Multimodal Interaction in Smart Environments: A Model-based Runtime System for Ubiquitous User Interfaces

vorgelegt von
Diplom-Informatiker
Marco Blumendorf

Von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
Dr.-Ing.

genehmigte Dissertation

Promotionsausschuß:
Vorsitzender: Prof. Dr. Volker Markl
Berichter: Prof. Dr. Sahin Albayrak
Berichter: Prof. Kris Luyten
Berichter: Prof. Jean Vanderdonckt

Tag der wissenschaftlichen Aussprache: 13.07.2009

Berlin 2009

D 83

Zusammenfassung

Die wachsende Verbreitung des Computers in allen Bereichen des Lebens birgt neue Herausforderungen für Wissenschaftler und Programmierer in verschiedensten Fachrichtungen der Informatik. Vernetzte Geräte bilden intelligente Umgebungen, die unterschiedlichste Geräte, Sensoren und Aktoren integrieren und leiten allmählich einen Paradigmenwechsel in Richtung des "Ubiquitous Computing" ein. Mit der wachsenden Durchdringung unserer Lebensbereiche durch Computer-Technologie, nimmt auch das Bedürfnis zu, die steigende Komplexität über neuartige Benutzerschnittstellen einerseits handhabbar zu machen und andererseits vor dem Nutzer zu verbergen. Diese Arbeit prägt den Begriff Ubiquitous User Interface (Allgegenwärtige Benutzerschnittstelle) um Schnittstellen zu bezeichnen, die einer Vielzahl von Nutzern erlauben mit verschiedenen Geräten über mehrere Modalitäten mit einem Satz von Diensten in wechselnden Situationen zu interagieren. Die Entwicklung und Bereitstellung solcher Benutzerschnittstellen stellt neue Anforderungen an Design und Laufzeit.

Der Einsatz von Modellen und Modellierungstechnologien ist ein vielversprechender Weg um der steigenden Komplexität von Software Herr zu werden. Diese Arbeit beschreibt einen modell-basierten Ansatz, der ausführbare Modelle von Benutzerschnittstellen mit einer Laufzeitarchitektur verbindet, um die wachsende Komplexität von Benutzerschnittstellen zu adressieren. Ausführbare Modelle identifizieren dabei die gemeinsamen Bausteine von dynamischen, in sich geschlossenen Modellen, die Design- und Laufzeitaspekte kombinieren. Die Überbrückung der Kluft zwischen Design- und Laufzeit innerhalb eines Modells ermöglicht die Heranziehung von Designinformationen für Laufzeitentscheidungen sowie Schlussfolgerungen über die Semantik von Interaktion und Präsentation. Basierend auf dem Konzept von ausführbaren Modellen wird ein Satz von Metamodellen eingeführt, der Designaspekte aktueller Benutzerschnittstellenbeschreibungssprachen aufgreift und zusätzliche Laufzeitaspekte wie Zustandsinformation und dynamisches Verhalten integriert. Die definierten Metamodelle umfassen dabei Kontext-, Dienst- und Aufgabenmodelle, ebenso wie abstrakte und konkrete Interaktionsmodelle. Sie ermöglichen die Definition der Elemente Allgegenwärtiger Benutzerschnittstellen auf verschiedenen Abstraktionsebenen. Beziehungen zwischen den Modellen ermöglichen den Austausch von Informationen zur Zustandssynchronisierung und den Datenaustausch zur Laufzeit.

Die Integration der Konzepte in die Multi-Access Service Platform, einer Architektur für die Interpretation von Benutzerschnittstellenmodellen, stellt einen neuartigen Ansatz zur Nutzung dieser Modelle zur Erstellung und Verwaltung Allgegenwärtiger Benutzerschnittstellen dar. Die Architektur bietet Unterstützung für die Anpassung der Präsen-

