prototypes quickly. In traditional WIMP usability engineering, it is common practice to quickly create mockups, sometimes just a sketch on a piece of paper, as a basis for discussion. However for UAR this would not be applicable, as 3D sketches illustrating dynamic aspects are difficult to draw and understand. The wide variety of used input and output devices makes it even more difficult to design UAR user interfaces in traditional ways. To shorten the development time for illustrative prototypes a framework for UAR user interfaces that supports experimenting is necessary.

Chapter 2

Thesis Context

To get a deeper understanding of the requirements for the developed components, we have to take a look at the target environment (DWARF) of the project and at the context within which it is developed (project CAR).

2.1 DWARF

The project CAR is designed on top of the DWARF [3, 35, 14] framework which is being developed at the Technische Universität München since the year 2000. The design of the Framework is geared towards distributed, ubiquitous Augmented Reality computing. DWARF consists out of the distributed service manager which locates and connects several services dynamically (e.g. input components, 3D-viewer and tracking components). The connectivity structure of components can be changed arbitrarily at runtime.

The combination of distributed and ubiquitous computing concepts in DWARF allows developers to create new Augmented Reality systems within short time because they can reuse already existing components, and combine them in new ways or add new components to create new functionality. The components in DWARF are named *services*.

To model how *services* depend on each other, we use the concept of *needs* and *abilities*. *Abilities* are functionalities provided by a component (e.g. a tracking component could have an *ability*: deliver position of the user's head) and *needs* describe functionalities that a component is dependent on (e.g. a viewer component could have a *need*: position of the user's head). *Needs* and *abilities* are typed, connections are only set up for matching types (e.g. pose data in the above example). To refine this concept further, we introduced attributes for *abilities* and boolean predicates over these attributes for *needs* [34].

The following features in DWARF enable developers to build UAR applications:

Flexibility Because of the loose coupling of components, DWARF systems are highly flexible.

Responsiveness Several communication protocols are implemented for the communication between components. Some of them are especially well suited for real-time applications, e.g. shared memory and CORBA structured events.

Distributed The components that form a DWARF system can be a combination of local and remote devices. Distribution is completely transparent to the components.

Adaptability With the inherent options for ad-hoc connections and reconfiguration of components, DWARF systems are also inherently adaptive.

OS independent To allow deployment among a variety of devices, DWARF has been designed to be independent of a specific operating system. We have successfully implemented DWARF systems on Linux, Windows and Mac OS X platforms.

Programming language independent Similarly, DWARF supports three programming languages so far: JAVA, C++ and Python.

2.2 The DWARF User Interface Architecture

This Section has been taken and adapted from a paper that is currently under review for the UIST'04 conference [23]. Here, several architectural principles and components that make up the user interface architecture within DWARF [49] are explained. An overview of the user interface architecture can be seen in Figure 2.1. An important distinction for communication channels is the frequency with which messages are passed. *Discrete Events* are typically sent every few seconds, whereas *Continuous Streams*, such as tracking data, are flowing constantly.

2.2.1 Layering and Device Abstraction

The UI components are arranged in three layers, *Media Analysis*, *Interaction Management* and *Media Design* (Figure 2.1). Most data flows linearly from the *Media Analysis* layer, which contains input components, to the *Interaction Management* layer, where the tokens are interpreted semantically. From there the data flow continues to the *Media Design* layer, where the output components reside.

A standardized format for tokens has been developed. The tokens are sent from the input components to the *Interaction Management* layer.

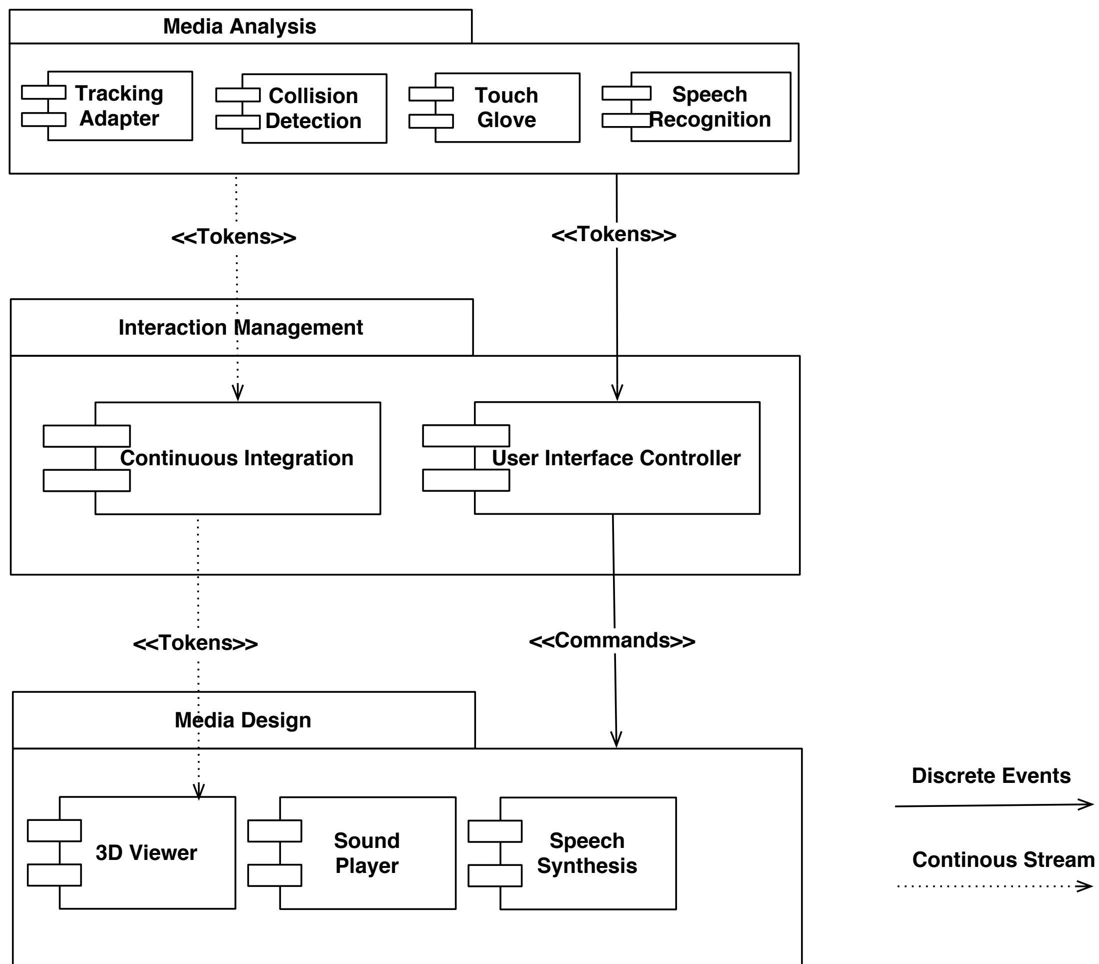


Figure 2.1: Functional decomposition of DWARF specific user interface components

Input tokens can be decomposed into four different types:
Analog values that can be either within a limited range (e.g., rotations) or an unlimited range (e.g translations) and discrete values that can be either booleans (e.g., pressing a button) or text strings (e.g., the output of a speech recognition component). Due to this standardized format, we can exchange one input device for another – as long as they emit the same type of tokens. A speech recognition component listening for a set of words could for instance be interchanged transparently for tangible buttons with the same set of labels.
Analog tokens with limited or unlimited range would typically be processed by the *Continuous Integration* package, whereas the discrete tokens usually go through the User Interface Controller (UIC).

Similarly, the *Interaction Management* layer sends commands to the *Media Design* layer. This setup corresponds to the *command pattern* described by Gamma et al. [19]. The commands consist of actions that have to be executed by the output components (e.g. by presenting the question "yes or no?" to the user). The exchange of I/O components works even at system runtime due to the flexible DWARF component model .

2.2.2 Lightweight and Stateless I/O Components

To address the flexibility requirement of user interfaces, as much state information as possible is kept in the *Interaction Management* layer. As a consequence, the I/O components were designed to keep as little state as possible. This allows us to add and remove I/O components conveniently at system runtime.

2.2.3 Set of Reusable I/O Components

The available set of I/O components is continuously extended (e.g. during student projects). Reusing these components does not demand any programming, because they are generic and meant to be reused among different applications. To tailor components to a specific application, the components are configured via an XML file. Furthermore, it is quite simple to write adapters for totally new devices since a variety of templates and helper functions are available to programmers.

Here is a short list of the most important I/O components:

Speech Recognition A Speech Recognition component that is configurable via a context-free grammar. The component is capable of recognizing predefined commands uttered by the user.

Touch-Glove The Touch-Glove is a special-purpose input device [6] developed at Columbia University. Input tokens that are emitted by this device can be fed into

a DWARF user interface. This device can emit both continuous and discrete input data.

Collision Detection Pose Data emitted by the tracking components is used by the *Collision Detection* component to detect collisions between objects. This includes collisions of real objects with virtual objects, real objects with real objects and virtual objects with virtual objects.

Sound Player The Sound Player component is a simple Java application configured via an XML file. It contains the mapping between incoming commands and sound files that are to be played.

3D Viewer The 3D Viewer component [22] displays specific views of the virtual world. An important design goal is the ability to update the virtual parts of a 3D scene in real-time. The current version accepts all important commands that are necessary for the display of dynamic 3D scenes in realtime. Additionally, several viewing modes are supported: video-background for video see-through displays or visualization of AR scenes or support for a variety of stereo modes for different stereoscopic displays. Furthermore, the display of head-fixed content (see [18]) is possible. Currently we are integrating the View Manager [4] from Columbia University into the 3D Viewer to support the automatic layout of presentation elements.

2.3 Problem Statement

The project CAR is part of an interdisciplinary project that aims at the development of concepts for next generation's automobiles. Today's cars already incorporate a wide variety of controls and displays (electronic and physical). The need to display an increasing number of functionalities on a limited number of displays has forced the automotive industry to find new paradigms for human computer interaction in automobiles. Since driving a car can lead to very dangerous situations if the driver gets distracted from his original task, secondary tasks including the usage of the cars auxiliary user interfaces (everything besides steering wheel, pedals and gear leaver) may not consume a lot of attention. Good usability is therefore very important for auxiliary user interfaces in automobiles.

To improve the comfort in the vehicle, more and more electronic entertainment and assistance systems are introduced by the industry. Those systems actively assist the driver in the performance of different tasks (i.e. park assistance, stop-and-go assistance, brake assistance etc.). While they support the driver in many traffic situations and therefore help the driver to cope with todays heavy traffic, the complexity of the auxiliary user interfaces incorporates two big problems. First, the learning threshold is

very high and users need to study complicated manuals before they can use them properly. Second, the complexity leads to usability problems which again draw away users' attention from the main driving task and hence are a safety issue.

New user interfaces techniques are necessary to overcome those problems. The project CAR has been founded to supply a collaboration platform for researches of different disciplines (e.g. computer science, human factor engineering, psychology and mechanical engineering).
We envisioned a car of the future with head-up displays (HUDs) placed everywhere. The information is presented on the most reasonable places based on the context, the current task of the driver and the driver's gaze direction. An interdisciplinary team was gathered to explore different aspects of such system and communicate their knowledge among each other to build prototypical user interfaces as a result of their collaboration.

2.4 Goals

This section describes the goals of the project CAR in more detail. This is done by splitting the goals into two blocks. First, the high-level goals which are the desired results for the end user (a driver).

Usability The auxiliary user interfaces must be as easy as possible to use to reduce the amount of time and attention spent to execute the desired task.

Intuitive usage The user interface design has to be intuitive to lower the learning threshold for a driver, so that the driver always knows what a control or display is good for, without having to read a lot of introductory material.

Safety The usage of the auxiliary systems may not decrease the safety while operating the car.

Aesthetics The developed system must be good looking and attractive because the visual impression is a very important factor in car sales.

Second, the goals that our collaboration platform has been tailored towards are those making the development of systems with the criteria described above possible.

Rapid prototyping While exploring new concepts in user interface design short development cycles are necessary to experiment with and discuss new ideas quickly.

Runtime development Discussions during prototype evaluation often lead to new solutions of certain problems which should be integrated immediately.

Authoring tools To enable rapid prototyping and runtime development, tools are necessary to modify and control different aspects of the running system. Among them data-flow, content and behavior of the user interface.

Sytem comprehension Due to the complex nature of the prototypes special purpose tools are necessary for easy evaluation and monitoring, visualization and debugging of the system by developers.

2.5 Setup

This section describes the setup of the developed collaboration platform (Figure 2.2) within the augmented reality research group [14]. The center of the collaboration platform is a table that serves as a display. We use a ceiling mounted video projector to display a virtual city on that table. The room is further equipped with an infrared tracking system that delivers position and orientation of special markers in real-time. We utilize toy cars equipped with such markers to simulate different traffic situations.

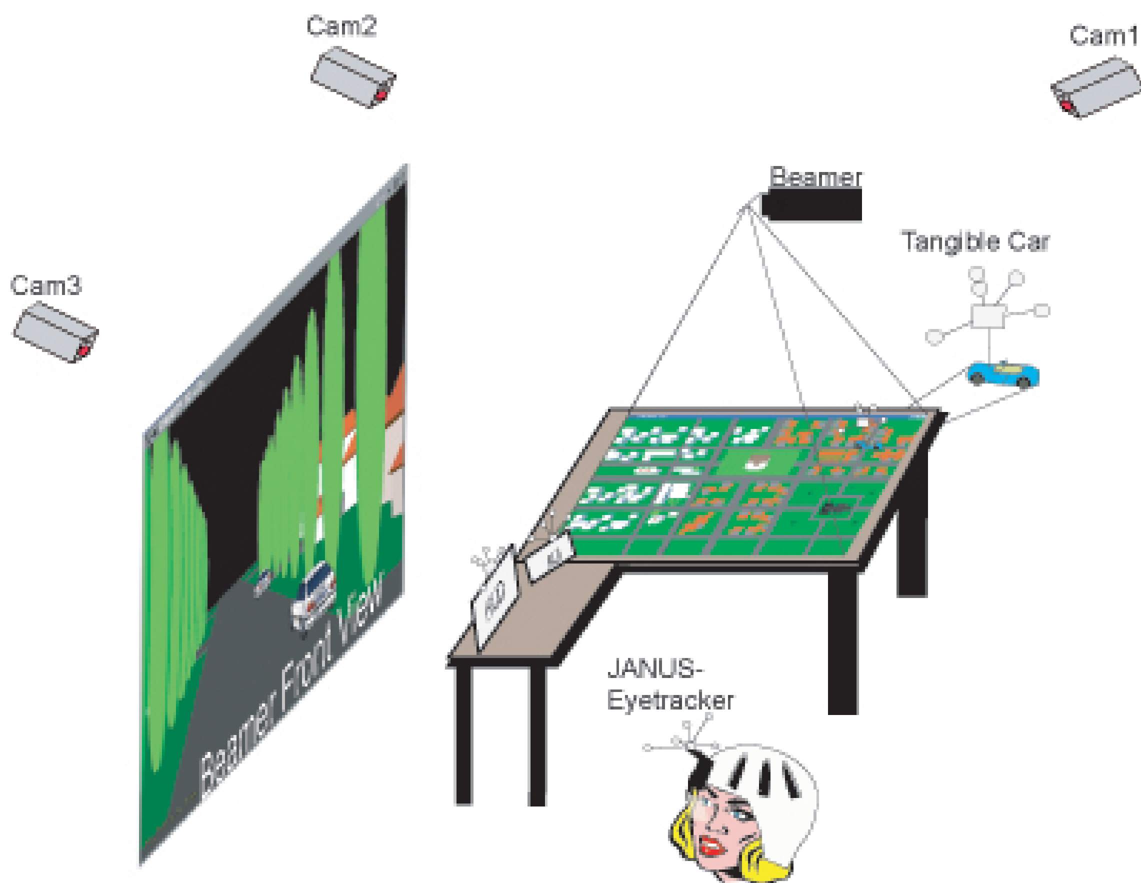
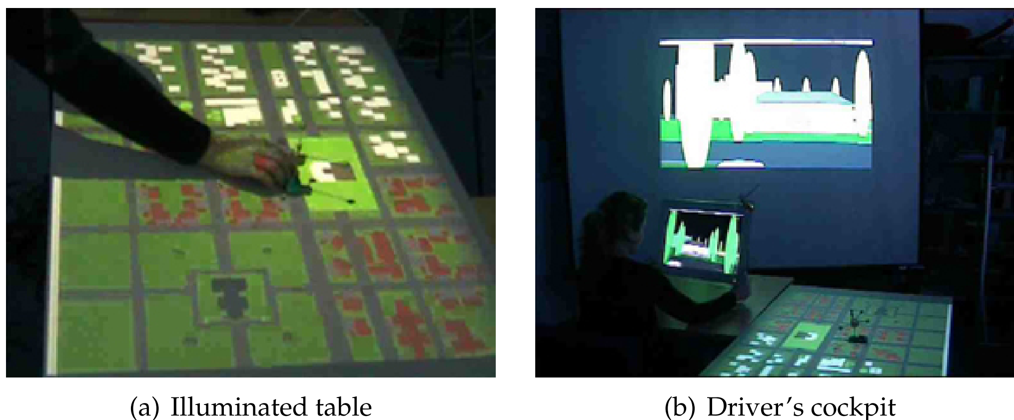


Figure 2.2: The setup of the CAR collaboration platform.

In addition to the central table, we installed a second smaller table that is a replacement for the car's cockpit. On that table, in front of the user, a laptop is installed, which

can be freely moved around the driver (See Figure 2.3). With the laptop we simulate a car equipped with head up displays (HUD) in all windows. Therefore the virtual city is displayed on the laptop as if the driver would be seated inside the toy car. As long as the laptop has not been moved the image on the screen is identical to the view through the toy car's front shield. The shown section of the virtual city reflects the driver's view direction whenever the laptop is moved. This means that if the driver rotates the laptop by 90 degree to the left, the shown scene is equivalent to the view through the toy car's left side window.

Besides the driver's view on the virtual city we can display graphical augmentations



(a) Illuminated table (b) Driver's cockpit

Figure 2.3: a) The table showing the virtual city and the tracked car. b) The driver's cockpit with the front shield simulation on a movable display.

on that screen such as navigation information, speed or other relevant information, dependent on the current traffic situation.

A second laptop is placed on the table as a replacement for a real car's central information display (CID). This is the display where secondary information such as controls for the air conditioning system, telephone and entertainment system are shown.

The driver wears a special helmet equipped with infrared markers and an eye-tracking system to acquire head and eye-gaze. The head and eye-gaze are visualized on secondary displays for evaluation purposes. It is of special interest to know how the driver's visual attention is influenced by the user interface elements, since the driving task requires as much attention as possible, for obvious safety reasons.

The room also provides a wall-sized display where the view from within the toy car is displayed. That display has two benefits. First, the driver's immersive impression is increased because the virtual city is displayed almost in original scale. Second, a larger audience can see what is going on in the driver's HUD.

2.6 Scenario

The selected scenario can be split into two parts. The first part is the simulation of the car environment and its behavior while the second part is the interaction of developers with the collaboration platform which is used to influence the simulation.