tation in Abhängigkeit der Geräteeigenschaften, multimodale Interaktion, Verteilung von Benutzerschnittstellen über mehrere Geräte und die dynamische Anpassung an Kontextinformationen. Die Integration zustandsbehafteter Benutzerschnittstellenmodelle mit der Welt außerhalb dieser Modelle wird durch die Projektion des Modellzustandes auf die Darstellung als Benutzerschnittstelle und die Stimulation von Zustandswechseln auf Basis von Benutzereingaben erreicht. Modelle für die Verteilung, multimodale Informationsverarbeitung (Fusion) und Adaption der Benutzerschnittstelle verbinden die äußere Welt mit der modellierten Benutzerschnittstellenbeschreibung. Verschiedenste Interaktionsgeräte werden unterstützt, um dem Nutzer den internen Zustand des Benutzerschnittstellenmodells, mit Hilfe von multimodaler Interaktion, über verschiedene Interaktionsressourcen zu präsentieren. Die Implementierung der Laufzeitarchitektur der Multi-Access Service Platform wurde als Teil des Service Centric Home Projektes in eine intelligente Heimumgebung integriert und diente als Plattform für die Implementierung verschiedener Applikationen für das intelligente Heim. Fallstudien wurden durchgeführt um die entwickelten Konzepte zu evaluieren. Die Umsetzung durch verschiedene ausführbare Modelle ermöglichte dabei die Kombination der Modelle in einem komplexen Netz zur Laufzeit und zeigte die Anwendbarkeit der entwickelten Lösung.

Abstract

The increasing popularity of computers in all areas of life raises new challenges for computer scientists and developers in all areas of computing technology. Networked resources form smart environments, which integrate devices and appliances with sensors and actors, and make an ongoing paradigm shift towards ubiquitous computing paradigms visible. With this growing pervasiveness of computing technology, their user interfaces need to transport and hide an increasing complexity. In this work the term Ubiquitous User Interface (UUI) is coined to denote user interfaces that support multiple users, using different devices to interact via multiple modalities with a set of applications in various contexts. The creation of such user interfaces raises new requirements for their development and runtime handling.

The utilization of models and modeling technologies is a promising approach to handle the increasing complexity of current software. This thesis describes a model-based approach that combines executable user interface models with a runtime architecture to handle UUIs. Executable models identify the common building blocks of dynamic, self-contained, models that integrate design-time and runtime aspects. Bridging the gap between design- and runtime models allows the utilization of design information for runtime decisions and reasoning about interaction and presentation semantics. Based on the concept of executable models a set of metamodels is introduced, that picks-up design-time features of current user interface description languages and integrates additional runtime aspects like state information and dynamic behavior. The defined metamodels range from context-, service- and task- to abstract- and concrete interaction model and aim at the definition of the aspects of UUIs on different levels of abstraction. Mappings between the models allow the exchange of information for state synchronization and data exchange.

The integration of the concepts into the Multi-Access Service Platform as an architecture for the interpretation of the models, provides a novel approach to utilize user interface models for the creation and handling of Ubiquitous User Interfaces at runtime. It provides components to support shaping according to interaction device specifics, multimodal interaction, user interface distribution and the dynamic adaptation of the user interface to context information. The integration of the stateful user interface models with the outside world is addressed by the projection of the model state to UUI presentations and the stimulation of state transitions within the models, based on user input. Integrating distribution, fusion and adaptation models bridges real-world needs and the modeled user interface definition. Various interaction devices are supported to convey the inter-

nal state of the user interface model via a multimodal presentation, distributed across multiple interaction resources. The implementation of the runtime architecture has been integrated into a smart home environment as part of the Service Centric Home project and served as implementation platform for different multimodal home applications. Case studies have been conducted, to evaluate the developed concepts. The realization of various executable models supported their combination into a complex net of models at runtime and allowed to prove the feasibility of the developed approach.

Acknowledgments

I would like to thank my adviser Prof. Sahin Albayrak for the great support and for giving me the opportunity to conduct my research in the stimulating environment of the DAI-Labor of the Technische Universität Berlin. You gave me the opportunity to work with and lead the HCI working group, providing the basis for my research. I would also like to thank Prof. Jean Vanderdonckt and Prof. Kris Luyten for their motivating support and feedback as members of the committee.

This work has been conducted as part of the research of the HCI working group and is based on collaborative work and a great amount of shared ideas. Many thanks also go to the great team forming that group; with every team member, namely Sebastian Feuerstack, Grzegorz Lehmann, Dirk Roscher, Veit Schwartze, Florian Weingarten, as well as Serge Bedime, being a hard worker, an excellent programmer and a remarkable scientist. It was a great pleasure to work with every single one of you during the last years. Additional thanks go to all the people I had the pleasure to work with at the DAI-Labor and that helped me by testing and implementing various ideas to support this thesis. Namely these are Andreas Rieger, Joos-Hendrik Böse, Nadine Farid, Daniel Freund, Alexander Nieswand, Daniel Käs, Maximilian Kern, Kalppana Sivakumaran, Tilman Wekel, Cornelius Wefelscheid, Tammo Winkler, and Mathias Wilhelm. I also want to thank the German Federal Ministry of Economics and Technology for funding the years of my research and thus provide me the opportunity to finish this work.