Scenario: **Car environment simulation**

Actor instances: `A family:Users`

Flow of Events:

1. A family starts a tour through a city, beginning at a parking lot. They start to drive along some roads of the city and after a while the co-driver decides to activate a tourist guide. This application is used to give more information about the scenery they see through the windows of the car.
2. When the tourist-guide is activated, it starts to add annotations to some of the buildings in the city by displaying small labels with names in one of the side windows' HUDs. The labels contain the name of the building they are pointing to and some historic information. The tourist guide uses automatic layout to present the information in a pleasant way.
3. The family now drives around the city to reach their final destination when suddenly a phone call arrives. Instead of just ringing, attention management is used to inform the driver about this new condition without disturbing him in a way which could lead to an accident. It is done by animating a symbol on the CID. At the same time the driver's eye-gaze is measured and only if that indicates that the driver has turned his attention to the CID by looking at it further information about the phone call is displayed.

   The driver now accepts the phone call by pressing a button on the CID or placing a speech command. After the phone call is finished, the driver pulls into a street with a parking lot and activates the parking assistant.
4. The parking assistant adds a small map of the city to the lower left corner of the windshield's HUD. This map shows a top view of the surrounding buildings and cars. Additionally, it draws a small symbol in the map to indicate the position of the car relative to the environment. The map is updated whenever the car moves and the direction of the map is always aligned with the driving direction of the car. When the car now gets closer to the parking spot the size of the map starts to increase and the map moves slowly from the lower left corner of the windshield to the center. When the map has reached a certain size it no longer grows but starts to zoom in and

shows greater detail of the location. In the case where the top view of the map is not helpful for parking, the driver can activate the use of a small plate as a tangible interface to change the viewpoint on the map.

5. Finally when the driver found a good enough view on the map he disables the tangible interface and proceeds parking and the journey ends.

Scenario: **Collaboration platform**

Actor instances: `Computer Science Engineers, Human Factor Engineers:Developers`

Flow of Events:

1. The above-mentioned simulation is achieved by interacting with the collaboration platform. The drive through the city is started by moving the small car on the table along the streets. Then either manually by clicking a button or through speech input, the tourist guide is started. During the time the tourist guide application is running the small car is moved to select nice viewpoints onto the different buildings in the city. The car is moved closer to a predefined parking lot after a while.
2. The parking assistant is started and now reacts to the movements of the car on the table. If the car is moved closer to the lot, a more detailed and bigger map is shown and if the car is moved away from the lot the map's size decreases again. All the parameters of the map, for example how fast it grows and at which distance to the lot it starts to zoom in, can be modified at runtime with a special configuration tool. Experimenting with these various parameters can be used to find the best settings for the different parameters, and to discuss different types of behavior with e.g. the human factor engineers, for instance.
3. Predefined actions can be started from the UIC application during the whole simulation, or new actions can be constructed at runtime from within the UIC. These actions can then be used to e.g. simulate the event of an incoming phone call or to load the parking assistant.

2.7 Subsystems

Figure 2.4 shows the functional decomposition of the CAR system. The blue components are the components that have been developed by the students participating in

project CAR and are described in more detail in this section.

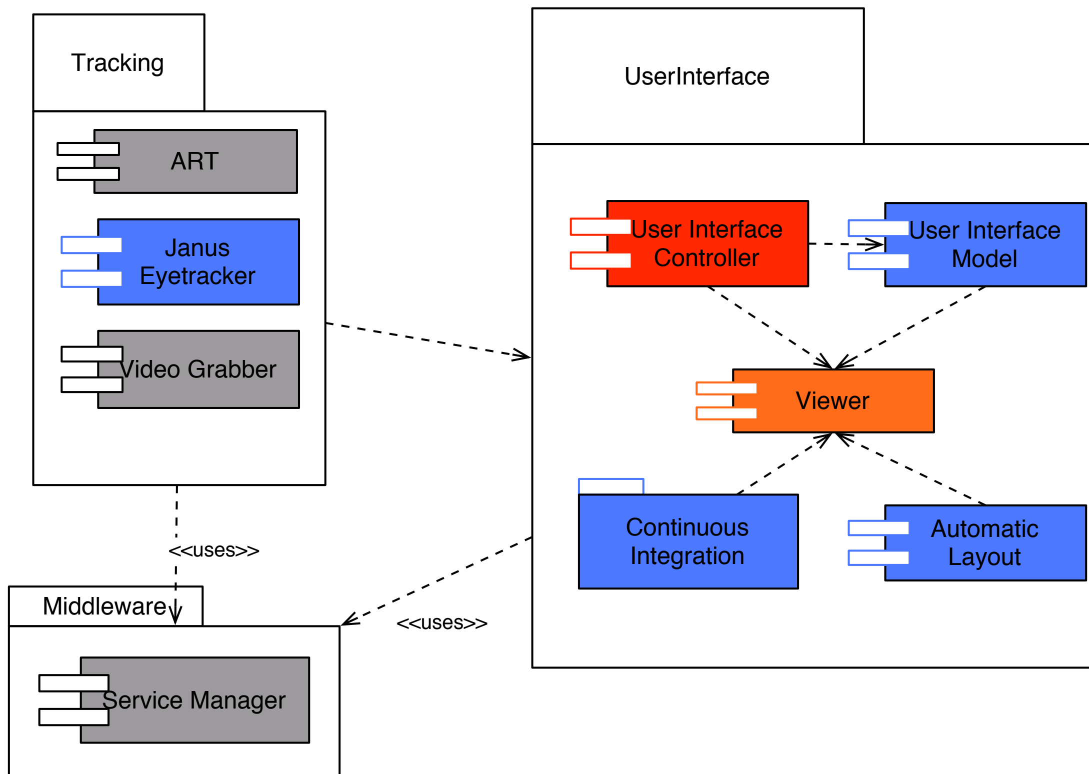


Figure 2.4: The CAR subsystem decomposition: components in blue have been newly developed or rewritten, components in orange have been modified, components in grey have been reused as is.

2.7.1 User Interface Controller

Interaction techniques for UAR user interfaces differ considerably from well-explored WIMP techniques, because these include new input and output devices and new interaction metaphors, such as tangible interaction. Interaction Management within DWARF is done by the UIC. It controls input and output components and handles dialogue control. The UIC is the core component of my thesis and is explained in more detail in Chapters 3, 4 and 5.

2.7.2 Attentive User Interface

The way users interact with user interfaces in cars is different from desktop environments, because the main amount of the visual attention has to be spent on the driving task and only bursts of attention are gained by the auxiliary displays. Besides, that input devices like mice and keyboards are not available nor suitable in cars.
To address these difficulties we utilize *Attentive User Interfaces* (AUI)[47] by using the visual attention as an additional input. First of all, we know from researches in the field of cognitive psychology and linguistics that the visual attention is used to control the flow of inter-personal group communication. Hence we use the attention as means to control the flow of communication between the driver and the car's user interface. An AUI is likely to respond to the driver's attention rather than to her explicit inputs. Furthermore, the AUI observes what the driver is doing and can adapt its behavior accordingly. AUIs use similar cues to signal their willingness to communicate to the driver, as people do in inter-personal group communications. For example to provide the driver with essential information, the presentation elements in the AUI perform an action (e.g. horizontal movements) in the peripheral field of view which attracts the user's attention. ! This behavior corresponds to a speaker who tries to attract the attention of the listener by waving to the listener or trying to get eye contact with the listener. After the AUI has gained the visual attention it increases the level of detail for the current context (it displays the corresponding dialogue), thus facilitates human-computer interaction by omitting the hierarchies of explicit dialogues that one usually has to click through to get to a certain dialogue. Accordingly, the speaker who has gained the attention of the listener starts to communicate without explicit request.
In summary, the goal of AUIs is to achieve communication between the user and the computer that is similar to inter-personal group communication.

2.7.3 View Management

The tourist guide and all the user interface widgets use view management [4] for their own rendering. This technique allows, in the case of the tourist guide, that street label positions do not have to be pre-specified by an author. This would be impossible anyway since the labels' positions depend on the viewpoint of the user and her position within the city. With fixed positions it would inevitably lead to overlapping labels. The same problem arises with the layout of the user interface widgets. The widgets that are simulaneously visible are not known a priori, but they all have to share the same limited space in the HUD and none should overlap the others. Automatic layout can also distribute the free space of the HUD to the different widgets and prevent overlap. At the same time it tries to provide as smooth as possible transition of widgets from one display to another. This provides an undisturbed experience of the real world with additional augmented information to the user.

2.7.4 Continuous Integration

The behavior of AR applications heavily depends on the use of incoming data. Position data from tracked objects is used to manipulate virtual objects. In some cases this data needs to be modified in a non-linear manner to enable advanced interaction techniques. For example, the pose data of the user's hand could be multiplied with a variable factor, to enable the user to reach for virtual objects further away than the user's actual arm length [44, 42]. Another example implemented within project CAR is the processing of the distance between the car and the parking lot. The system behavior is here non-linearly mapped to different actions, such as the amount the parking assistant scales the map or how much it is zoomed. An abstraction layer has been introduced between the data input and the data display layers which allows at-runtime configurable manipulations of incoming data streams. That abstraction layer has been designed as a pipe and filter architecture that processes incoming data streams and allows arbitrary modifications on the data values. A special tool has been implemented to modify and exchange the functions through which the data is passed.

Chapter 3

Interaction Management

For WIMP user interfaces, interaction management has been eased by the standardization of input and output devices. For UAR user interfaces, this standardization has not been achieved, as the wide spectrum of everyday tasks demands the usage of a wide variety of devices. Additionally, multi-user interfaces that are distributed over multiple flexibly interchangeable devices imply new challenges on interaction management such as input fusion and output fission.

In this section I want to give an overview of the problems and issues that have to be considered while designing UAR user interfaces. A special focus is put on the concept of *Interaction Management*. Interaction management means to adapt, configure, and control I/O components as well as defining how those components can be used in HCI. Interaction management incorporates the concepts of multimodal integration, dialogue control and multimedia presentation.

Interaction Management on the one hand, enables UAR systems to give users the ability to communicate, to interact, and to manage communications the way they want, with the devices they prefer the most. On the other hand it enables developers to define rules for how different interactions lead to changes in the system and the content presented to the user. Interaction management also allows UAR systems to adapt themselves to the current user and her preferences, as well as to the surrounding context (e.g users' current activity, users' inter-social engagement or the available devices). Hence the system can present content to the user in the preferred way and optimized to the currently available display devices.

The following issues differentiate UAR HCI from classic WIMP HCI and therefore have to be considered while designing an interaction management component:

Number of users WIMP systems are single user systems while UAR systems are inherently multi-user systems.

Number of devices UAR systems commonly incorporate a variety of devices. Among

them tracking, tangible and recognition devices that do not appear in classic 2D user interfaces.

Number of modalities While today's desktop is controlled via keyboard and mice, UAR systems provide a richer set of HCI possibilities, such as speech recognition or gesture input and of course combinations of different modalities.

Standardization The I/O devices (mice, keyboard) and the interaction techniques (e.g. drag and drop or point and click) are well-known and standardized for 2D user interfaces. This has not been achieved for UAR user interfaces, where in most cases I/O devices and interaction techniques are custom tailored for every application. Because of the lack of standards, and hence reuse, the amount of time spent for developing such user interfaces consumes super-proportional time and man power.

The issues listed above will be discussed throughout the rest of this chapter. I will describe them further in Sections 3.1 and 3.2. I will identify the resulting scenarios, use cases and finally the requirements for an interaction management component for UAR systems in Section 3.3. In Section 3.6 I will outline the proposed solution.

3.1 Multiple Users

The first distinction between single-user and UAR systems is the number and relationship of users and computer systems. In the past there was always only one user communicating with one system (e.g a PC). In recent years the increasing success of electronic devices in all areas of private and working life have changed this relationship significantly.

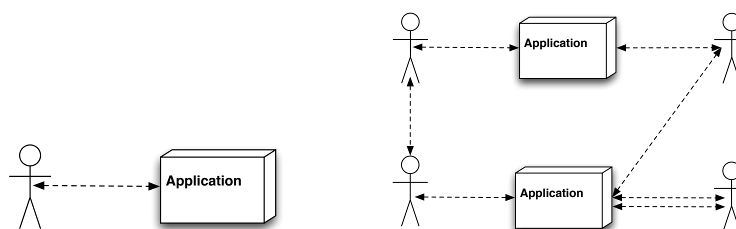


Figure 3.1: Single user system vs. ubiquitous computing environments.

Nowadays a user is most of the time surrounded by many electronic devices (e.g handheld computers, mobile phones, in-car electronics, and multi-media home entertainment). In addition, most of the activities in today's professional life have reached a

level of complexity where a single individual can impossibly solve all problems alone. In consequence, today's computer systems must support team work and collaboration. Figure 3.1 shows the change in the relationships between humans and computers over time.

In multi-user (and multi-system) environments we can distinguish between several types of collaboration. Figure 3.2 shows both extremes. On the one hand we have co-allocated working where several users utilize the same resources to carry out different, independent tasks (the top row of Figure 3.2). On the other hand we have a group of users combining effort to accomplish a shared mission (the bottom row of Figure 3.2).

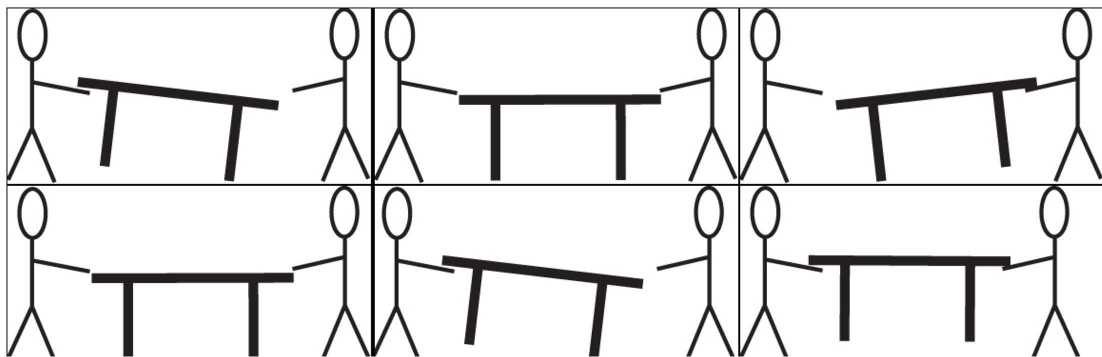


Figure 3.2: Co-allocated working vs. Collaborative working.

Between those extremes there is a continuum of collaboration. Co-allocated working and collaboration can be mixed. A user could, for example, read private e-mail (an independent task) while he is involved in a collaborative architectural design process, and two users, each of them working on an independent task, might share one display to present commonly used information needed for both tasks.
This continuum of collaboration raises new issues for interaction management. First, an interaction management system has to identify users and the devices they use. Secondly, such a component must have some concept of access restriction. Which user may use which devices, and what resources is she allowed to see and modify? And which devices can be shared amongst the users, which need to be used exclusivly? Third, must joint editing of one resource (e.g. in a 3D form finding application) be handled and managed.

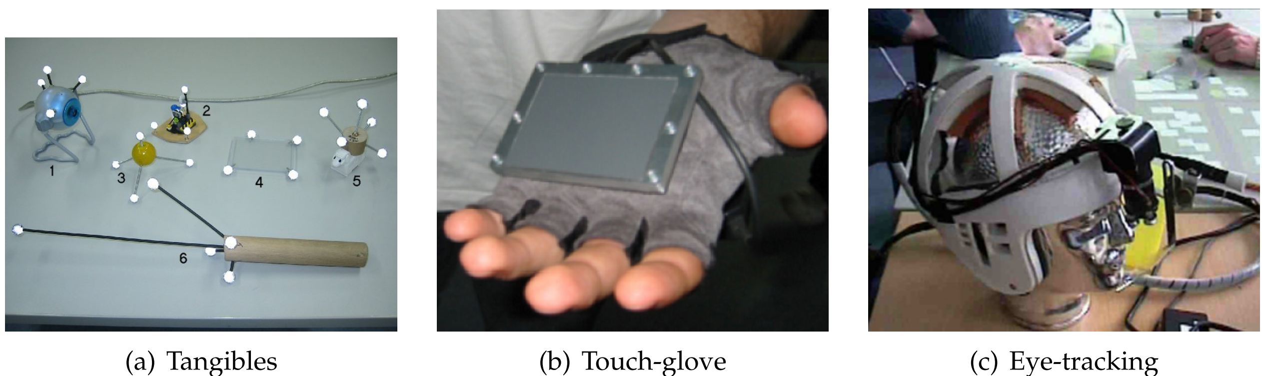
Another issue for interaction management in UAR systems is the distinction of public vs. private information [8]. It would, for example, be rather inconvenient if my private e-mail would be displayed on a wall-sized display, while other users are in the room. It is, on the other hand, desired that all users that participate in a collaborative, distributed (each user at a separate location) application get to see the same content.

Lastly, an interaction management component must differentiate between HCI and

inter-human communication (especially in systems that support speech recognition), and switches between human-computer and inter-human communication must be detected and handled.

3.2 Multiple Devices

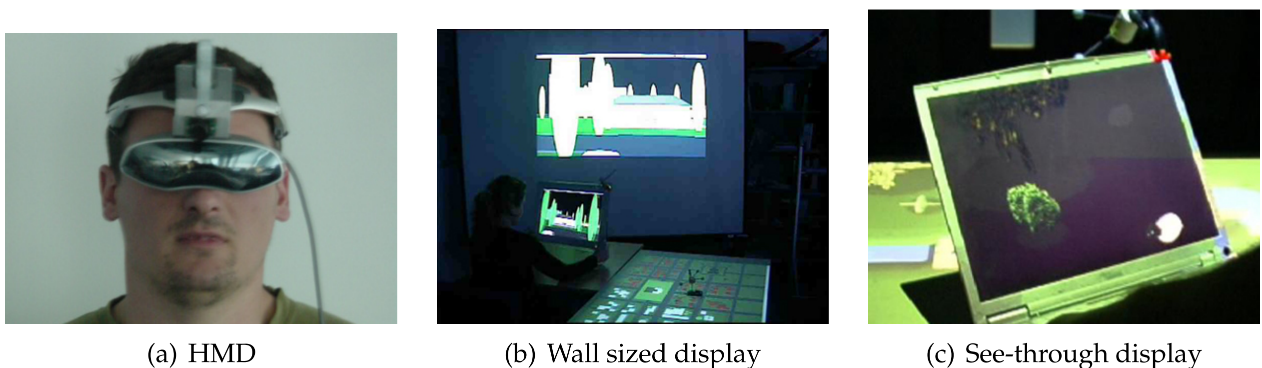
In order to interact with computer systems, a user (or a group of users) uses devices to express intention and to gather information. In classic setups these devices are commonly mice and keyboards for input, and graphical displays, printers, and speakers for output. In UAR environments there are far more devices involved, among them tangible, tracked, and recognition based input devices (Figure 3.4). On the output side



(a) Tangibles (b) Touch-glove (c) Eye-tracking

Figure 3.3: Various input devices that are readily available to DWARF programmers.

there are, for example, wall sized displays, head mounted displays (HMDs), haptic, and see-through displays. Figure 3.3 shows a selection of output devices.



(a) HMD (b) Wall sized display (c) See-through display

Figure 3.4: Different output components for Augmented Reality applications.

Some of the devices are only used for input, others for output and some for both. For example is a microphone used for input, speakers for output and a touchscreen can be used for both.
A big challenge for an interaction management component is to flexible adapt all those I/O devices. That includes the communication on the hardware level, as well as the interpretation of the generated input on the software level. Content for the output devices must also be generated and presented.

Since it requires much more management and coordination to utilize a bigger number of devices, a user must gain an advantage out of the increased number of input and output possibilities.

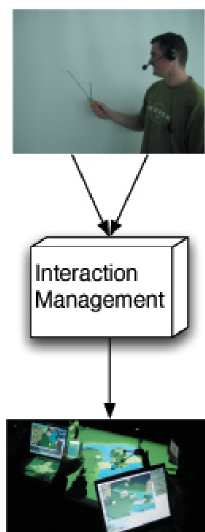


Figure 3.5: Integration of pointing gestures and speech commands.

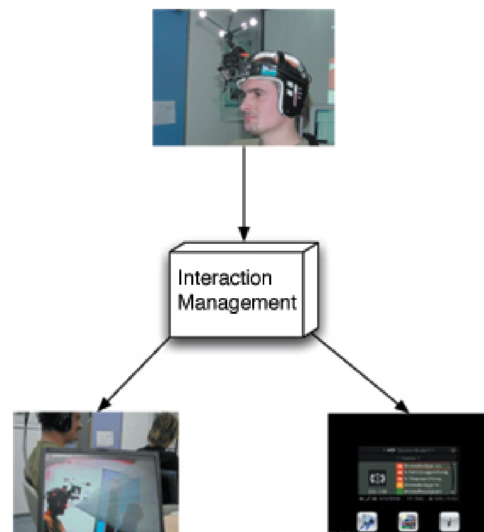


Figure 3.6: Controlling a 3D and a 2D view simultanously.

The new interaction techniques must simplify the usage of computer systems and reduce their error-proneness. To assure this, the different input modalities must be integrated in a manner that enriches the combined input with semantics [43]. For example, are both the pointing gesture at a location on a map, and the spoken question "What is the name of that town?" not expressive enough to deliver a meaningful information if they are interpreted as standalone input. In contradiction, if they are integrated and interpreted together, the gesture clarifies which town on the map is meant. Figure 3.5 shows an example for multi-modal integration.

The same considerations have to be done for the presentation of information to the user. Which modality is the best to display the current set of informations (2D or 3D graphics, animated or still pictures, with or without sound) for the current situation and the current user? To achieve this an interaction management component must be able to control the presentation components in a very fine granular manner. That gives

developers full control over the way they present information to their users. Figure 3.6 illustrates the simultaneous display of content in two different modalities.

Lastly, one has to consider privacy issues again. When a large group of users is competing for a small amount of devices, who has the right to use which device? Which devices must be shared (e.g. wall-sized displays, speakers)? And which can not be shared (e.g. personally trained speech recognition system)?

3.3 Requirements Analysis

Requirements elicitation focuses on describing the purpose of the system. The client, the developers, and the users identify a problem area and define a system that addresses the problem[7].

In this section I work out what requirements have to be fulfilled by an interaction management component for UAR user interfaces (the DWARF UIC).

3.3.1 Actors

An actor can be either human or an external system according to [7]. The different actors that are involved in the interaction management of UAR are introduced in this section. Since actors are role abstractions and do not necessarily map to persons [7], the same person or system may fulfill several roles at the same time.

UIdesigner The user interface designer creates the user interface for an UAR system. The designer specifies the devices incorporated into the user interface, specifies the flow of data through the user interface that is necessary to perform certain tasks, and specifies and creates the content (information) that is presented to the user of the system.

UIcontroller The user interface controller is part of the UAR system. It executes the user interface specified by the UIdesigner. Therefore it controls input and output components. It processes the input done by the user, and modifies the state of the system and the presentation accordingly.

User The user actually uses the system to perform special tasks, such as navigation, performing repair steps or playing a game.

UIStateChanger The user interface state changer is an entity from within the system that influences the state of the user interface. It is not a human being but it is providing input for the user interface.

3.3.2 Scenarios

A *Scenario* is a concrete, focused, informal description of a a single feature of the system from the viewpoint of a single actor [7]. Here I will describe the scenarios that have been used while designing the UIC component. They are not necessarily related to the project CAR 's scenarios described in 2.6 but rather describe typical problems for an interaction management component. Most of them have been realized with the current or an earlier version of the DWARF UIC. Since the UIC component has been designed to be highly generic, it is virtually impossible to describe all scenarios for its possible usage. The following scenarios have been chosen because, in my opinion, they help a lot in understanding the requirements for the developed component.

Scenario: **Define UI Devices**

Actor instances: `Alice:UIdesigner, UIC:UIcontroller`

Flow of Events:

1. Alice enters her laboratory which contains a running DWARF infrastructure and several input components among them a speech recognition component, a gesture detection component and various tangible, special-purpose devices. Additionally there are several output components available, such as 3D viewers in different sizes ranging from a wearable LCD display to a wall-sized display. Furthermore, she has a description of the task that has to be completed with the system she is about to design.
2. Alice decides which the appropriate input devices to perform the needed tasks are. She may consider that several input devices are of equal expressiveness and hence allow the user to choose between different options, or to use a combination of different input modalities to perform the user's tasks.
3. Now she decides how to present information to the user. Again, she might use several options in parallel or combine them to enrich the presentation.
4. Finally she sets up the needed DWARF communication structure so that the chosen I/O components can communicate with the UIC.

Scenario: **Define Workflow**

Actor instances: `Alice:UIdesigner, UIC:UIcontroller`

Flow of Events:

1. Alice needs to specify the data-flow through the user interface. This means that she defines rules that control which input triggers a change in the state of the user interface and the underlying system.
2. Finally, she configures the UIC, such that it is capable of executing the defined workflow.

Scenario: **Execute Workflow**

Actor instances: `UIC:UIcontroller`

Flow of Events:

1. The UIdesigner loads a user interface description into the UIC, specifying the connectivity structure between the UIC and the I/O components, and the data-flow through the UIC and as such the behavior of the complete user interface.
2. The UIC now receives input tokens from the input components, analyzes them and, if the input is complete, composes corresponding commands that are sent to attached output components and the application. The state of the application and the presentation is then updated.

Scenario: **Calibrate the Devices**

Actor instances: `Bridget:User, UIC:UIcontroller`

Flow of Events:

1. Bridget comes into the laboratory. She is equipped with a set of wearable devices such as a speech recognition component and a laptop that has a head-mounted display (HMD) attached. The laboratory is also equipped with an infrared tracking system. Since the speech recognition component was defined by the UIdesigner as an input device and the laptop's HMD as an output device, they both connect dynamically to the running DWARF system.
2. The UIC sends out an command to the 3D viewer component running on the laptop, that leads to the display of an initial 3D scene on the HMD.

Figure 3.7: HMD calibration with a pointing device

3. Bridget can now see the current virtual 3D scene but it is not calibrated (aligned correctly with the real world). She starts the calibration by uttering a speech command. Now a special superimposed calibration scene appears superimposed in her HMD as well and she is asked to align the peak of a tangible 3D pointing device with the corresponding calibration point in the 2D overlay. When she has aligned the pointing device, she utters another speech command to confirm the measurement (See Figure 3.7).
4. As the calibration method needs at least six measurement points to calculate the desired projection parameters, Bridget will be asked to repeat the last step several times.
5. After confirming the last calibration measurement the newly calculated calibration parameters will be transmitted to the viewing component.
6. Her HMD is now calibrated and can augment the surrounding (real) world.

Scenario: **Play a Game**

Actor instances: `Bridget:User`

Flow of Events:

1. Bridget (again equipped with speech recognition and a HMD) enters a room where a pastoral landscape is projected on a table (Figure 3.8). There is a plastic sheep on the table with special markers attached to it, as well as three tracked input devices: A magic wand used as the pointing device, a tracked iPAQ wearable computer, and a special marker that can be used to track a user's hand.
2. Bridget uses the magic wand to point at a position in the landscape. Then she utters a speech command ("insert") to create a new, virtual sheep at that position. She repeats this several times to create a herd of sheep. The virtual sheep begin to center around the plastic sheep, which can be used to guide the herd over the landscape.
3. Later Bridget decides to change the color of some of the virtual sheep. Therefore she takes the hand marker and places her hand over one of the sheep. The systems recognizes the collision. The UIC generates a command that removes the sheep from the projected scene and adds it to the 3D scene inside the HMD. Bridget now sees three colored bars, fixed to her viewpoint and the sheep standing on her hand. When she moves the sheep through one of the bars the sheep gets colored. When she places her hand back on the table the sheep starts to stroll around the landscape again.
4. Bridget can also use the iPAQ to pick up the sheep (and drop them

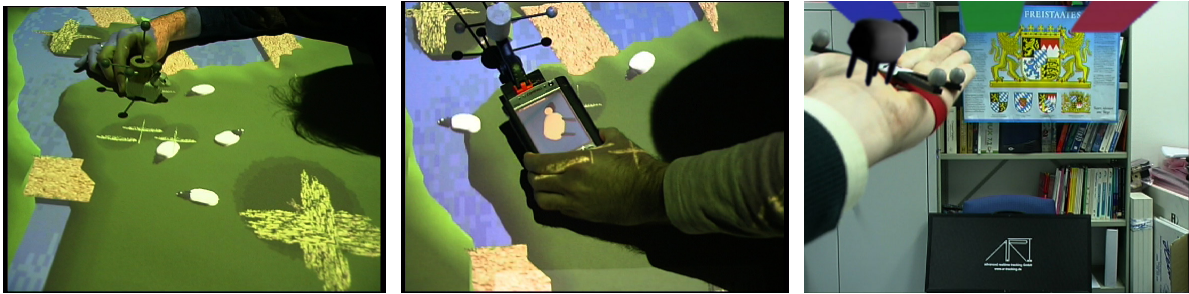


Figure 3.8: From left to right: Herding sheep with a tangible leader sheep. Scooping sheep with a tracked iPaq. Coloring sheep by moving it through the display fixed color bars.

later), for a detailed view of the sheep on the iPAQ display.

5. Whenever the landscape gets too crowded Bridget can point at a sheep and utter a speech command (”remove”) and the respective sheep gets removed from the game.

Scenario: **Attentive User Interface Management**

Actor instances: `UIC:UIcontroller, Bridget:User, UIModel:UIStateChanger`

Flow of Events:

1. Bridget drives around in her car.
2. Bridget receives a phone call.
3. The UIC sends out a command to the UIModel to perform an action to attract Bridget’s attention. The UIModel executes a horizontal movement of an object in Bridget’s peripheral field of view. When Bridget looks at the moving object the movement stops, and a message about the incoming phone call is displayed (Figure 3.9 shows the AR visualization of this scenario).
4. Bridget answers the phone and the display is switched back to its original state.

3.3.3 Use Cases

Every scenario is an instance of a use case, that is, a use case specifies all possible scenarios for a given piece of functionality [7]. Here, I present the use cases for the UIC, as extracted from the scenarios in 3.3.2.

Use Case: **Define User Interface Behavior**

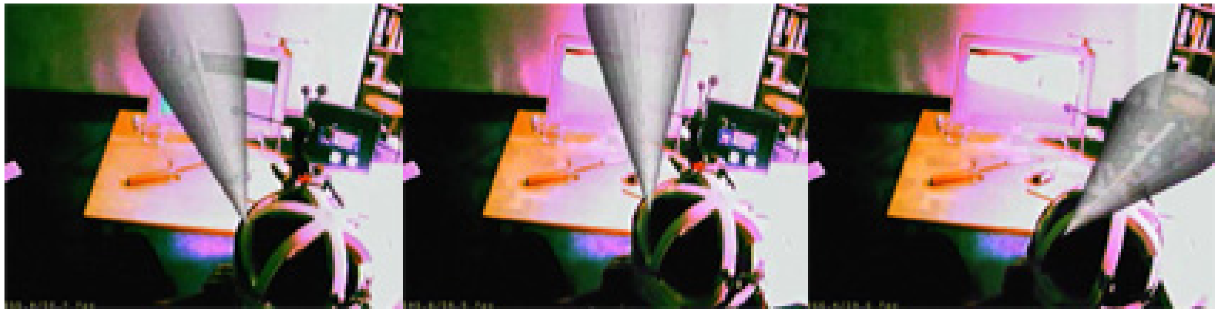


Figure 3.9: Attentive user interface prototype within project CAR and its AR visualization.

Initiated by: `UIdeveloper`
Communicates with:
Flow of Events:
1. *(Entry condition)* None.
2. The UIdeveloper defines the data-flow through the user interface, hence defines the user interface's behavior. Therefore she uses a formal description model.
3. *(Exit condition)* A user interface behavior description is available that can be executed by the UIC.

Use Case: **Start Interaction Management Component**
Initiated by: `UIdeveloper`
Communicates with:
Flow of Events:
1. *(Entry condition)* User interface behavior description is available.
2. The UIdeveloper starts the UIC and loads a user interface description (control structure) into the component. Furthermore she checks wether the DWARF `needs & abilities` are set up correctly.
3. *(Exit condition)* The UIC is up and running waiting for input and output components to get connected.

Use Case: **Connect Input Devices**
Initiated by: `System`
Communicates with:
Flow of Events:
1. *(Entry condition)* The UIC has been started

2. The UIC gets connected to an input component via DWARF. It now starts receiving input tokens from the connected component. These tokens are received and extracted such that the contained data can be processed.
3. *(Exit condition)* The UIC is able to receive input tokens.

Use Case: **Connect Output Devices**
Initiated by: `System`
Communicates with:
Flow of Events:

1. *(Entry condition)* The UIC has been started.
2. An output component (e.g. a 3D Viewer) gets connected to the UIC via DWARF. The UIC can now send commands to that component. The component displays the information contained in the command or changes its state accordingly to the command.
3. *(Exit condition)* The user interface can display content.

Use Case: **Process Input**
Initiated by: `User`
Communicates with: `UIcontroller`
Flow of Events:

1. *(Entry condition)* The UIC has been started and at least one input and output component are attached.
2. The user places an input command (e.g. a button press). The respective input component sends a token to the UIC. There, the data contained in that token is extracted and analyzed (semantically interpreted). Eventually a command is generated that can be sent to an output component where it triggers the display of content or a user interface state change.
3. *(Exit condition)* The user can interact with the system.

Use Case: **Integrate Different Modalities**
Initiated by: `User`
Communicates with: `UIcontroller`
Flow of Events:

1. *(Entry condition)* The UIC has been started and at least two input and one output component are attached.
2. The user places at least two input commands (e.g. gesture and a speech command). The respective input components send

tokens to the UIC. There, the data contained in the tokens is extracted. Both tokens' data content is needed to complete the current task. For example could the speech command dispose the insertion of a object and the gesture could contain the object's initial position. A command is generated and is sent to the output component.

3. *(Exit condition)* The user can interact with the system using multi-modal input.

Use Case: **Resolve Ambiguities**

Initiated by: `User`

Communicates with: `UIcontroller`

Flow of Events:

1. *(Entry condition)* The UIC has been started and at least two input and one output components are attached.
2. The user places at least two input commands (e.g. a speech command and a button press). The respective input components send tokens to the UIC. There, the data contained in the tokens is extracted. Since both tokens can contain data of equal semantic expressiveness, ambiguities can occur (The first denoting "yes" and the second "no" as an answer to a "yes/no" question). The UIC now has to resolve these ambiguities. That can be done by weighting the different modalities, such that one has precedence over the other or by using more advanced techniques from the field of artificial intelligence.
3. *(Exit condition)* The user can interact with the system using multi-modal input and the UIC can resolve ambiguities.

3.3.4 Functional Requirements

Functional requirements describe the interactions between the system and its environment independently of its implementation [7].

Adapt I/O components The UIC is designed to glue together input and output components. Therefore it must be able to set up connections with, communicate with, and control input components.

Identify users To handle input from multiple co-allocated or collaboratively working users, the component must provide means to identify which input has been done by whom.

Access control In collaborative applications the interaction management component must take care which resource may be modified by which entity (e.g. users).

Process data What remains of HCI on the system (control) level, technically spoken, boils down to the flow of data and its manipulation. To handle user input and translate it into commands for the *Media Design* layer the UIC must provide means to analyze, modify and transfer data.

Formal behavior description To describe the behavior of the user interface in an unambiguous and well-defined way a formal model has to be provided. That model shall be used to describe how data flows through the user interface and as such, the user interface's behavior. Additionally, that model needs a formal notion (possibly in textual as well as in graphical form) and needs to be executed in realtime.

Load and store behavior descriptions Since the UIC is the only component within the DWARF user interface architecture that knows about and holds state information, it is necessary to load and store the behavior description in a persistent manner (e.g. a document in the filesystem).

Multi-modal integration As stated in Chapter 1 UAR systems are inherently multimodal. The UIC has to handle different types and modalities of user input. It has to integrate it, i.e., to extract data and semantics from the single modalities and combine them in a way that produces well-defined commands for the *Media Design* layer components.

Disambiguate input The UIC must differentiate between human-computer communication and human-human communication. Further must the component detect whether input is combined (different modalities used to express one intent) or exclusive (to users performing different tasks).

3.3.5 Non Functional Requirements

Nonfunctional requirements describe the user-visible aspects of the system that are not directly related to its functional behavior [7].

Availability The UIC is an important part of the user interface and the whole system becomes unusable if it fails.

Reliability The component must perform its tasks in a correct and deterministic way, hence the user can be assured that the same action will lead to the same result every time it is repeated.

Robustness Since interacting with new systems always includes learning, users will give wrong or improper input because they are trying to find out how the system works. The component must not fail because of misusage.

Fault tolerance The component must be able to handle connection faults or recurring disconnections from attached components.

Responsiveness Since usability of a system is very dependent on feedback (did an operation succeed or not), the component must stay responsive in all circumstances and must process input by the user without delay.

3.3.6 Pseudo Requirements

Pseudo requirements are requirements imposed by the client that restrict the implementation of the system [7].

DWARF The component must be embedded within its DWARF environment. It must be able to communicate with other DWARF services especially with I/O components.

Programming language The component must be implemented in one of the programming languages supported by DWARF : Java, C++, or Python. This requirement should not limit the functionality of the implementation.

3.4 Related Work

In this section I give a brief overview over related work that has been done by others in this research field. I will also compare that work with what I have done and work out the advantages and disadvantages of the introduced projects. I will put a special focus on the comparison between finite automata and Petri nets as control structures that can be used to describe the data-flow through the user interface. Finally, I will explain which approach has been chosen and why.

3.4.1 Multi-modal Integration

Quickset The Quickset [31] system has introduced many pioneering ideas in the field of multi-modal interaction and multi-modal integration. Quickset is a multi-modal interface developed for speech and pen interactions in a military simulation application. It uses a semantic unification process to combine the meaningful multimodal information carried by two input signals, both of which are rich and multidimensional. Oviatt et al. [43] have shown that multi-modal interactions - when carefully designed - can not only lead to faster task completion but also result in more robust and less error-prone systems than unimodal approaches. They also

stated that research in the field of multi-modal systems is of interdisciplinary nature. We propose a system that adapts different modalities very flexibly and that is not limited to speech and pen input. This would allows us to quickly experiment with new paradigms and give us the possibil! ity to integrate feedback given at runtime by users and researchers of other disciplines, such as cognitive-science or human factor engineering.