I want to thank my friends, for always supporting me during all the years of hard work, for listening to my ideas and not understanding a single word sometimes, for distracting me when I needed a break and for good times in Berlin and everywhere on the planet. Special thanks go to Katrin and Florian for all the fun weekends and nights out. Finally, I want to thank my family for always being there for me, giving me a stable basis I could always return to. You gave me the roots I needed to grow and the courage to finish what I started.

When completing a work of this size and duration, it is almost impossible not to forget to acknowledge someone. So a big thank you to everybody I forgot to mention.

Contents

1. Introduction

Computer systems are currently changing our lives and the way we handle technology. The growing number of computers, miniaturized and embedded into TVs, washing machines, on-board computers of cars, or the various generations of mobile devices, transform the computer from a business machine, dedicated to specific tasks in a well defined environment, to a universal problem solver in all areas of live. Interconnecting these systems drives information exchange and cooperation and blurs the boundaries of standalone devices. Connected systems in homes and offices form smart environments and integrate devices and appliances with sensors and actors. An ongoing paradigm shift towards ubiquitous computing concepts (Weiser, 1993) becomes observable as environments become smart and provide new and innovative applications.

These developments also lead to a change in human-computer interaction. Interaction within smart environments becomes highly distributed and situation dependent. It is driven by the utilization of multiple interaction devices, sequentially or simultaneously. Interaction takes place via a network of interconnected devices, instead of a single device. Multiple usage situations influence the interaction style or the usage of the applications and multimodal interaction gains importance within mobile usage and changing context situations. Different interaction styles and modalities (e.g. voice, pen-based input, gestures, or touch screens) support flexible interaction techniques. Additionally, users interact with a multiplicity of applications and affect virtual data as well as real world actors (e.g. controllable appliances). Sharing resources, data, and applications for collaboration becomes important for multi-user environments, but users also compete for restricted and limited resources. The resulting interaction complexity and multiplicity raises the need for integrated interfaces, that provide user tailored universal access to applications and services, supported by the available resources of the smart environment.

Handling this interaction complexity and the multiple dimensions of interactive applications for smart environments requires new user interface features, not supported by current runtime architectures. User interfaces that adjust to device capabilities or usage situation, that support multimodal interaction, and that can be distributed across mul-

tiple interaction devices, are complex to handle and pose new development challenges. Competing for resources, sharing applications between multiple users, integrating applications and merging user interfaces, even increase this complexity. Additionally, the changing conditions ((dis-)appearing user, devices, situations) make the dynamic adaptation of such user interfaces by adjusting and (re-)configuring the underlying features an important aspect.

This work addresses the runtime handling of *Ubiquitous User Interfaces (UUIs)* that combine shapeability, multimodality, distribution, shareability and mergeability as features to address these multiple dimensions of user interfaces for smart environments. While this raises the need for a User Interface Description Language (UIDL) to express the identified features at design-time, the creation of UUIs also raises the need to handle interaction between user and interactive system at runtime, which is the main focus of this thesis. A model-based approach is followed to address the user interface complexity, the integration of design- and runtime issues and the combination of runtime concepts with the UIDL. This aims at a central user interface model, that stores information about the user interface on different levels of abstraction. In contrast to most other model-based approaches the main aspect is not the development of models that allow the derivation of a final user interface description, but the definition of models that are the final user interface description. Utilizing these models at runtime provides an internal representation of the state of the computer system, that can be understood and used to mediate between human and computer. It allows the direct access to design decisions and thus the better interpretation of the meaning of the user interface for adaptation and configuration purposes. Underlying the user interface models is the concept of executable models, which aims at the utilization of the developed models and the exchange of information between them at runtime. A net of stateful models provides the possibility to observe the interaction state at any moment in time and to stimulate model elements to trigger state changes and complex chain reactions on different abstraction levels. The models are defined by a set of metamodels, reflecting the needs of UUIs. A common meta-metamodel reflects the meta concepts, making each of the models executable. The Multi-Access Service Platform incorporates the meta and meta-meta concepts to provide a runtime architecture for the generation of user interfaces and the interpretation of the interaction. It addresses the need to project the internal state of the modeled system to the outside world in form of a Ubiquitous User Interface. The architecture combines a set of core UI models with additional runtime specific models and incorporates means to derive user interfaces, manage and synchronize multiple parts of the user interface, interpret and fuse user input from multiple modalities and devices and incorporate the context

of use into the interaction. Focusing on features like context adaptation, distribution across and migration between multiple devices as well as support for multimodal interaction, different applications have been build and evaluated in two case studies. Main parts of this work have been conducted as part of the Service Centric Home (SerCHo, www.sercho.de) project, sponsored by the German government (BMWi). A major focus was thus the application of the results to realize innovative services and applications for smart home environments.