OpenTracker and Unit OpenTracker [45] and Unit [42] are both frameworks for continuous integration in virtual and augmented reality systems. Both use a pipe and filter architecture to modify continuously changing data. They are mostly targeted towards tracking and pointing devices (mice etc.) data. Both frameworks propose a similar abstraction to ours. Still, our approach differs in two aspects. First, we strictly differentiate between discrete events and continuous data streams. User input that causes state changes of the user interface is almost always discrete. In contrast, continuous data modifies properties of objects in user interfaces (e.g. position, size, color). Second, we have a stronger focus on modularity, distribution and dynamic behavior.

VR UIMS In [29] a user interface management system, based on a user interface description language in SGML and extensible components in C++, is proposed. They also make the distinction between continuous data streams and discrete events. They provide a graphical editor for the specification of the changes in the Media Design layer according to those events. There are however a number of differences because they focus exclusively on Virtual Reality where input hardware is, to a certain degree, standardized and known in advance. Dynamic device exchange is not really an issue. In contrast we are concerned with semantical equivalence and flexible exchange of I/O devices and distribution of software components over several hosts. Additionally, we consider output fission and control of multiple output components to be an issue.

3.4.2 Tangible User Interfaces

TUI Ulmer et al. [55] try to physically instantiate elements of the user interface to let users interact naturally with digital worlds. The metaDESK [54] integrates multiple 2D and 3D graphic displays with an assortment of physical objects and instruments. Those objects are tracked by a variety of sensor technologies to enable them to communicate with virtual worlds. TUIs incorporate new paradigms for both multi-modal input and multi-media output. We do not focus on the invention or development of new input or output devices. But we try to provide a platform to adapt such tangible or physical devices easily.

Papier-Mâché Papier-Mâché [53] is a toolkit for building TUIs without having expertise in hardware design and computer vision. The proposed toolkit consists of a set of methods to adapt various sensor techniques (e.g. barcodes, computer vision) and to abstract input hardware. Further it provides a high-level event model and an API for programming event based applications incorporating tangible input. A graphical debugging environment has also been implemented. The authors have shown that the toolkit lowers the threshold for building TUIs significantly. Our approach contains a similar device abstraction and event model. We do not limit ourselves to TUIs but support all kinds of input devices that are able to emit a defined set of events. We support a formal model to describe the flow of events and a visual programming environment while Papier-Mâché is an API and developers have to write code. We do not however provide explicit ! recognition support as Papier-Mâché does. Instead we utilize third party recognition techniques.

3.4.3 Semantic Interpretation

Robotics Integrating different modalities isn't a trivial task since it includes a wide variety of sensors and data processing tools. Finally all collected data must be semantically interpreted. In the field of robotics, several approaches have been made to combine sensor data processing and semantic interpretation in one control structure. Among these approaches are Bayesian networks [52] and agent based approaches [17] combined with hidden Markov chains. Those systems are often focused on fulfilling one special task (e.g. robot navigation). In contrast, we try to separate sensor technology and sensor fusion from semantic interpretation to gain flexibility and generality.

Attentive User Interfaces Horovitz et al. describe in [15] a three-tier architecture for multi-modal systems similar to ours. They propose layering input components, sensor components, control structure, and output components. But they focus on attentive user interfaces while we focus on UAR user interfaces.

3.5 Formal models

3.5.1 Finite Automata

Definition In [25] a finite automaton FA is defined formally as follows:

A **Finite State Automaton** or **FA** is a five tupel $\{S, \Sigma, \delta, s_0, A\}$ where
S is a set of states
Σ is an alphabet
δ is a transition function $(\delta : S \times \Sigma \rightarrow S)$

s_0 is a special start state and
A is the set of accepted end states $(A \in S)$

A **Finite State Automaton** is called **Deterministic Finite State Automaton (DFA)** if $\forall s \in S$ and $\forall \sigma \in \Sigma : \delta(s, \sigma)$ is unequivocal.

For each non-deterministic FA a deterministic FA of equal computational power can be constructed with an algorithm (e.g. the Myhill construction [25]).

Description The DFA starts in the start state s_0 and reads in a string of symbols $\sigma \in \Sigma$. It uses the transition relation δ to determine the next state(s) using the current state and the symbol just read or the empty string. If, when it has finished reading, it is in an accepting state A, it is said to accept the string, otherwise it is said to reject the string. The set of strings it accepts form a language, which is the language the DFA recognizes.

Computational Power DFAs can only recognize regular languages, and hence they are less computationally powerful than *Turing machines* - there are decidable problems that are not computable using a DFA (e.g. DFAs can not count).

Notation FAs are commonly represented as state transition table or as a state diagram (directed graph with circles representing states and edges representing transitions)

During the design and implementation phase of the original DWARF version Sandor and Riss [48, 46] used DFAs to model both workflow and user interface behavior. The rationale behind that decision had three main arguments:
First, they stated that the learning threshold for the design and usage of DFAs is very low.
Second, they made the assumption that a user interface can only have one active state at a time and that a task completion is always done linear.
Third, they assumed that interaction would rather happen once at a time and not concurrently. DFAs match that understanding of user interfaces very well.

While I agree with the first argument I disagree with number two and three. Comparing HCI with a step by step workflow is an idea that has got its roots within the WIMP world of user interfaces, where HCI is mostly happening through dialogues. In dialogue based user interfaces, there is only one active state and there are only few possibilities to succeed from that state. This is no longer true for UAR systems because here a user might change the state of a widget or answer an question and at the same time modify a 3D object constantly (e.g. change position or size).

We know, from the field of psychology, that humans do prefer to only carry out one task at a time, over carrying out several tasks in parallel. In contradiction to Riss and Sandor, I think that this comprehension does not mean that we do not have to deal with concurrency in user interface design. Since UAR systems are multi-modal and multi-user systems, a lot of input can occur in parallel even if each user is only performing

one task at a time (e.g. because solving one task is done using several modalities). Modeling concurrency is very difficult with DFAs, since they always have only one active state at a time. Therefore we need a more powerful formal model to handle multi-modal input from multiple users in distributed systems.

3.5.2 Petri Nets

A **Petri net** [58] is a mathematical representation of discrete distributed systems. **Petri nets** were defined in the 1960s by Carl Adam Petri. Because of their ability to express concurrent events, they generalize automata theory.

Definition A **Petri Net** is a four tuple $\{P, T, IN, OUT\}$ where
$P = \{p1, p2, ..., pn\}$ is the set of all places
$T = \{t1, t2, ..., tn\}$ is the set of all transitions
$P \cup T \neq \emptyset, P \cap T = \emptyset$
$IN \subseteq (P \times T)$ is an input function that defines directed arcs from places to transitions, and
$OUT \subseteq (T \times P)$ is an output function that defines directed arcs from transitions to places.
Places of Petri nets usually represent states or resources in the system, while transitions model the activities of the system.

Description A Petri net consists of places, tokens, arcs, and transitions. The arcs connect places and transitions. Places and arcs may have capacities. A transition fires when all places at the end of incoming arcs contain enough tokens.

Computational Power Petri nets are *Turing complete*. A given programming language is said to be *Turing complete* if it can be shown that it is computationally equivalent to a Turing machine. That is, any problem that can be solved on a Turing machine using a finite amount of resources (i.e., time and tape), can be solved with the other language using a finite amount of its resources[1]. That means that Petri nets are equivalent in computational power to almost any programming language such as C++ or Java.

Notation Petri nets are commonly represented as directed acyclic graphs, whereby transitions are drawn as rectangles, and places as circles. The transitions and places are connected with arcs.

Petri nets, or place-transition nets, are classical models of concurrency, non-determinism, and control flow. Petri nets provide an elegant and mathematically rigorous modeling framework for dynamic systems. They also provide an easy to learn and

[1] http://c2.com/cgi/wiki?TuringComplete

understandable graphical notation.
Because of their expressiveness they are used in such diverse fields as hardware design, software engineering [2], telecommunication, business process modeling and workflow systems [16]. A variation - object-oriented Petri nets are used to model a variety of dynamic systems (e.g. autonomous multi-agent systems [39]).

The UIC uses Petri nets to model the behavior of UAR user interfaces. The designed user interfaces are deployed by directly executing the underlying Petri net model.

Petri nets are specifically useful to model the integration of different modalities and of course to handle occurring problems like concurrency and synchronization.

3.5.3 Petri Net Frameworks

Renew Renew [33] is a Java-based, high-level Petri net framework that provides a flexible modeling approach based on reference nets, which are the equivalent to classes in object-oriented programming. Thus Renew allows object-oriented modeling on a net level. It also provides a simulator to execute Petri nets and a graphical editor to design Petri nets. While the framework supports all standards and definitions of classic high-level Petri nets, it also provides a variety of custom extensions to the Petri net concept, such as timed nets and custom arcs (e.g. testing arcs). The vast amount of possibilities provided by Renew require a significant amount of learning, which adds to the already complex problems that have to be considered for UAR user interface design, and thus Renew is not feasible for our purposes.

CPN Tools CPN Tools[3] is a tool for editing, simulating and analysing Coloured Petri nets. It is geared towards large scale Petri net (thousands of net elements) design, simulation, and analysis. The most interesting aspect of CPN Tools is it's GUI. The user interface incorporates the latest state of the art concepts in HCI including direct manipulation, toolglasses, marking menus and palettes. But the highlight of CPN Tools is the bi-manual[4] input mechanism that renders pull-down menus, scrollbars, and even the concept of selection obsolete [37]. Thus Petri net designers can manipulate all GUI elements without having to select them in advance, which eases the management of very complex net structures significantly.
As with Renew, CPN Tools' scope is way to large for our purposes. In addition, it is not an Open Source framework. Although ! it can be used for free, developers have no access to its source code.

JFern JFern [41] is an object-oriented Petri net simulation framework. JFern's Petri net model is based on the traditional model of hierarchical Petri nets with time [58] and the additional concept of *object-based* tokens - places can contain arbitrary Java

[2]UML activity diagrams are a variant of Petri nets
[3]http://wiki.daimi.au.dk:80/cpntools/cpntools.wiki
[4]using two mice simultaneously

objects as tokens. It consists of a lightweight Petri net kernel, providing methods to store and execute Petri nets in realtime, and a simulator, including a simple GUI for runtime visualization. JFern also supports XML based persistent storage of Petri nets and their markings.
Besides the small footprint of the JFern engine, the delightful compact but yet powerful API provided by JFern was the crucial factor for choosing it as foundation for the DWARF UIC.

3.6 Proposed Solution

Interaction Management within DWARF is done by the UIC. It controls input and output components and handles dialogue control. The UIC component handles all discrete user input, such as button presses, speech input and gestures. Therefore it receives input tokens [57] and does a rule based token fusion. *MediaAnalysis* means to analyze the incoming tokens' values and interpret the user's intention. In the *CommandSynthesis* phase new command tokens are assembled, and sent to the output components to change the state of the user interface.

The DWARF UIC utilizes Petri nets to model user interface behavior and to integrate various modalities. It utilizes the DWARF middleware to adapt, configure and communicate with I/O devices.

The original idea to use Petri nets was brought up by my supervisior Christian Sandor during the design phase of SHEEP [36, 50]. The surrounding concepts and the first implementation have been developed in collaboration by Christian Sandor and me during the implementation phase of SHEEP. Throughout the projects ARCHIE [12] and CAR the concepts especially concerning device adaption, connectivity management, and behavior description have been extended and the implementation has been revised.

In addition, the UIC incorporates a runtime development environment for flexible adaptation of input and output devices, and to change the behavior of the whole user interface. That enables developers to build working user interface prototypes very quickly, and enables them to change, tune, and adapt those prototypes at runtime.

I did not implement a concept of user identification or access control, however. I think that such an approach would contradict the modular approach of the DWARF user interface architecture. I propose to integrate mechanisms to identify users and devices at a very low level. The device driver level would be appropriate in my opinion since very effective approaches, such as biometrics, could be applied on that level.

To guarantee robust and safe user identification and access control, the DWARF middleware has to be extended such that it is capable of allowing and denying access to services, or even single needs and abilities, on a network transfer level.

Chapter 4

Interactive Runtime Development Environment

To enable rapid prototyping of UAR user interfaces, a developer needs a supporting tool for designing the user interface behavior at runtime, as well as the connectivity structure between the user interface components.

I will show the typical tasks in rapid prototyping of UAR systems in Section 4.1, which will be further refined in Section 4.2 with the requirements for such a development environment specified in Section 4.3.

4.1 Scenarios

In this section I explain the desired functionality of a runtime development environment for UAR systems using example scenarios. Due to the generic tasks that are done within a programming environment, it is not possible to describe all possible applications in a dedicated scenario.

Scenario:	**Create New UI Description**
Actor instances:	`Alice:UIdesigner, UIC:UIcontroller`
Flow of Events:	1. Alice wants to design a new user interface prototype. Therefore she starts up the UIC displaying an empty Petri net on the main working bench.
	2. She now starts to define the data-flow by adding input places, transitions and finally output places. She connects the places and transitions with directed arcs.
	3. The data will now flow along these arcs, and transitions will fire whenever tokens pass them.

4. To define what happens when a transitions fires, Alice uses *actions* that are encapsulated within the transitions. In most cases, these actions use data from several incoming tokens and combine it into a new command for the output devices.
5. At the end Alice sets up several DWARF *needs & abilities* to connect the UIC with I/O components. *Needs* are mapped onto input places and *abilities* to output places.
6. The UIC registers the new *needs & abilities* at the service manager and available components get connected dynamically.

Scenario: **Modify Behavior**

Actor instances: `Alice,Bridget,Claudia:UIdesigner,UIC:UIcontroller`

Flow of Events:

1. The user interface designed by Alice is up and running. Now Bridget enters the laboratory. She is an expert in human factor engineering. Alice and Bridget evaluate the user interface and its functionality together.
2. Bridget likes the multi-modal interface but thinks that one has to do too many steps to fulfill a task. To change that, Alice removes several places, transitions and arcs from the net.
3. Alice now refines the remaining transitions' actions such that each of them sends several commands at once. That reduces the amount of steps to be done to complete the task but also reduces the control over what is going on for the user.
4. Now Claudia, who is a mechanical engineer, comes into the room. She carries a wearable that has a new input device attached to it and a DWARF system running on it. She wants to test it within the setting Alice designed. Alice therefore refines one of the *needs* predicate so that the UIC gets connected to the new input device instead of an old one.

4.2 Use Cases

Here, I present the use cases for the UIC interactive runtime development environment, as extracted from the scenarios in Section 4.1.

Use Case: **Create New UI Description**

Initiated by: `UIdeveloper`

Communicates with: `UIcontroller`

Flow of Events:

1. *(Entry condition)* A running DWARF system.
2. To design a new prototype from scratch the UIC is started without providing any Petri net description.
3. To receive input the UIdeveloper inserts input places into the net structure.
4. To send commands to output components the UIdeveloper inserts output places.
5. To execute actions (compose commands in most cases) the UIdeveloper adds transitions to the net structure. The transitions include executable entities describing the transition's behavior, called *actions*. The UIdeveloper specifies the behavior and compiles it at runtime.
6. Finally the UIdeveloper connects places and transitions with arcs and thus specifies the data-flow.
7. *(Exit condition)* A user interface behavior description is available that can be executed by the UIC.

Use Case: **Modify UI Behavior**
Initiated by: `UIdeveloper`
Communicates with: `UIcontroller`
Flow of Events:

1. *(Entry condition)* A running DWARF system and a UI behavior description executed by the UIC.
2. During the usage of a new prototype the UIdeveloper recognizes several shortcomings of the user interface.
3. Just as in use case `Create New UI Description` the UIdeveloper can change the structure of the net, change the actions and refine the connections with other DWARF services at runtime.
4. *(Exit condition)* The functionality of the user interface has been adapted.

Use Case: **Integrate New Input Component**
Initiated by: `UIdeveloper`
Communicates with: `UIcontroller`
Flow of Events:

1. *(Entry condition)* A running DWARF system and a UI behavior description executed by the UIC.
2. A new input component becomes available to the UIdeveloper.

3. Without stopping the running UIC, the new component should be integrated into the current setup.
4. The UIdeveloper adds a new *need* to the UIC component which gets registered at the servicemanager.
5. The UIdeveloper also adds a new input place into the net structure.
6. Incoming events now get placed as tokens inside that new input place.
7. *(Exit condition)* The new input component has been adapted.

Use Case: **Integrate New Output Component**
Initiated by: `UIdeveloper`
Communicates with: `UIcontroller`
Flow of Events:
1. *(Entry condition)* A running DWARF system and a UI behavior description executed by the UIC.
2. A new ouput component becomes available to the UIdeveloper.
3. Without stopping the running UIC the new component should be integrated into the current setup.
4. The UIdeveloper adds a new *ability* to the UIC component which gets registered at the servicemanager.
5. The UIdeveloper also adds a new output place into the net structure.
6. The action of a transition generating output commands must be refined to produce commands appropriate for that new component.
7. *(Exit condition)* The new output component presents content to the user.

Use Case: **Load And Save**
Initiated by: `UIdeveloper`
Communicates with: `UIcontroller`
Flow of Events:
1. *(Entry condition)* None.
2. The UIdeveloper starts the UIC without providing any net or service description.
3. The UIdeveloper uses the UICs load&save dialogue to load both a Petri net description and a DWARF service description from the file system.

4. The UIdeveloper now refines the Petri net and the service description as described in the use cases `Modify UI Behavior`, `Integrate New Output Component`, `Integrate New Input Component`.
5. In the end the UIdeveloper uses the load&save dialogue to save both descriptions into a XML file again.
6. *(Exit condition)* The Petri net and service descriptions are stored in the file system.

4.3 Requirements

4.3.1 Functional Requirements

Functional requirements describe the interactions between the system and its environment independently of its implementation [7].

Design Petri nets To model user interface behavior Petri nets are used. To make the modeling process as easy as possible a graphical editor is needed to build Petri nets.

Specifiy Guards and Actions To have full control over the behavior of the user interface, a developer can utilize *actions* (encapsulated in transitions) to specify what happens when a transition fires, and *guards* to define constraints that incoming tokens have to fulfill in order to let a transition become active. These actions and guards must be changeable at runtime to guarantee the developer full flexibility in modifying the user interface behavior.

Modify Petri net at runtime In order to change the data-flow inside the UIC at runtime it is necessary that the developer can change the structure of the Petri net (places, arcs and transitions) while the UIC is running.

Specify Needs and Abilities All communication with I/O components is done via DWARF connections. To have full control over the connectivity structure it is necessary that the developer can add and remove *needs&abilities* and define *attributes* on *abilities* and *predicates* over *needs*. Again, all that must be possible at runtime.

Load and Save To avoid doing the same work over and over again, it must be possible to save and load both the Petri net and service descriptions.

4.3.2 Non Functional Requirements

Nonfunctional requirements describe the user-visible aspects of the system that are not directly related to its functional behavior [7].

Predefined Net Elements To simplify the task of creating Petri nets a set of predefined input places must be provided (representing the token types defined in the input taxonomy).

Action Templates There must be template actions available for common and recurring problems since the specification of actions requires some knowledge about DWARF (e.g. composing stuctured events) and the underlying Petri net concept.

Online Help System The learning threshold for the development environment would be lowered significantly by an online help system, that guides first-time users through the functionality.

4.4 Related Work

In the fields of 2D GUI design a lot of tools are available to create user interfaces with not much more than a few clicks. The Qt designer[1] or Kdevelop[2] are well known examples. Those tools have shown to ease the development of graphical applications significantly.

Visual programming [21] is another related approach. The goal of visual programming environments is to enable non-technicians to program computers without knowledge about programming languages or deeper insights into computers in general.

There are a lot of projects and systems for GUI design or visual programming and combinations of those. I chose to discuss only one project in more detail because it sticks out of the mass. Squeak is a system that has been built for educational purposes. It is used in several schools to teach children in such disciplines as math, physics and computer programming.

Squeak [26] is a self contained interactive development and execution platform written in Smalltalk. Its goal is to provide an environment for educational software that can be used and programmed by non-technical people and even children. Squeak's strict object-orientation allows users and programmers to manipulate literally everything - even the virtual machine executing Squeak and the GUI toolkit.

[1] www.trolltech.com
[2] www.kdevelop.org

It is remarkable how easy it is to achieve results within Squeak. Ten to twelve year old children are able to build simple race-car simulations or other graphical applications within minutes. That provided me with a vision. In this thesis I have explored first steps towards creating a platform that provides options to create UAR user interfaces very quickly, and to modify them at runtime. In the future the developed platform shall provide more and more flexibility to change and manipulate all elements of the user interface at runtime.

Chapter 5

Implementation

In this chapter I will describe how the requirements, gathered in Section 3.3 and Chapter 4, have been implemented. I will also describe all classes, interfaces (and the usage of those) that are part of the developed component.

The core component of the *Interaction Management* layer is the UIC. It combines the functionalities of *Dialog Control* and *Discrete Integration*. It interprets input tokens sent by the *Media Analysis* components and then triggers actions that are dispatched to components in the *Media Design* layer.
Throughout this chapter I will describe the UIC in increasing detail, starting with a description of how Petri nets are utilized for modeling user interface behavior (Section 5.1), and followed by a description of a Petri net execution engine that has been built (Section 5.2). Then I will describe the design decisions that have been made for the interactive runtime development environment (Section 5.3). Finally, I will describe the details of the component's implementation (Section 5.4). That section is rather technical since I describe my implementation of the DWARF UIC and the packages, objects and classes.

5.1 Petri Nets for Interaction Management

Petri nets have been chosen to model interactions, as is common practice in the field of workflow systems [1] (for a discussion on why Petri nets are the appropriate model for interaction management purposes, please refer to Section 3.5). In this section I give a short introduction to Petri nets and how they are utilized within the DWARF UIC.

A Petri net consists of places, tokens, arcs, and transitions. The arcs connect places and transitions. Places and arcs may have capacities. Transitions execute actions when fired.
Optionally all arcs can have guards on both ends. Guards can define constraints on the type and number of the tokens as well as on the value of the tokens. Transitions only fire

when all guards evaluate to true, meaning that all constraints are fulfilled. Transitions fire when all places at the end of incoming arcs contain tokens if the arcs do not have any guards.
Transitions are used to encapsulate atomic interactions in this approach. More complex interactions can be modeled by combining several transitions.

The characteristics exhibited by the activities in a multimodal user interface such as concurrency, decision making, and synchronization are modeled very effectively with Petri nets. In Figure 5.1 some of these characteristics are represented using a set of simple constructs:

Sequential Execution In the left Petri net of Figure 5.1 transition $t2$ can fire only after the firing of $t1$. This impose the precedence of constraints $t2$ after $t1$. This construct models the casual relationship among activities.

Concurrency In the middle net of Figure 5.1, the transition $t2$, $t3$, and $t4$ are modeled to be concurrent. A necessary condition for transitions to be concurrent is the existence of a forking transition that deposits a token in two or more output places. Such a construct could be used to manipulate three different output devices at once.

Synchronization In the right net of Figure 5.1, $t1$ will only be active if all places contain tokens. Synchronized transitions would be typically used to combine different modalities to perform a single task such as speech, gaze, and a button

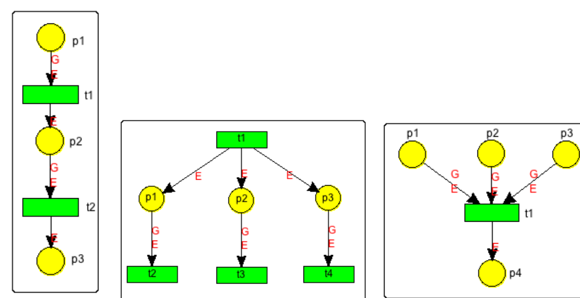


Figure 5.1: from left to right: Petri nets modeling Sequential Execution, Concurrency, and Synchronization

Petri nets are used to model interactions. Therefore constructs like the ones described above are utilized to model recurring problems in interaction management, such as integration of different modalities or the generation of content for output components. For a better understanding of the UIC's functionality one has to know how I/O devices and the Petri nets correlate with each other. As described in Section 2.2, the I/O components are adapted on a very high level of abstraction. That means that from

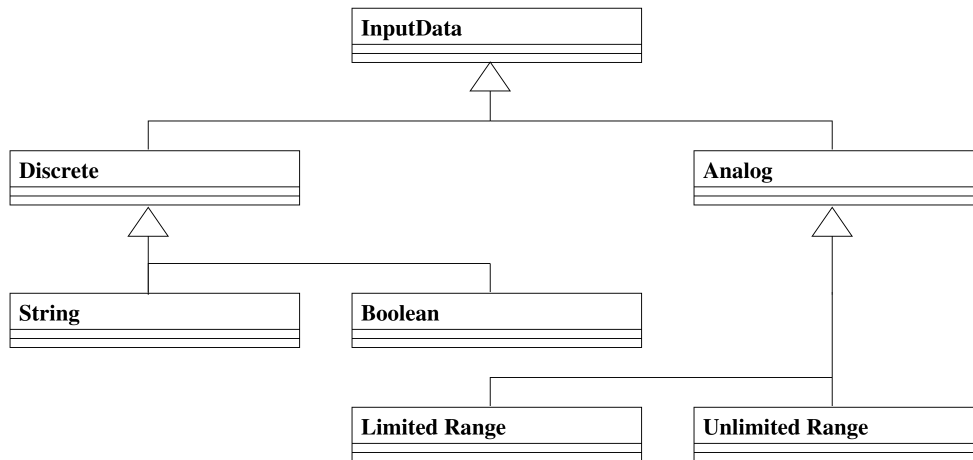


Figure 5.2: The taxonomy for DWARF input events.

the application programmer's point of view, every input device is regarded as a component that is emitting special DWARF structured events, specified in a taxonomy [57]. Figure 5.2 shows the taxonomy of input events. Every entity in that taxonomy correlates with a class of input devices that can be exchanged with each other. The following listing explains what type of data is contained in every entity and names some example devices:

Boolean All discrete data can be decomposed into booleans. Physical switches, for example, are mapped to boolean values. Also mouse buttons or the answer of a "yes/no" dialogue in a 2D GUI can be mapped to boolean values.

Limited Range Every analog value with a lower and upper bound can be mapped to a Limited Range entity. This entity has always a range between 0 and 1. Examples are sliders and dials. Also the orientation of a gyroscope is of the type Limited Range.

Unlimited Range Every irrational number which has only one or no bound. Examples for this are 6DOF coordinates of a tracked object. Any other sensor (e.g. thermometer) could also emit Unlimited Range tokens.

String Text strings are used as complex commands. For example textual input to an dialogue box in a 2D GUI or the tokens recognized by a speech recognition component.

Thus neither the UIC component nor the application programmer needs to know what input components are attached and how the underlying device is specified and implemented internally. The only thing that is relevant is which events can be sent or received.

For output components the situation is slightly different. Currently there is no taxonomy for output commands available. Due to the variety of human cognitive abilities, there are way more possibilities to present information to humans than to computers (see also Chapter 6). The lack of a specialized set of output events prevents the exchange of output components at runtime without modifying the transitions that construct the corresponding commands. For example do commands that are sent to 3D views contain implementation specific descriptions of geometric content (e.g. an Open Inventor [1] scene). And thus the receiver could not be replaced with another view that has the same semantic expressiveness but a different implementation (e.g. a pure OpenGL view).
Output components can however be attached and detached to the UIC in the same manner as input components. Currently there are no alternative implementations for the utilized output components available anyways.

A variety of input and output components is readily available for application programmers (for details see Section 2.2.3). The connections between the UIC and I/O components are managed via DWARF *needs&abilities*.

Events that are received by the UIC are handled as tokens that are put into incoming places of the Petri net. Those tokens then travel along the arcs through the Petri net. The actions encapsulated in transitions extract the content from user input tokens, and interpret it whenever a transition fires. Finally, new tokens containing commands for the *Media Design* components are composed. Tokens generated by transitions are sent to output components. Those commands cause state changes at the output components (e.g. the display of different content or a new dialog could be shown). That approach makes it possible to model interactions and implicitly the behavior of the whole user interface, without depending on the input or output devices attached to the control component.

Transitions play a very important role in modeling interactions because they contain the rules for the integration and modification of data contained in the input tokens. Transitions link inputs to a semantic entity. Transitions can be seen as *predicates* over input *attributes*. Whereas the *attributes* can specify the tokens' type, value, cardinality, or time of arrival. And only the right number and correct types of tokens can trigger a change in the system. A transition encapsulates actions which are executed when the predicate evaluates to true, and hence the transition fires.

A set of places, arcs, and transitions forms a expression, or in terms of user interaction, a declaration of intent. Such constructs (see Figure 5.1) can be used as patterns [19] - reusable entities to model common problems (e.g. insert an element into a view, selection/de-selection). In [30] a collection of patterns for Petri nets and examples for their application are described.

[1] a 3D graphics standard

5.2 The Petri Net Kernel

The described mechanisms have been implemented on top of the Java-based JFern [2] project. We take advantage of two main features, the ability to use arbitrary objects, in-

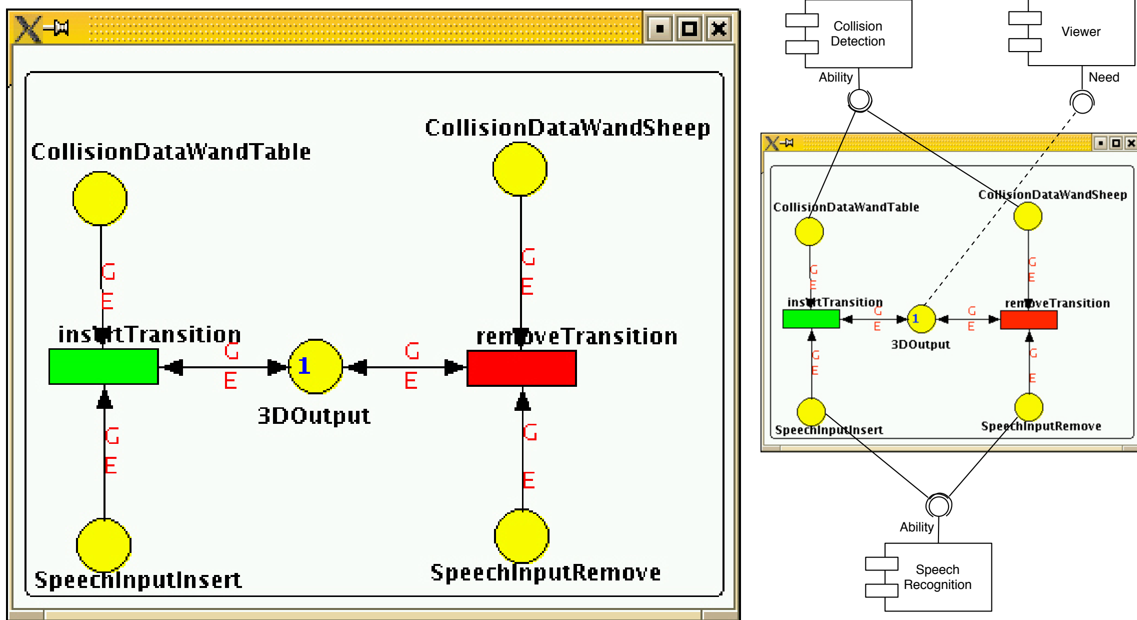


Figure 5.3: On the left, a Petri net that is executed within the JFern simulator, the currently active transition is highlighted in red. On the right, the same Petri net and how it is connected with the I/O components.

cluding DWARF structured events, as tokens and the possibility to describe transitions' guards and actions in native Java code. That implies a high expressiveness of the single statements and very little learning required for programmers familiar with the Java language. The following example code shows a guard on an input arc which checks if there is a single (one and only one) token in the input place, and checks if the token has an appropriate value:

```
public boolean guard() {
  //check the arity of the multiset
  if(getMultiset().size() != 1) return false;
  //check the condition
  Number t = (Number) getMultiset().getAny();
  if(t.intValue() == 10)
    return true;
```

[2] http://www.sourceforge.net/projects/jfern

```
  else
  return false;
}
```

The Petri nets are executed by the JFern simulator that does not only provide a convenient way of executing the designed control structure but also provides a simple visualization of the running user interface. Figure 5.3 shows a running user interface description.

A communication layer around the JFern kernel based on DWARF has been built, connecting input places with components of the *Media Analysis* layer and connecting output places with components of the *Media Design* layer. Figure 5.3 shows how a running Petri net is connected with I/O components.

To give more insight into the UIC component and how it works, I explain how interactions in an example system have been modeled. SHEEP [50, 36] has been built to demonstrate the possibilities of DWARF and especially its usefulness for building UAR user interfaces. SHEEP is a multiplayer game centered around a physical table with a pastoral landscape that contains a herd of virtual and plastic sheep. The landscape and virtual sheep are projected from a ceiling-mounted video projector.

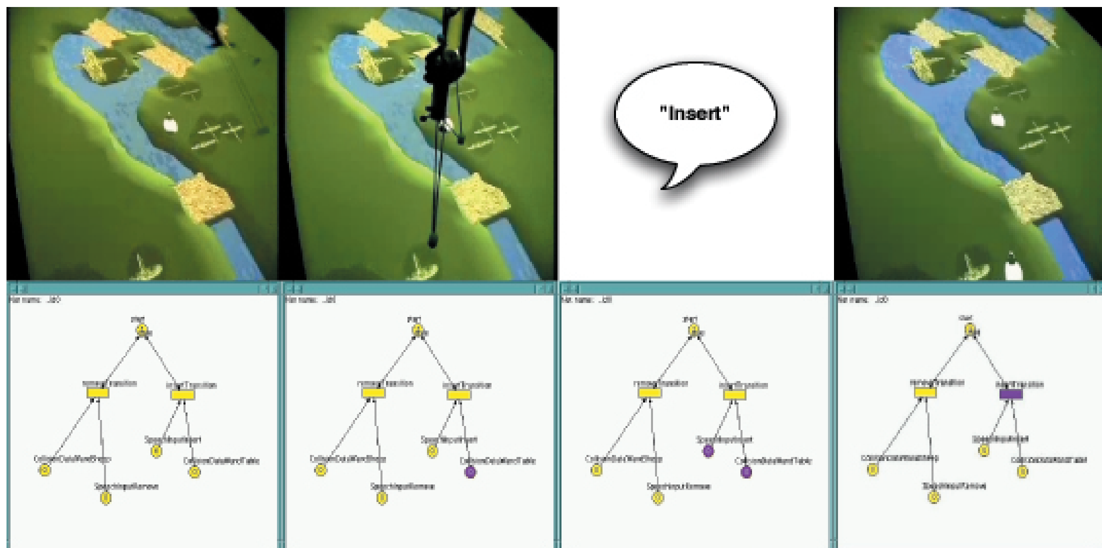


Figure 5.4: Point-and-Speech interaction in SHEEP. The tangible pointing device is used to indicate the position of a new sheep, the user then utters "insert" and a new sheep gets created. On the bottom one can see the Petri net in different stages.

Players can assume one of several roles. According to their different roles, players use different input devices and interaction technologies to interact with the game.

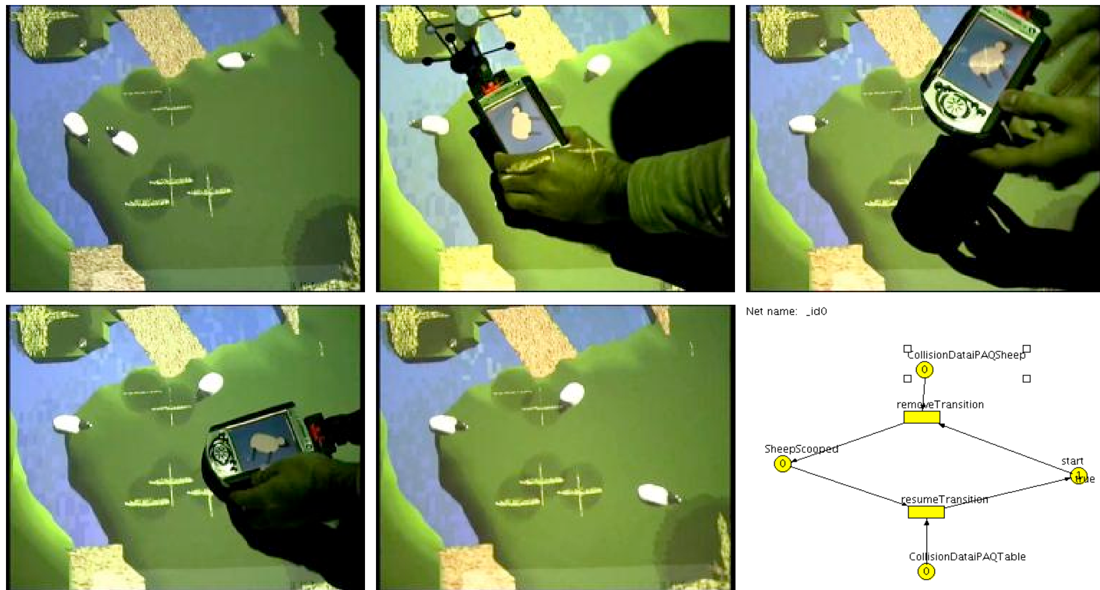


Figure 5.5: Sequence of images for Scoop-and-Drop interaction with a virtual sheep and an iPAQ. In the lower right corner, the corresponding Petri net is shown.

Creating and removing sheep With a microphone and a tracked magic-wand, a player can create new sheep and remove sheep from the table. This is done by multimodal *point-and-speak* input (Figure 5.4). This example illustrates how simple it is to model multimodal interactions with Petri nets.

Scooping sheep Players equipped with a tracked iPAQ can use it to scoop sheep up from the table. Scooped sheep can be dropped back somewhere else on the table (Figure 5.5). During the scooping operation, the scooped sheep is displayed on the palm-sized computer.
Within this example we modeled *causal constraints* - pick-up sheep before dropping them - and *temporal constraints* - do not pick-up recently dropped sheep immediately - with the corresponding guards.

5.3 Interactive Runtime Development

In Sections 5.1 and 5.2 I have described which formal approach is utilized to model interactions in UAR user interfaces, and how a runtime engine for that approach has been designed. The JFern package delivers a standard method to describe Petri nets, a combination of XML - for the net structure - and Java code - for the guards and transition declarations.