1.1. Goals and Contributions

Aiming at the development of a framework for the creation and runtime management of user interfaces for smart environments, UI development and runtime handling are two important aspects. While the UI development process has been described in (Feuerstack, 2008), this work aims at realizing a runtime system providing the means to handle user interfaces for smart environments. It focuses on the specification of the user interface metamodels according to runtime needs and the computer-based interpretation of conforming user interface models. In summary, the overall goal of this work can be described as the simplification of human-computer interaction regarding two perspectives:

- *user support:* The user is more and more focused in recent systems and should be the ultimate target of development efforts. The overall goal from this perspective is thus to provide optimal interaction capabilities for users at all times.
- *development support:* The system developer and UI designer face the challenge of an increased complexity of the interaction and the user interface. Thus there is the need to express the interaction possibilities while giving room to their actual representation and adaptation to multiple usage contexts and runtime aspects.

As sketched in figure 1.1, the Multi-Access Service Platform acts as runtime system, mediating between user and backend services. A main concept of the approach is the definition of the meta-metamodel of executable models. This provides the general elements to support the execution of models within the architecture of the system. It also allows the creation of a set of conforming metamodels that express the concepts of Ubiquitous User Interfaces and provide the core concepts of the architecture to create UUIs. Using these metamodels, the user interface developer is able to define a user interface model, comprising multiple models that conform to the different metamodels. These models describe the anticipated interaction and can be "loaded" into the runtime architecture to create a UUI. The user is then able to utilize the UUI to interact with

the system and the connected backend services within the smart environment. Main aspects of the architecture are the exploitation of the smart environment by delivery and distribution of the UI with continuous synchronization and management of the interfaces at runtime, the definition of multimodal interaction including the distribution of output across modalities and the fusion of multimodal input as well as the adaptation of the interaction to the current context of use. Based on this general approach, three main

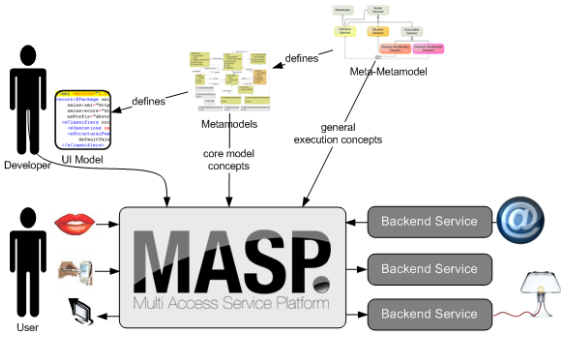

Figure 1.1.: The runtime system, mediating between user and backend services.

contributions of this work to build Ubiquitous User Interfaces can be identified and are described in the next sections:

- A **meta-metamodel,** identifies the common building blocks of user interface models to be used at runtime in form of executable models.
- A **set of metamodels**, conforming to the meta-metamodel, addresses the runtime needs of UUIs.
- An **architecture**, integrates the meta-metamodel concepts and the set of metamodels to provide the means to handle UI models and create UUIs at runtime.

The application of the approach has been empirically evaluated within two case studies, addressing the creation of Ubiquitous User Interfaces for smart home environments.

1.1.1. Executable Models

Models and domain specific languages have been identified as a promising approach to address the complexity of software systems and have also been applied to the domain of user interfaces recently. This work uses models for the description of user interfaces and extends this approach by making these models dynamic and executable. This allows to utilize them as central interaction definition in combination with the runtime architecture

and to describe interaction on multiple levels of abstraction in a flexible way. Underlying this approach is the definition of a meta-metamodel, identifying the common building blocks of executable models. This provides the foundations to create metamodels to describe Ubiquitous User Interfaces and express their static and dynamic aspects at runtime. The meta-metamodel combines static definition elements with their semantic meaning in form of executable elements and situation elements as explicit execution state. At runtime this allows the inspection and manipulation of the current state of the model and its execution. Based on these building blocks, multiple metamodels and generic links between them can be defined. The executable models concept is described in more detail in chapter 4.

1.1.2. Reference Metamodels

Utilizing the meta-metamodel of executable models to create UUIs and especially executable UUI descriptions additionally requires its application to the next lower level of abstraction. This leads to the definition of a set of metamodels, applying the concept of executable models to the domain of Ubiquitous User Interfaces. The set of metamodels aims at picking up findings from other model-based approaches, but puts a strong focus on the issues arising during the runtime interpretation of user interface models. Modeling UUIs, main aspects of the metamodels have been shapeability and support for distribution across multiple devices and modalities sequentially and simultaneously. Besides the adaptation at runtime, multimodality has played a major role, supported by the expression of interaction possibilities on an abstract, modality independent- and a concrete, modality dependent level to facilitate the dynamic combination of modalities. The reference metamodels, that comprise task-, domain-, service- as well as abstract interaction-, concrete input and concrete output model, are described in detail in chapter 5.

1.1.3. A Runtime Architecture

The third contribution of this work is the combination of the concept of executable models and the developed metamodels with an architecture to build a runtime system, handling user interface descriptions to create Ubiquitous User Interfaces. This has been addressed by building the Multi-Access Service Platform (MASP) that supports the utilization of the defined models to derive user interfaces and define their behavior over time. Utilizing the same models at design-time and at runtime makes design decisions explicit at runtime and provides meaning to the different user interface elements. Additional aspects, specific

for the combination of the user interface metamodels with the architecture, like the handling of input fusion, user interface distribution or context adaptation are addressed. The MASP is described in more detail in 6.

1.2. Thesis Structure

Illustrating the described contributions, this thesis is structured as follows.

Chapter 2 introduces the basic concepts of smart environments and illustrates the idea of Ubiquitous User Interfaces in detail. It describes five underlying features - shapeability, distribution, multimodality, shareability and mergeability - and derives the main building blocks, needed for their realization at runtime.

Chapter 3 illustrates the current state of the art in the related areas, providing a general overview of model-based approaches and multimodal systems. It also describes selected research issues in this area in greater depth and identifies shortcomings and requirements.

Chapter 4 introduces the meta-metamodel of executable models.

Chapter 5 describes a set of reference metamodels for user interface development and emphasizes the runtime aspects of these models.

Chapter 6 introduces the Multi-Access Service Platform, combining the concept of executable models and the introduced reference models with an architecture.

Chapter 7 presents the validation of the approach by two case studies and the evaluation of the concepts against the original requirements.

Chapter 8 concludes this work with a summary and outlook on future work.

Details about the conducted case studies can be found in appendix A and B.

2. Ubiquitous User Interfaces for Smart Environments

The advent of ubiquitous access to various networks and the Internet from any kind of device as well as the direct interconnection between different devices in smart environments raise new needs for the interaction with remote services and distributed systems. This chapter introduces smart environments and the challenges the development of applications for smart environments raises. From these challenges a set of features (shapeability, distribution, multimodality, shareability and mergeability) are identified and presented in section 2.3. The term Ubiquitous User Interface (UUI) is coined for user interfaces incorporating these features.

2.1. Smart Environments

Smart environments are characterized by the availability of numerous devices and appliances that are interconnected and thus able to exchange information with each other. They integrate sensors to monitor context information, usually host multiple services and applications and support interaction with multiple users. Figure 2.1 shows a sketch of a smart home environment build at the DAI-Labor of the Technische Universität of Berlin in cooperation with the Deutsche Telekom Laboratories. The figure shows four rooms equipped with multiple networked resources: interaction devices, sensors and controllable appliances as actors. The connection of these resources forms a complex system offering new possibilities for services and interaction. Considering the set of networked resources and the basic structure of smart environments, some issues and challenges for the creation of applications can be identified. Real-time and real life issues like continuous availability, extensibility, resource efficiency, safety, security or privacy (Nehmer et al., 2006; Becker, 2008) are major challenges for such systems. However, interactive applications for such environments face additional challenges. In contrast to PC-based systems, personal devices and applications, interactive systems and applications embedded in the

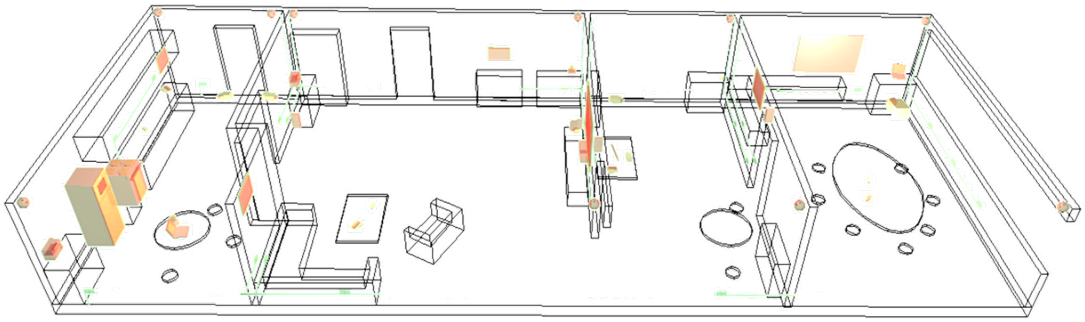

Figure 2.1.: A smart home environment with various networked devices.