Since a pure source code based approach does not take advantage of the high

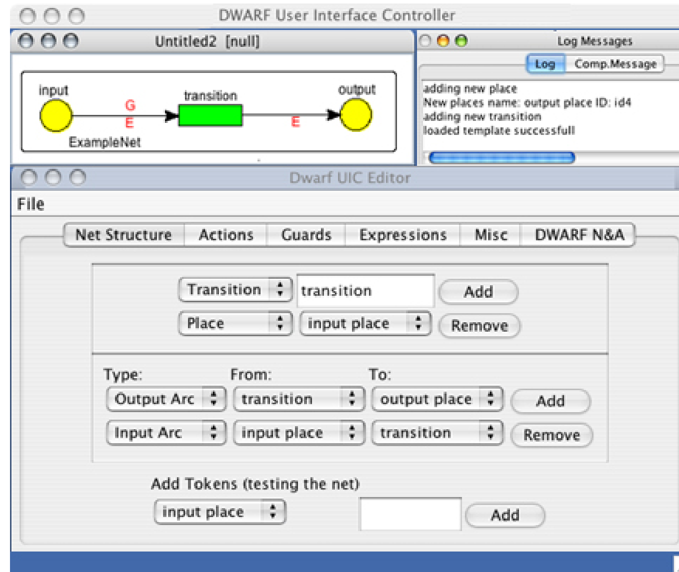


Figure 5.6: The DWARF UIC showing a very simple Petri net and the net structure modification tab.

expressiveness of the Petri nets graphical notation, the decision to build a graphical editor on top of that has been made. To render runtime prototyping of UAR user interfaces possible, a visual programming environment [28, 9] to specify Petri nets has been built. Besides specifying Petri net structure, the guard declarations and the transition declarations, the development environment must provide the developer with full control over the *needs & abilities* of the UIC and their mapping to input and output places. The visual programming environment allows developers to cover three main tasks:

1. Building and modifying the Petri net structure, and hence controlling the data flow and behavior of the complete user interface. Figure 5.6 shows the tool in net modification mode.

2. Modifying transition *actions* and *guards* on arcs.

3. Controlling and modifying DWARF *needs & abilities* dynamically (Figure 5.7 shows the UIC in *need&abilities* mode). This allows at-runtime connection and disconnection of devices from *Media Analysis* and *Media Design* layers. That also allows developers to constrain those connections by defining *attributes* and *predicates* on single connections.

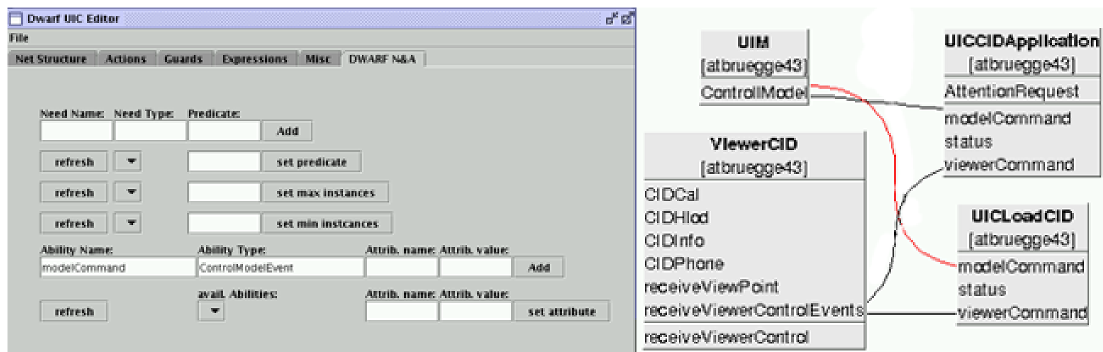
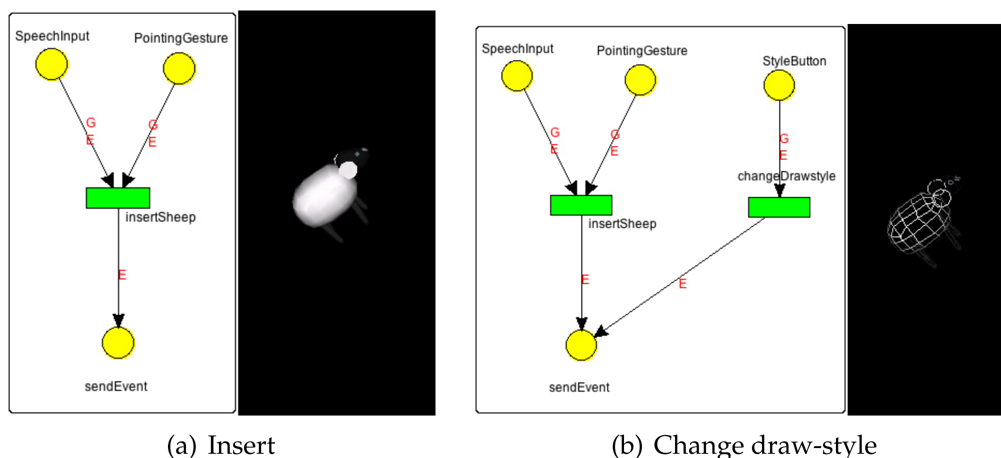


Figure 5.7: Adding a new ability to the DWARF UIC and the new connection shown in red.

5.3.1 Net Structure Modification

Modifying the net structure means to define the information flow through the UIC. On the one hand, that means to define how many inputs and what sort of inputs are needed to execute one task (e.g. a gesture and a speech command) and on the other hand, what sort of output is generated. With the net structure we also define how different tasks are related to each other (Figure 5.6 shows the UIC in net modification mode).

The following simple example shows how easy this method can be used to design user interface behavior.



(a) Insert (b) Change draw-style

Figure 5.8: a) A simple Petri net modeling the insertion of a sheep with voice and gesture commands. b) A new place and transition are inserted to the net to control the sheep's draw-style.

- The Petri net in Figure 5.8 shows a net that allows a user to insert sheep into a 3D scene utilizing speech commands and gestures (to assign a initial position).
- To change the sheep's draw-style, a new place and a transition are added to the net at runtime. The new place gets connected to a switch that can be used to change the draw-style of the sheep.

To keep the complexity of the nets as small as possible the concept of sub nets is introduced. Sub nets are small interaction entities that model one secluded, but not atomic, interaction (e.g. insertion of an object to a 3D scene coupled with the creation of a control entity for that object). Those sub nets can be inserted into the overall nets

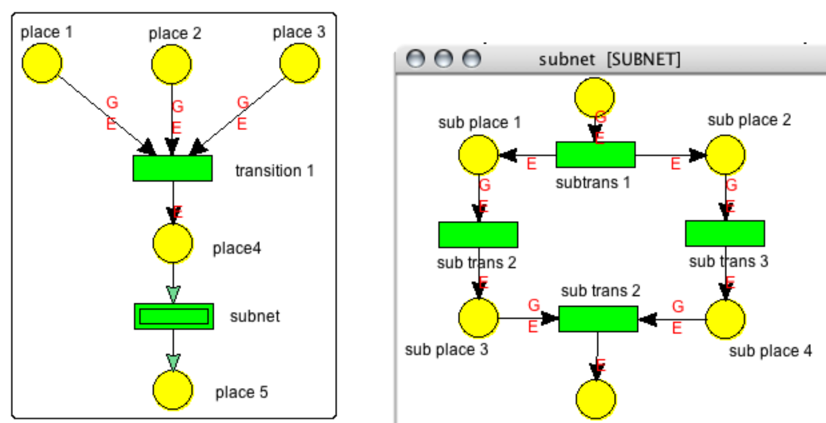


Figure 5.9: On the left, a Petri net containing a sub net. On the right, the sub net in a separate pop up window executing two transitions separately.

without showing all included places, transitions, and arcs (see Figure 5.9). Each net (including sub nets) can contain an unlimited number of sub nets. The current implementation allows us to add, remove, and edit all net atoms - places, transitions, arcs, and sub nets at runtime.

During the design phase of the UIC component, it was planned to add palettes of net elements to the GUI. From those palettes a user could drag and drop input places, output places, and special, predefined transitions for very common and recurring interactions into the current Petri net. Those palettes have been left out for future work due to time constraints (See Section 6.2).

5.3.2 Dynamic Code Modification

In the previous section I stated that the data flow through the user interface's control structure can be changed. To really change the behavior of the system, developers need

to modify the data manipulation that is done within the control structure. That means to exchange the code of the *guards* and *actions*. The *guards* check whether special constraints on the input are fulfilled (e.g. if there are three tokens of type speech command). The *actions* are executed when all *guards* on incoming arcs of the encapsulating transition evaluate to true and hence the transition fires. An action executes arbitrary Java code and has got access to the tokens that have been put into connected input places. Within the DWARF UIC actions are commonly used to compose DWARF structured events, which are used as tokens and can be sent to components in the *Media Design* layer. The code contained in actions boils down to a few lines in most cases. The following example code shows how a new token is composed out of two input tokens. That token will later be sent to a 3D viewer component to display a new object.

```
public void execute() {
    //get input tokens
    //identify tokens by class type
    UserInput ui = get(UserInput.getClass());
    CollisionData cd = get(CollisionData.getClass());

    //load 3D description from file
    String scene = getModel(ui.id, ui.name);

    if(scene!=null&&!(scene.equals(" "))){

      //Compose a new event that inserts an object to the scene:
      ViewerControlEvent command = new ViewerControlEvent();
      //create an object that can be moved:
      command.type = ViewerControlType.createConnectedObject;
      //give a name to the object:
      command.label = ui.id;
      //and to its pose data receiver:
      command.posedatareceiver = ui.id;
      //pass the object's 3D description:
      command.scene = scene;
      //give the object an initial position:
      command.keyvaluelist = { position cd.pose }
    }
}
```

Within JFern, actions are defined using pure Java code. That code gets compiled into a Petri net object which is executed by the JFern runtime engine.
The standard JFern mechanisms requires a developer to shut down the running Petri net, change the code, compile it, and start the execution again to change the actions that

are executed when a transition fires. This is not feasible for rapid prototyping purposes since it is necessary to change the whole user interface behavior at runtime.

To allow the exchange of the code executed when transitions fire a way to dynamically modify currently running Java code was needed. The standard Java API does not support dynamic code modification, which is essential for my approach. To surpass this problem a third party extension library for dynamic compilation [3] has been used. The Graham-Kirby compiler provides an interface to access the native Java compiler dynamically from a running program. A little source code editor and an extra text area to display error messages has been included with the visual programming environment. This makes it possible to rewrite the code of *guards* and *actions*, compile it and replace the original *guards* and *actions* at runtime. That enables developers to modify the user interface behavior dynamically.

Within the project CAR we made heavy usage of the possibility to refine the rules how the system reacts to user's input. The runtime development environment showed to be especially useful to model and control the behavior of the project CAR 's AUIs. AUIs monitor the user's visual attention and coordinate their behavior accordingly. For user interfaces in automotive environments it is very important to consume as little of the user's attention as possible since the user has to concentrate on the driving task.

In the project CAR we used an eye-tracking technology to measure when the user looked at the car's central information display (CID). Several techniques to attract user's attention (visual, spatial audio and combinations of those) have been used. Whenever the user looked at the currently active part of the user interface [40], the information displayed was adapted to show only relevant information. We used the UIC to control the described AUI (see Figure 5.10). We specifically used it to adapt the various sensors and filters and connect them to the graphical representation. We also utilized it to change the behavior of the user interface at runtime (e.g. duration and kind of attempts to get user's attention) and the content shown when the attention has been attracted. This has been done by replacing the rules encapsulated within the corresponding transitions at runtime.

5.3.3 Connectivity Management

So far I have described how Petri nets are utilized to model multimodal interactions. Now I show how the communication with attached components of *Media Analysis* and *Media Design* layers is controlled and modeled. Connections between all components are based on DWARF *needs & abilities*, and communication channels are set up at runtime. Developers can define attributes on *abilities* and predicates on *needs* to specialize the connection criteria [34].
Within the runtime development environment the developer can add new *needs* to attach input components to input places of the Petri net. Furthermore, the developer can

[3] www-ppg.dcs.st-and.ac.uk/Languages/Java/DynamicCompilation/

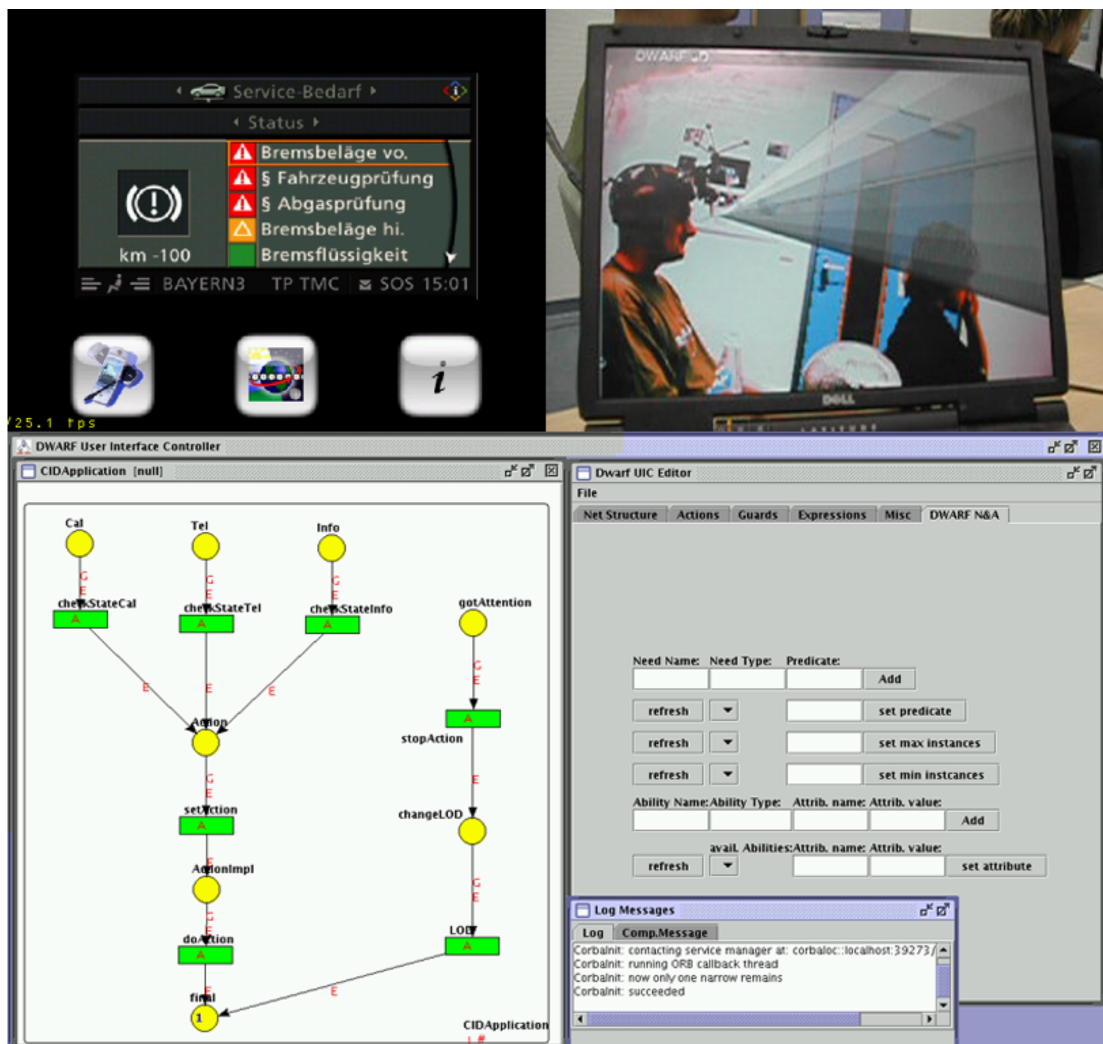


Figure 5.10: On top-left the CID, on top right the AR visualization of user's gaze. On the bottom the UIC we used to modify the AUI at runtime.

define predicates on what *needs* to select from different components which have *abilities* of the same type (e.g. *SpeechInput*). The connections will be set up whenever a matching pair of *needs & abilities* is present in the network environment. Whenever attributes or predicates change, the corresponding connections are disconnected and, if available, new communication partners are connected.

The DWARF approach allows developers (and even users) to detach, attach, or exchange input components at runtime, and thus experiment with different modalities. One could also think of replacing currently unavailable components by others that simulate the behavior of the original component, which showed to simplify the testing process in systems incorporating a variety of rather complex devices.
Abilities and output places can be modified accordingly. This allows developers to flexibly adapt different output components, and control which components receive which commands. So application programmers can use different modalities to present content to the user, or show different content on devices belonging to different users or user groups, for example private vs. public information. Alternatively, one can set up one-to-many connections so that many output components are connected with one output place e.g. controlling several views simultaneously. Figure 5.11 shows a sample connectivity structure illustrating different possibilities to connect user interface components.
Another aspect of the architecture allows developers to keep full control over the granularity of their Petri nets. Since any arc in a Petri net can be replaced by a DWARF connection, a developer can model everything in one self-contained component or on the other end have several interwoven components each modeling just one single interaction. Such atomic Petri nets can then be reused in different applications.
This is also interesting in aspects of ubiquitous computing, where several more or less independent UICs can connect to each other at runtime and thus form a richer, more powerful control structure enabling user interface aspects not available to the single sub applications. That lets users roam freely in a building, not only carrying a special input device with them (e.g. a PDA), but also the control component for that device, which can connect dynamically to different running applications and thus enable the user to participate and benefit from those emerging applications. DWARF connections can, formally spoken, extend the reachability graph of fused Petri nets.

5.4 Implementation Details

In this section I describe how the described concepts have been implemented. This Section is mainly of interest for those being interested in further developing the DWARF UIC.