environment need to address **multiple users** and various user groups with different skills and preferences. Such systems are used in scenarios much less predictable than the usual "user in front of a PC" usage schema (Abascal et al., 2008). Special needs of different target groups like supportive usage, non disruptiveness, invisibility, low acceptance for technical problems and the involvement in the active everyday life (Abascal et al., 2008; Weber et al., 2005; Becker, 2008) have to be considered carefully. While personalization puts a strong focus on the user as the main actor for any kind of system, context of use adaptivity goes one step further and comprises adaptation to **multiple situations**, including user, platform and environment at runtime. The complexity of such adaptive systems is massively increased by the distributedness of smart environments. Combining **multiple applications** requires the close integration and data exchange between these applications to create the image of a single integrated system for the user. The availability of multiple resources (interaction devices, sensors and appliances) raises the need to utilize different resources for interaction, making the adaptation to the different capabilities or even different modalities an essential issue. The interactive systems must be capable of dealing in real-time with the distribution of input and output to **multiple devices** in the environment to provide humans with continuous, flexible, and coherent communication (Emiliani and Stephanidis, 2005). The distribution of interaction across multiple interaction devices sequentially can provide a richer interaction by addressing the fact that the user moves around in the environment during interaction. Using multiple interaction devices simultaneously takes into account the appropriateness of a combination of resources for a given task over the utilization of a single device. The combination of multiple different interaction devices also leads to the usage of **multiple modalities** and interaction paradigms. Interaction shifts from an explicit paradigm, in which the user's attention is on computing, to an implicit paradigm, in which interfaces

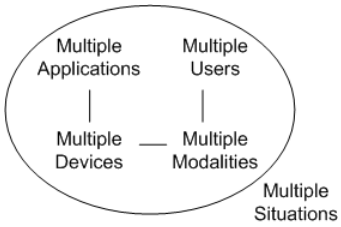

Figure 2.2.: Multiplicity in smart environments: Multiple users use multiple modalities to interact via multiple devices with multiple applications in multiple situations.

themselves proactively drive human attention when required (Emiliani and Stephanidis, 2005). Multimodal interaction can provide greater robustness of the system and natural communication between user and system by voice and gestures can enhance usability, especially if keyboard and mouse are not available or suitable to use. Distributed multimodal interaction requires the ability to consider an unpredictable number of devices and device federations ranging from mobile phones or headsets to multi-touch wall displays and needs to address the lack of a dominating interaction paradigm in smart environments. Looking at all these challenges, a main factor is the overall interaction experience, which has to be excellent, so that users like the vision of being surrounded by computers, which is usually not the case with today's graphical user interfaces (Mühlhäuser, 2007). It is required to establish an appropriate balance between automation and human control (Emiliani and Stephanidis, 2005). While it is sometimes appreciated, that the system learns human behavior patterns, human intervention directing and modifying the behavior of the environment should always be possible (Emiliani and Stephanidis, 2005; Mühlhäuser, 2007).

From this general analysis of the properties and overall challenges of smart environments, a set of five dimensions, that affect the development of user interfaces for smart environments, is selected and covered by this work. The selected dimensions are illustrated in the next section.

2.2. User Interfaces for Smart Environments

The five identified dimensions that have to be covered by user interfaces for smart environments can be summarized as: multiple users using multiple modalities to interact via multiple devices with multiple applications in multiple situations. As illustrated in figure 2.2, this leads to configurations with multiple dimensions.

Multiple Users denote the support for different users using the system simultaneously or sequentially. To make use of this, shared interaction, shared resources and shared information are crucial aspects. While distinguished personal interaction spaces and privacy are still necessary, collaboration between users should also be a main focus.

Multiple Modalities identify the need to support different interaction styles in different situations. In contrast to current PC-based systems, smart environments do not provide a dominating interaction paradigm and users are often busy with a primary task, while seeking support by the computer. Multimodality can also help realizing more robust and natural interaction.

Multiple Devices reflect the need to support multiple interaction resources and devices. This addresses the need to change the used interaction devices sequentially, e.g. while changing rooms, as well as the need to utilize multiple devices at the same time. The latter can e.g. support multimodality or collaborative work. The flexible and dynamic combination of multiple interaction devices lowers the boundaries and limitations that each of the devices has.

Multiple Applications address the fact that a smart environment comprises multiple interactive applications that are utilized by the different users. While there is the need to separate these applications to avoid side-effects, there is also the need to integrate the different applications to exchange information and provide the view of a single comprehensive system.

Multiple Situations reflect the incorporation of context information into the interaction and thus denotes the fact that an application can be used under different circumstances. In some cases interaction optimized for one situation can not be performed in another (gesturing while cutting onions, speech commanding while on the telephone) and thus e.g. a different interaction paradigm might be more appropriate.

The complexity of these dimensions is vastly increased by the runtime dynamics in smart environments. Users and devices can enter and leave the scene, situation and context parameters change at all times, modalities can become (un-)suitable, new applications can be installed or old ones removed. Reflecting these alterations in the application and its user interface is usually referred to as (runtime) adaptation (Vanderdonckt et al., 2007). In the following section, a set of features addressing the identified dimensions is derived and user interfaces incorporating these features are defined as Ubiquitous User Interfaces.

2.3. Ubiquitous User Interfaces

Based on the selected five dimensions, five features, defining the design space of user interfaces for smart environments, are described in the following. The features comprise

- **shapeability** to address different layouts for users, device capabilities and usage contexts,
- **distribution** across multiple interaction devices,
- **multimodality**, to support various input and output modalities,
- **shareability** between multiple users,
- **mergability** and interoperability of different applications.

They can on the one hand be used as general framework to evaluate the feasibility of applications for their utilization in smart environments; on the other hand they guide the development of architectures and integrated systems for smart environment. From a design-time perspective the features identify static properties of the developed user interfaces. However, more important for this work is their applicability as dimensions for the runtime adaptation of user interfaces. In this latter case, the dynamic alteration of the configuration of the features as interaction parameters allows the dynamic adaptation of the application if usage situation and context change at runtime.

Before the features are described and defined in detail in the following sections, some basic terms and general concepts have to be defined.

2.3.1. Basic Terms

This section introduces some basic terms to ensure the common understanding of the definitions and explanations in the reminder of this work.

Interaction Resource

Interaction Resources (IRs) are defined in (Vandervelpen and Coninx, 2004) as "atomic I/O channels that are available and that a user can exploit for executing a task. In this context atomic means the I/O channel is "one-way" and limited to a single modality". Examples for IRs are keyboards, mice, screens, speakers, microphones, or cameras. A user interface spanning multiple interaction resources is distributed, if it also spans multiple modalities, it would be considered multimodal. An Interaction Resource is usually part

of an interaction device, that often provides a higher level of abstraction and allows the utilization of the interaction resources for the interaction with an interactive application.

Interaction Device

Interaction Devices (IDs) are defined in (Vandervelpen and Coninx, 2004) as "computing systems that handle the input of or send output to individual IRs that are connected to it. In other words, an ID is a collection of IRs together with the computing unit." An interaction device usually comprises the hardware used for the interaction (e.g. screen, keyboard, touch-pad) as well as a software platform for communication and presentation tasks. Additionally, it either provides local applications or the capability to connect to a remote system for application usage. Examples for interaction devices include desktop computers, laptops, PDAs, or mobile phones.

Platform

The term Platform has been defined as "the set of variables that characterize the computational device(s) used for interacting with the system. Typically, memory size, network bandwidth, screen size, etc. are determining factors" in (Calvary et al., 2001a). In this work platform refers to the combination of hardware and software of an interaction device, used to interact with the system. From the user interface point of view especially interesting are the available interaction resources and the software, providing the rendering capabilities and the input possibilities for the interactive application. Examples for a platform would be a PC running a web browser, a PDA, running Java, or a mobile phone, running a mobile browser.

Context of Use

The context of use of an interactive system is defined in (Calvary et al., 2003) as a combination of three entities:

- the users of the system, who are intended to use (and/or who effectively use) the system,
- the hardware and software platform(s), that is, the computational and interaction device(s) that can be used (and/or are used, effectively) for interacting with the system,

- the physical environment where the interaction can take place (and/or takes place in practice).

Configuration

The term Configuration (or user interface configuration) will be used in the following, to denote the current characteristics of a user interface. A user interface can have one or multiple static configurations to support multiple contexts of use. Assuming the changing contexts of use of smart environments, adaptation to these contexts is expressed through the dynamic alteration of the user interface configuration at runtime, which in turn leads to an alteration of the perceivable user interface. Altering a configuration at runtime is referred to as reconfiguration and can be system or user initiated.