My descriptions will increase in detail throughout this section. First, I want to give a overview of what programming languages, techniques and libraries have been used and how they correlate to each other. To get a first idea of the UIC implementation one has to look at the layers that constitute the UIC component. Figure 5.12 shows

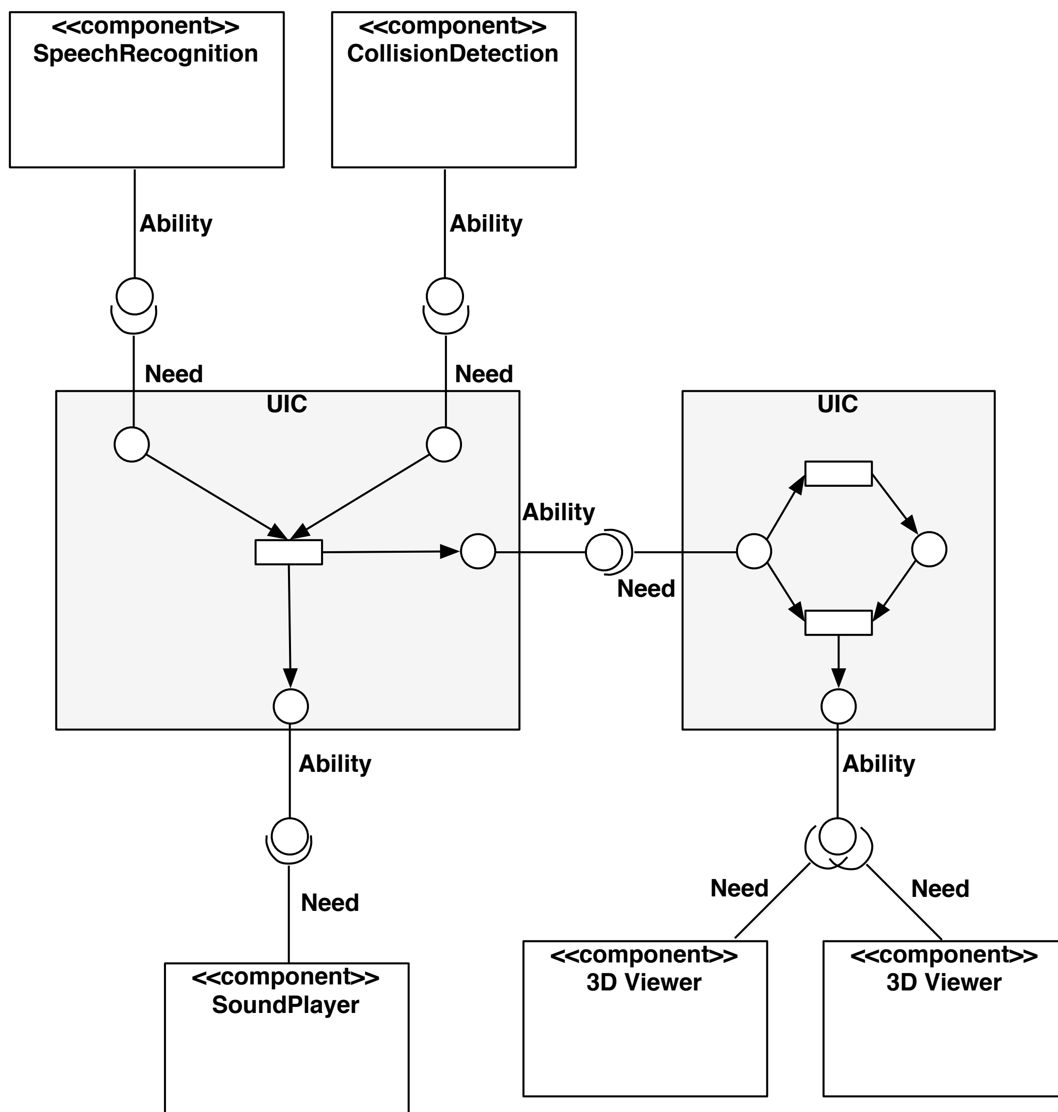


Figure 5.11: Schematic DWARF user interface incorporating different connection possibilities.

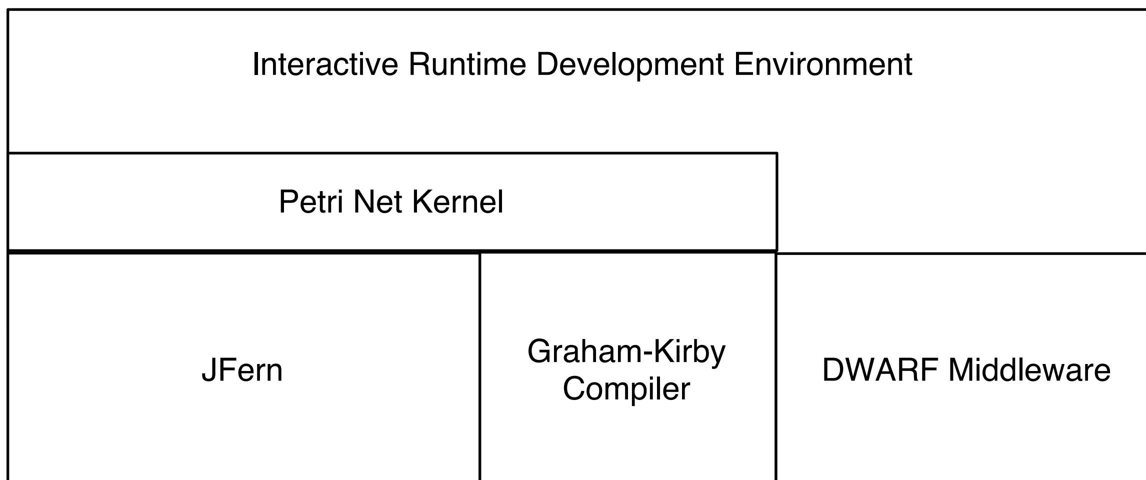


Figure 5.12: The layering of the DWARF UIC.

the component's layering. At the bottom are the DWARF middleware, the Graham-Kirby Compiler and the JFern engine. The DWARF middleware is used to communicate with other components of the system. This is done via DWARF structured events and remote method calls. Whereby the structured events hold the overwhelming majority. Structured events are used as tokens inside the Petri net, and also as commands that are sent to the output components.

The JFern engine is used to load and store the Petri net descriptions and, more importantly, to execute the Petri nets in realtime.

On top of the available infrastructure for Petri net execution and network communication, I have built a Petri net kernel that functions as an interface between the JFern engine and the application. This layer handles all modifications on the net structure and the addition of tokens into places, as well as the sending of commands to the ouput components. Finally it provides means to dynamically compile code of the guards and transitions.

The topmost layer is the interactive runtime development environment. That layer contains the GUI of the UIC and uses the functionality of the Petri net kernel to carry out changes on the net made by the user. It also interfaces with the DWARF middleware to handle changes on the UIC's *needs* and *abilities* .

In the next step I describe the classes that aggregate to the needed functionality. Figure 5.13 shows all classes that have been implemented. I encapsulated several classes into packages to increase the expressiveness of the diagrams. Each of the packages contains classes with related functionality, since there is still a strong correlation between those packages I did not break up the UIC component into smaller subsystems.

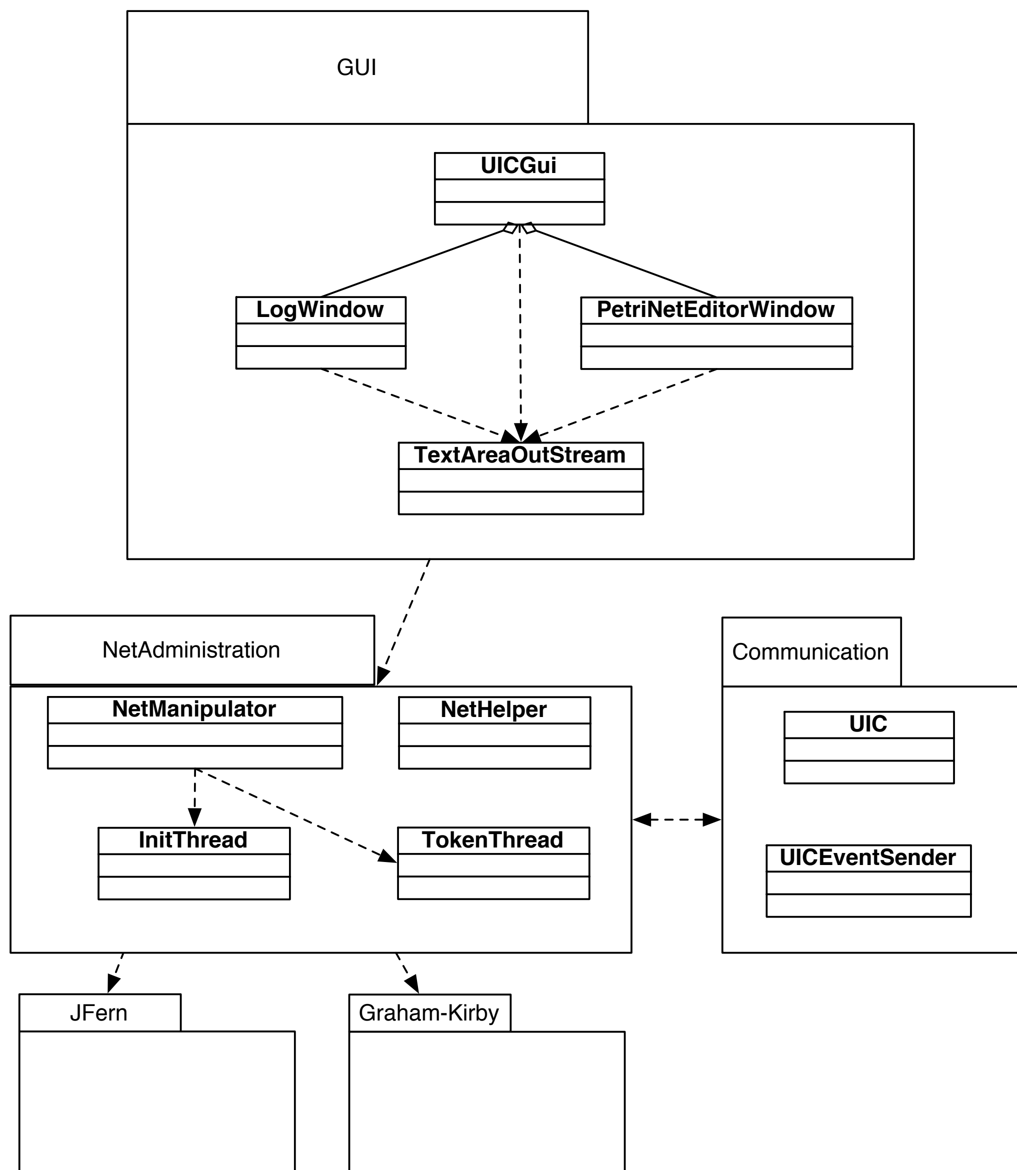


Figure 5.13: The packages of the DWARF UIC component.

5.4.1 Communication and Event Processing

The *Communication and Event Processing* package contains two classes. The `UIC` class handles all DWARF startup issues and provides methods to change and query the component's service descriptions:

- `addAbility()` Adds a new ability to the service description and registers it with the servicemanager.
- `deleteAbility()` Deletes an ability from the current service description.
- `getAbilities()` Returns a list of all currently registered abilities.
- `setAttribute()` Set a new attribute for an existing ability.
- `getAttributes()` Returns a list of all attributes that are defined for an ability.
- `addNeed()` Adds a new need to the service description and registers it with the servicemanager.
- `deleteNeed()` Deletes a need from the current service description.
- `getNeeds()` Returns a list of all currently registered needs.
- `setPredicate()` Set a new predicate for an existing predicate.
- `getAttributes()` Returns all predicates that are defined for the specified need.
- `getXMLServiceDescription()` Returns the current service description in XML format allowing it to be written to a file.

Since the `UIC` class handles all startup issues, it also instantiates the Petri net kernel classes and some helper classes for communication and net modification purposes. It also provides accessory methods that return a reference to those helper classes. The `UIC` class is designed as a singleton class.

The `UICEventSender` class has been developed to carry out the composed commands to the output components. The number of different types of DWARF structured events is constantly increasing. Because of that, the `UICEventSender` has been designed to be completely generic. Since every event type got different CORBA skeleton classes within DWARF , it is necessary to know the type of the event before it can be sent. The `UICEventSender` utilizes the Java reflection API to gather information about the current event that ought to be sent, and based on that information packs the event and sends it to the output components. There is one instance of this class for every ability the UIC component has got.

5.4.2 Net Manipulation

The classes inside the *Net Manipulation* package are the interface between the JFern engine and the rest of the application. They do allow both changes on the Petri net and all activities needed to execute the Petri net, such as addition and removal of tokens and control of the JFern Petri net simulator. The *Net Manipulation* package forms together with the JFern engine the *Model* in a MVC[19] pattern architecture.

The most important class in this package is the `NetManipulator` class. At startup it instantiates all other classes in this package, stores references to them and initializes the JFern runtime engine. It also compiles the Petri net description if one is given, otherwise a new, empty Petri net is created. Further, the JFern simulator is started and attached to the newly created Petri net instance.
The `NetManipulator` class offers a variety of methods to modify the Petri net and to query information about it after the Petri net startup.
To get a detailed documentation of all methods please refer to the API documentation that is available online [14], since explaining every method would go beyond the scope of this thesis. The methods offered by the `NetManipulator` class include the addition and removal of all net atoms - places, arcs, transitions, sub-nets, and tokens. Following the MVC pattern [19], the `NetManipulator` class notifies all attached views whenever the Petri net changes. There are methods available to query information about the whole Petri net and about every single element contained in it. The `NetManipulator` class provides methods to configure and invoke the dynamic compiler for exchanging the code of the guards and actions.
And at last the class enables the developer to save created or modified Petri net descriptions into an XML file.

The `InitThread` class handles the startup and initialization of the JFern net simulator. That is done whenever at least one of the `UICEventSender` instances gets connected to one consumer. From that point on the component is able to process incoming events and send out new ones.

The `TokenThread` class is a wrapper class around the JFern Petri net execution mechanism (the net simulator). This class takes incoming DWARF structured events as input, identifies the corresponding place inside the Petri net, and places the event as token inside that place. Finally the net simulator is notified to process the nets new marking. That means, that all transitions are tested if they are activated (willing to fire) and if there are any they fire.
To guarantee the responsiveness of the Petri net execution a new thread for every token is started. Some transition might take long time to execute their actions but because of the multithreaded approach that does not stop the rest of the Petri net from processing data. To avoid consistency issues with that approach (eventual loss of tokens) during any net structure modifications, those modifications are only carried out, when no transition is firing or ready to fire.

The `NetHelper` class provides a variety of convenience methods for the user inter-

face designer that can be accessed from the actions encapsulated in the transitions. The current implementation includes methods to load 3D scene descriptions or sounds from the file system that can be sent to output components. The loaded files are accessible throughout the whole runtime of the Petri net, thus the file system access is minimized. The `NetHelper` class also provides the user interface designer with access to code templates for modeling common recurring problems within actions.

5.4.3 The Graphical User Interface

The *Graphical User Interface* package includes all classes that form the GUI of the UIC component. These classes are used to give input to the system and to present the current state of the Petri net to the developer. These classes are the *View* (Petri net visualization) and the *Controller* (editing controls) parts of the MVC [19] architecture.

The whole GUI follows the desktop metaphor known from common computer operating systems, where different windows with different functionalities are grouped together in one container - the desktop.

The GUI creation and window event processing - dragging windows around, minimize, maximize, and close them - is handled by the `UICGui` class. When the UIC is started a main window pops up containing a window with a Petri net view, an editor window, and a logging window.

The `PetriNetEditorWindow` contains all controls to change the net structure, the properties of net elements, and to change the code of guards and actions. Further it contains controls to change the connectivity structure via DWARF *needs & abilities*. In the current implementation the net modifications are mostly done via dropdown boxes, text fields, and buttons which provides rather bad usability as soon as the Petri nets grow in complexity. This should be changed to direct manipulation techniques following the *tool* metaphor commonly used in 2D GUI systems.

The `LogWindow` class provides a little window with two tabs. It displays normal log messages in one tab and compiler messages and errors in the other one. I decided to overwrite the standard Java *OutStream* class inside the `TextAreaOutStream` class to have full control over the system's standard and error output streams. Thus log and error messages are printed to the `LogWindow`'s tabs instead of the normal console. The `TextAreaOutStream` can be used to customize the format of the printed messages.

Chapter 6

Conclusion

It is rather complicated to design and implement user interfaces for UAR systems. Especially because the standardization of HCI that helped 2D graphical user interfaces to become tremendously successful, has not been achieved for 3D or UAR user interfaces. Nor is a common set of I/O devices for UAR applications available. UAR user interfaces usually have custom tailored solutions for the usage of I/O devices, whether it is mouse/keyboard, 6DOF trackers, speech or gesture recognition. Binding the I/O devices close to the application makes it cumbersome to tweak, improve, and experiment with interaction techniques.

I have presented a method that deals, at least, with parts of that problem. I think that the proposed solution works especially well for I/O device abstraction and adaption of I/O components, and also for modeling the information flow inside UAR user interfaces. The described approach has successfully been used in various systems [11]. I think that it bears a lot of potential for the development of UAR user interfaces.

While full flexibility for exchanging input devices at run-time has been achieved, that flexibility couldn't be fully achieved for output components. The underlying problem is rather complex. It turned out that it is very difficult to define the semantic expressiveness of an output component, e.g. which auditory interfaces can be mapped to GUIs and which cannot? For input components the definition of expressiveness was relatively simple, because the receiver of emitted tokens is a *computer*. For output components, the receiver of content is a *human*. In the current state of implementation output commands contain implementation specific information (e.g. Open Inventor scenes). That restricts the runtime exchange to components that can understand the same format for information.

There have been efforts to assemble a taxonomy for all output modalities [5]. I doubt that such a taxonomy would be applicable for UAR systems with reasonable effort. Because the human perception and cognition is very complex, a taxonomy that addresses all its aspects would necessarily become bulky. Furthermore would an adapter for every output component be necessary, to translate tokens from that taxonomy into a format

that can be understood and displayed by the component.

The Petri net transitions function as adapters for the output components in the current implementation. The transitions translate the abstract input that is coming from the *Media Analysis* layer into, implementation specific, commands for the output components. The decision wether to put effort into the development of such an output taxonomy has to be evaluated carefully. Currently there are no output components of equal expressiveness that could be exchanged for each other, available for DWARF programmers. So the benefit of a taxonomy would be very limited right now, but that might change in the future.

The interactive runtime development environment has reached a level where programmers that have a significant level of experience in the usage of computers (e.g. graduate students in computer science) and especially DWARF can use it to quickly assemble and tune UAR user interfaces. It proved to speed up the development of UAR user interfaces significantly within the CAR project. I also used the new UIC to develop little example systems within few minutes during the test and development phase. However it needs more work to make it useful for less experienced programmers or even the end user in the long term.

Since the UIC communicates with a component abstraction, the developed component is not restricted to the control of I/O devices but could as well be used to model application logic or workflow definitions. It is virtually predestinated to control distributed systems because of its foundation on Petri nets, that have been developed to model distributed systems.

6.1 Lessons Learned

I had to deal with many new situations and problems during the project CAR . Some of them required a lot of research and preciseness, others required spontaneous and rather unconventional solutions. But after half a year and a lot of work I consider project CAR and my thesis to have resulted in a lot of deliverables, which might be of value for the DWARF project and function as a basis for further research.

6.1.1 Social Lessons

Because the project CAR was a team effort of eight students, a lot of (self-) management and communication was required. I have learned a lot about collaboration and team work.
Since we had a real client from the automotive industry, several presentations and live demonstrations have been given to that client. This taught me some tough lessons about time management and critical deadlines that have to be kept. I did also gather remarkable insights into marketing issues.

The variety of topics that have been regarded in the different work that form the project CAR have led to a lot of very interesting discussions with the other team members and our supervisors. That showed me lots of interesting issues in all areas of computer science and how important communication is.

Furthermore, we wrote one Technote [24] for a workshop at the International Conference on Intelligent User Interfaces[1] in Madeira, Portugal. We also submitted one full paper that is currently under review for the Symposium on User Interface Software and Technology[2]. I learned a lot about scientific work and reasoning during that work.

6.1.2 Technical Lessons

During the project CAR I have been confronted with several technical problems. Among them networking, computer graphics, GUI design, and software engineering issues. I also had to deal with three different programming languages, namely Java, C++, and Python. Furthermore I made extensive use of XML as preferred technique to describe and store all sorts of configurations.
To document the project CAR the team produced two videos [13] whereby I learned a lot about video technology and video editing software.

6.2 Future Work

In this Section I want to discuss features that have been designed but not implemented and new ideas that have been developed during the implementation and testing phase of the CAR project.

6.2.1 The Runtime Development User Interface

In the current implementation stage, the UIC development environment does not resemble the latest state of the art for 2D HCI. For example, drop-down boxes and text fields are used to modify the net structure instead of a tool metaphor based approach that is commonly used in graphics and layout software. This is partially due to the limited GUI support of the JFern version that was used at the time of implementation. Right now, a new version of JFern is available that includes dramatically improved GUI support. Figure 6.1 shows a non-functional mock-up of a possible new user interface utilizing the new features in JFern.

Another limitation of the current implementation is that the runtime development environment can only be used to work on exactly one Petri net. So for every new Petri

[1] www.mu3i.org
[2] http://www.acm.org/uist/

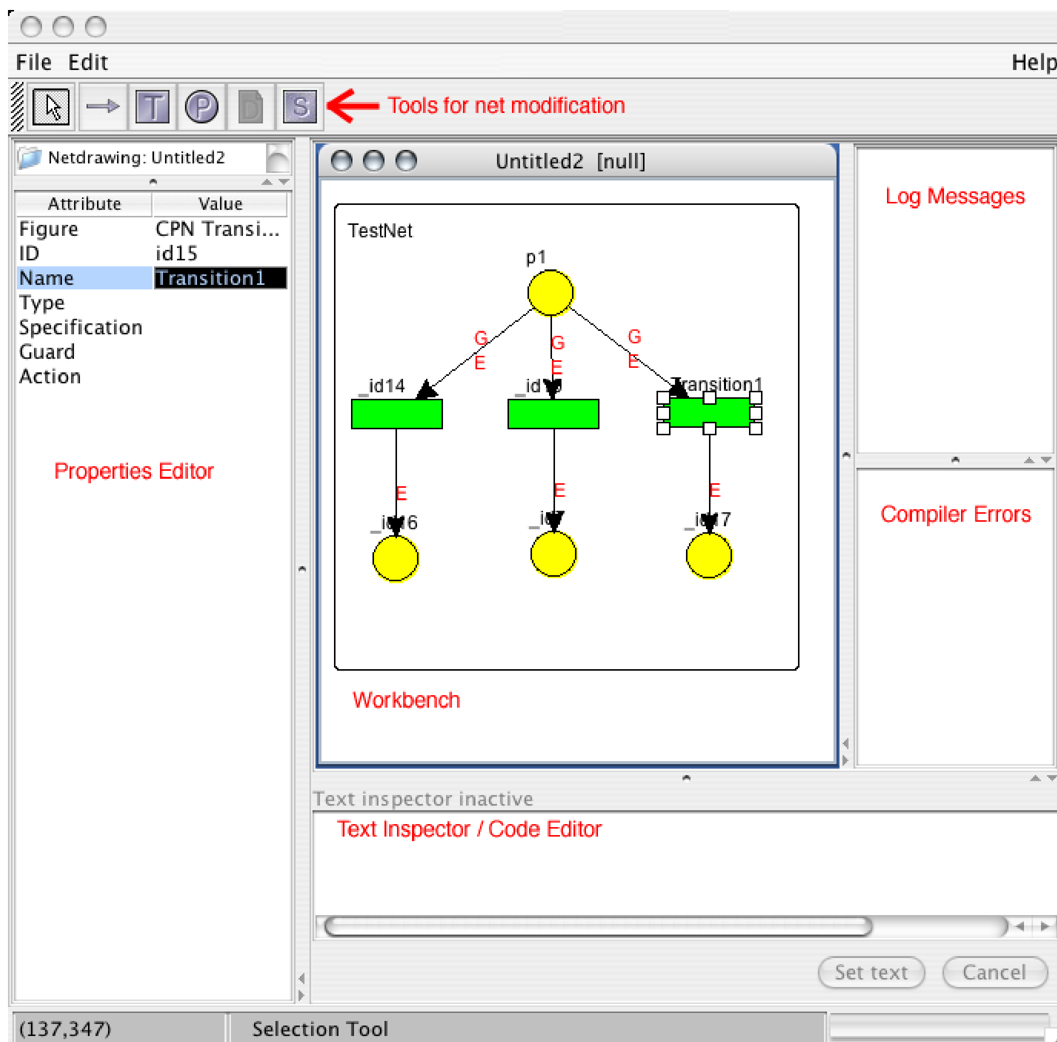


Figure 6.1: Mock-up for a new version of the DWARF UIC.

net a new development environment has to be started too. This is especially inconvenient if a developer has chosen a highly modular approach (e.g. one Petri net for each interaction), because in that case the desktop would be cluttered with several UIC windows. A solution to that problem would be to decouple the Petri net execution from the development environment, so that the running Petri nets could be attached and detached to and from the development environment. The current software architecture already would support such a mechanism but the GUI does not at this point.

To further speed up the development process, a repository of reusable interaction entities could be implemented. Developers could drag and drop net atoms or complete sub nets from that repository into the Petri net that is currently developed. Such a repository could contain typed input places (according to the taxonomy) and transitions that contain interaction patterns that are commonly used in UAR applications (e.g. addition/removal of 3D objects).

Finally, an integration with the DWARF filter network for continuous integration, developed within the CAR project, would be desirable. Such an integration would allow developers to model the complete data-flow through the system from one component; discrete events with the UIC and continuous data streams with the pipe and filter architecture.

Alternatively could the UIC be extended to process continuous data streams within the Petri nets. While is theoretically possible, I would expect performance issues if a lot of transitions would execute complex computations at a very high frequency. In addition, a continuous arrival of events would cause the Petri net execution engine to produce a significant overhead.

6.2.2 Programming by Example

Currently there is still some programming needed to define the actions and guards of the Petri nets. In the future this could be reduced or, ideally, completely replaced by a programming by example approach. For example could a developer use a microphone to record a message and then point at a speaker to signify that it should be played later.

6.2.3 Authoring Within Augmented Reality

I am also thinking of moving the whole development environment into the AR domain, using the same interaction techniques and output devices used in the running application to modify the control structure on the fly. Authoring UAR applications in UAR would be especially interesting in mobile settings where no classic 2D desktop is available. Also the possibility to display the Petri net in 3D would increase the overview over more complex nets and facilitate the navigation within the net.

A tighter integration of the development environment with its target system would additionally lower the learning threshold for both, since users could apply the knowledge, once it has been gathered, to both applications.

In addition to author the control structure of the UAR system within itself, it would be desirable to be able to create 3D content with AR interaction and visualization techniques. Such an approach would in my opinion be applicable to a wide variety of applications, such as industrial design, architectural design and 3D computer graphics design.

6.2.4 System Feedback

For now the user interfaces that are designed with DWARF are able to give feedback to the user whether an interaction succeeded or not. In contradiction, if errors occur or the user interface gets in a disfunctional state the control structure has to be modified manually. Currently there is not even a sophisticated exception handling mechanism available.
One could think about a system feedback mechanism were the system, once running, monitors and maintains itself.

6.2.5 Extensions for the DWARF UI Architecture

The UI architecture could be extended to further improve the quality of applications that can be built with DWARF .

Semantic Interpretation An additional layer between the *Media Analysis* and the *Interaction Management* layer could be introduced. That layer had to be configurable in such a manner that it could use artificial intelligence techniques in combination with application domain knowledge to semantically interpret the different input modalities. Such an approach would increase the robustness of the multi-modal integration process, and hence make the user interface less error prone.

User Model To configure the user interface to fit the needs of all users best, a user model could be implemented. That user model would observe the user and try to predict the user's intention and thus help users to fulfill their task more efficiently.

Device Integration Currently, developers have to write low-level device drivers and adapters to integrate new hardware into the DWARF framework. An API or generic driver framework would be desirable to reduce the amount of knowledge and time needed for that process.

Appendix

Chapter 7

Abbrevations

1D - One-dimensional

2D - Two-dimensional

3D - Three-dimensional

UI - User Interface

GUI - Graphical User Interface

HCI - Human Computer Interaction

WIMP - Windows Icons Menus Pointers

AUI - Attentive User Interface

AR - Augmented Reality

UAR - Ubiquitous Augmented Reality

UIC - User Interface Contoller

DWARF - Distributed Wearable Augmented Reality Framework

SHEEP - Shared Environment Entertainment Pasture

CAR - Car Augmented Reality

API - Applications Programmer's Interface

HMD - Head Mounted Display

WIM - World In Miniature

CORBA - Common Object Request Broker Architecture (An open standard for distributed computing and network communication)

Bibliography

[1] W. AALST, *The Application of Petri Nets to Workflow Management*, The Journal of Circuits, Systems and Computers, 8 (1998), pp. 21–66.

[2] R. AZUMA, *A survey of augmented reality*, 1995.

[3] M. BAUER, B. BRUEGGE, G. KLINKER, A. MACWILLIAMS, T. REICHER, S. RISS, C. SANDOR, and M. WAGNER, *Design of a Component–Based Augmented Reality Framework*, in Proceedings of the 2nd International Symposium on Augmented Reality (ISAR 2001), New York, USA, 2001.

[4] B. BELL, S. FEINER, and T. HOLLERER, *View Management for Virtual and Augmented Reality*, in Proceedings of UIST'01, 2001, pp. 101–110.

[5] N. O. BERNSEN, *A Toolbox of Output Modalities - Representing Output Information in Multimodal Interfaces* . CCI Working Papers in Cognitive Science and HCI, WPCS-95-10, 1995.

[6] G. BLASKO and S. FEINER, *A Menu Interface for Wearable Computing*, 6th International Symposium on Wearable Computers (ISWC 2002), (2002), pp. 164–165.

[7] B. BRÜGGE and A. H. DUTOIT, *Object-Oriented Software Engineering. Conquering Complex and Changing Systems*, Prentice Hall, Upper Saddle River, NJ, 2000.

[8] A. BUTZ, C. BESHERS, and S. FEINER, *Of Vampire Mirrors and Privacy Lamps: Privacy Management in Multi-User Augmented Environments*, in ACM Symposium on User Interface Software and Technology, 1998, pp. 171–172.

[9] W. CITRIN, M. DOHERTY, and B. ZORN, *Design of a Completely Visual Object-Oriented Programming Language*, in Visual Object-Oriented Programming, M. Burnett, A. Goldberg, and T. Lewis, eds., Prentice-Hall, New York, 1995.

[10] J. M. DAVIES, *An Ambient Computing System*, Master's thesis, Department of Electrical Engineering and Computer Science at the University of Kansas, Kansas, United States of America, 1998.

[11] DWARF, *Complete list of DWARF based Projects*.
http://www1.in.tum.de/DWARF/ProjectsOverview.

[12] DWARF, *The ARCHIE Homepage*.
http://www1.in.tum.de/DWARF/ProjectArchie.

[13] DWARF, *The CAR Homepage*.
http://www1.in.tum.de/DWARF/ProjectBar.

[14] DWARF, *The DWARF Homepage*. http://www.augmentedreality.de.

[15] E.HORVITZ, C.KADIE, and D.HOVEL, *Models of Attention in Computing Communication: From Principles to Applications*, Communications of the ACM, 46 (2003), pp. 52–59.

[16] R. ESSER, J. JANNECK, and M. NAEDELE, *Applying an Object-Oriented Petri Net Language to Heterogeneous Systems Design*, in Proceedings of Workshop PNSE'97, Petri Nets in System Engineering, 1997.

[17] C. FAURE and L. JULIA, *An Agent-Based Architecture for a Multimodal Interface*, in Proceedings of AAAI'94 - IM4S (Stanford), pp. 82-86., 1994.

[18] S. FEINER, B. MACINTYRE, M. HAUPT, and E. SOLOMON, *Windows on the World: 2D Windows for 3D Augmented Reality*, in ACM Symposium on User Interface Software and Technology, 1993, pp. 145–155.

[19] E. GAMMA, R. HELM, R. JOHNSON, and J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Publishing Company, 1994.

[20] D. GARLAN, D. SIEWIOREK, A. S MAILAGIC, and P. STEENKISTE, *Project Aura: Toward Distraction-Free Pervasive Computing*, IEEE Pervasive Computing, 1 (2002).

[21] E. P. GLINERT, *Visual Programming Environments: Paradigms and Systems*, IEEE Computer Society Press, 1990.

[22] O. HILLIGES, *Development of a 3D-Viewer for DWARF based Applications*. Systementwicklungsprojekt, Technische Universität München, 2003.

[23] O. HILLIGES, C. SANDOR, and G. KLINKER, *Interactive Prototyping of Interaction Management for Ubiquitous Augmented Reality Systems* . Under review for the UIST'04 conference proceedings., 2004.

[24] HILLIGES, OTMAR AND SANDOR, CHRISTIAN AND KLINKER, GUDRUN, *A Lightweight Approach for Experimenting with Tangible Interaction Metaphors*, in Proc. of the International Workshop on Multi-user and Ubiquitous User Interfaces (MU3I), 2004.

[25] J. E. HOPCROFT, R. MOTWANI, and J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Pearson Addison Wesley, 2000.

[26] D. INGALLS, T. KAEHLER, J. MALONEY, S. WALLACE, and A. KAY, *Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself*, in Proceedings of OOPSLA '97, ACM SIGPLAN Notices, November 1997, pp. 318–326.

[27] H. ISHII and B. ULLMER, *Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms*, in Proc. CHI 97, Atlanta, USA, March 1997, ACM.

[28] R. J. JACOB, *A State Transition Diagram Language for Visual Programming*, in Visual Programming Environments: Paradigms and Systems, E. P. Glinert, ed., IEEE Computer Society Press, 1990.

[29] R. J. K. JACOB, L. DELIGIANNIDIS, and S. MORRISION, *A Software Model and Specification Language for Non-WIMP User Interfaces*, ACM Transactions on Computer-Human Interaction, 6 (1999), pp. 1–46.

[30] J. JANNECK and M. NAEDELE, *Introducing Design Patterns for Petri Nets*, 1998.

[31] M. JOHNSTON, P. R. COHEN, D. MCGEE, S. L. OVIATT, J. A. PITTMAN, and I. SMITH, *Unification-Based Multimodal Integration*, in Proceedings of the Thirty-Fifth Annual Meeting of the Association for Computational Linguistics and Eighth Conference of the European Chapter of the Association for Computational Linguistics, P. R. Cohen and W. Wahlster, eds., Somerset, New Jersey, 1997, Association for Computational Linguistics, pp. 281–288.

[32] G. KORTUEM and J. SCHNEIDER, *An Application Platform for Mobile Ad-hoc Networks*, in Workshop on Application Models and Programming Tools for Ubiquitous Computing, Atlanta, USA, 2001.

[33] O. KUMMER, F. WIENBERG, and U. HAMBURG, *Renew - User Guide*, 1999.

[34] A. MACWILLIAMS, T. REICHER, and B. BRÜGGE, *Decentralized Coordination of Distributed Interdependent Services*, in IEEE Distributed Systems Online – Middleware Work in Progress Papers, Rio de Janeiro, Brazil, June 2003.

[35] A. MACWILLIAMS, C. SANDOR, M. WAGNER, M. BAUER, G. KLINKER, and B. BRUEGGE, *Herding SHEEP: Live Development of a Distributed Augmented Reality System*, The Second International Symposium on Mixed and Augmented Reality (ISMAR 2003), (2003).

[36] A. MACWILLIAMS, C. SANDOR, M. WAGNER, M. BAUER, G. KLINKER, and B. BRÜGGE, *Herding Sheep: Live System Development for Distributed Augmented Reality*, in Proceedings of ISMAR 2003, 2003.

[37] MICHEL BEAUDOUIN-LAFON AND WENDY E. MACKAY AND PETER ANDERSEN AND PAUL JANECEK AND MADS JENSEN AND HENRY MICHAEL LASSEN AND KASPER LUND AND KJELD HOYER MORTENSEN AND STEPHANIE MUNCK AND ANNE V. RATZER AND KATRINE RAVN AND SOREN CHRISTENSEN AND KURT JENSEN, *CPN/Tools: A Post-WIMP Interface for Editing and Simulating Coloured Petri Nets*, in ICATPN, 2001, pp. 71–80.

[38] P. MILGRAM and H. C. JR., *A Taxonomy of Real and Virtual World Display Integration*.

[39] D. MOLDT and F. WIENBERG, *Multi-Agent-Systems Based on Coloured Petri Nets*, in Application and Theory of Petri Nets 1997, 1997, pp. 82–101.

[40] V. NOVAK, *Attentive User Interfaces for DWARF*, Master's thesis, Department of Applied Software Engineering, Technische Universität München, 2004.

[41] M. NOWOSTAWSKI, *JFern, Java-based Petri Net framework*. http://sourceforge.net/projects/jfern.

[42] A. OLWAL and S. FEINER, *Unit: Modular Development of Distributed Interaction Techniques for Highly interactive user Interfaces*, in To appear in: Proceedings of International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia, 2004.

[43] S. OVIATT, *Ten Myths of Multimodal Interaction*, Communications of the ACM, 42 (1999), pp. 74–81.

[44] I. POUPYREV, M. BILLINGHURST, S. WEGHORST, and T. ICHIKAWA, *The Go-Go Interaction Technique: Non-Linear Mapping for Direct Manipulation in VR*, in ACM Symposium on User Interface Software and Technology, 1996, pp. 79–80.

[45] G. REITMAYR and D. SCHMALSTIEG, *OpenTracker–An Open Software Architecture for Reconfigurable Tracking Based on XML*, in Proceedings of VR, 2001, pp. 285–286.

[46] S. RISS, *An XML based Task Flow Description Language for Augmented Reality Applications*, Master's thesis, Department of Applied Software Engineering, Technische Universität München, 2001.

[47] ROEL VERTEGAAL, *Introduction: Attentive User Interfaces*, Communications of the ACM, 46 (2003), pp. 30–33.

[48] C. SANDOR, *CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces*, Master's thesis, Department of Applied Software Engineering, Technische Universität München, 2001.

[49] C. SANDOR and G. KLINKER, *Ubiquitous Augmented Reality: Towards a Unification of Current Paradigms in Human-Computer Interaction*. Accepted for the Journal of Personal and Ubiquitous Computing, 2004.

[50] C. SANDOR, A. MACWILLIAMS, M. WAGNER, M. BAUER, and G. KLINKER, *SHEEP: The Shared Environment Entertainment Pasture*, in Demonstration at ISMAR 2002, Darmstadt, Germany, 2002.

[51] B. SHNEIDERMAN, *Designing the User Interface*, Addison-Wesley Publishing, 1997.

[52] A. SINGHAL and C. BROWN, *Dynamic Bayes Net Approach to Multimodal Sensor Fusion*, in Proceedings of the SPIE - The International Society for Optical Engineering, 1997.

[53] S.KLEMMER, J.LI, J.LIN, and J.A.LANDAY, *Papier-Mâché: Toolkit Support for Tangible Input*, in CHI Letters, Human Factors in Computing Systems: CHI2004, 2004.

[54] B. ULLMER and H. ISHII, *The MetaDESK: Models and Prototypes for Tangible User Interfaces*, in ACM Symposium on User Interface Software and Technology, 1997, pp. 223–232.

[55] B. ULLMER and H. ISHII, *Emerging Frameworks for Tangible User Interfaces*, IBM Syst. J., 39 (2000).

[56] M. WEISER, *Hot Topics: Ubiquitous Computing*, IEEE Computer, (1993).

[57] J. WOEHLER, *Driver Development for TouchGlove Input Device for DWARF based Applications*. Systementwicklungsprojekt, Technische Universität München, 2003.

[58] W.REISIG, *Petri Nets, An Introduction*, EATCS Monographs on theoretical Computer science, 4 (1985).