Based on these definitions, the following sections describe the identified five features of UUIs and their dynamic aspects in greater detail.

2.3.2. Shapeability

Shaping the user interface (also referred to as remolding (Vanderdonckt et al., 2007)) adjusts appearance and presentation to different contexts of use. It puts a main focus on the consideration of platform capabilities, but also comprises the consideration of user capabilities and preferences and can take information about the environment into account. Figure 2.3 shows an example, where the size of user interface elements changes with the distance of the user to the screen.

A user interface is shapeable if different variants of it can be produced, reflecting different context situations. Adjusting graphical user interfaces (also referred to as layouting within this work) can e.g. be performed in terms of orientation, size, containment, order, and style of the user interface elements and usually has the goal to enhance the usability of the user interface. Adjusting a vocal user interface can happen in terms of temporal arrangement or intonation. In the area of user interfaces for smart environments shaping addresses a single interaction resource at a time, but can reflect user interface distribution or multiple used modalities. Altering the shape of a user interface at runtime is required to reflect context parameters like user preferences, platform properties or the environment changing at runtime. However, alterations of the shape of a user interface have to be applied very carefully to ensure consistency and avoid confusion of the user interface.

Definition 1.1: *Shapeability identifies the capability of a user interface to provide multiple representations suitable for different contexts of use on a single interaction resource.*

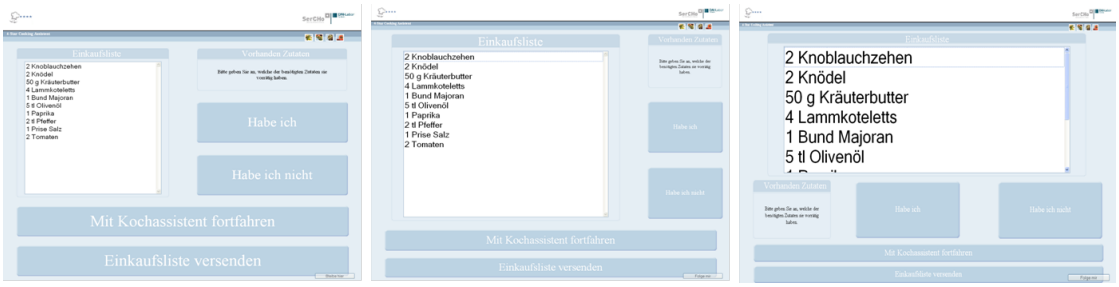

Figure 2.3.: Shaping Example: The size of the output elements is increased with the distance of the user to the screen, the size of the input element is reduced.

2.3.3. Distribution

The availability of multiple (networked) interaction resources in smart environments provides the possibility to simultaneously exploit them for interaction. Partitioning the user interface across multiple IRs increases communication bandwidth and allows the utilization of the different features of each IR. Directly addressing interaction resources like keyboard, mouse, speaker or screen, distribution e.g. allows their recombination (e.g. controlling one computer with a keyboard connected to a different computer). Replicating the complete UI or parts of it on multiple resources realizes redundancy within the interaction. Additionally, the utilization of different IRs allows the customization of the interaction dependent on the suitability of the IRs to carry out specific tasks. Figure 2.4 shows two configurations of a user interface distributed across a fixed screen and a mobile device.

Adjusting the distribution of a user interface at runtime pays tribute to the dynamics of interaction resources in smart environments. Mobile devices, appearing and disappearing in the environment, users entering and leaving rooms and moving around in the environments as well as multiple applications available to users, make the alteration of the used IRs and thus the redistribution of the UI an important aspect. The dynamic alteration of the distribution includes the (re-)partition of the user interface to multiple interaction resources, the replication of (parts of) the user interface on multiple resources as well the dynamic migration of (parts of) a user interface from one device to another one.

Definition 1.2: *Distribution identifies the capability of a user interface to present information simultaneously on multiple interaction resources, connected to different interaction devices.*

Figure 2.4.: Distribution Example: The user interface can be distributed across multiple interaction devices and is continuously synchronized.

2.3.4. Multimodality

The capability of the user interface to support not only multiple interaction resources but also multiple modalities can provide more natural and robust interaction capabilities to human users. A main goal of multimodal user interfaces is to provide the most suitable interaction modalities for the current task in the given context of use to optimally support the user. It provides great potential to enhance interaction e.g. in hands-free scenarios or applications that go beyond the mouse/keyboard interaction of standard desktop situations as shown in figure 2.5. Modalities can thereby be used one after another according to the current task and usage context or simultaneously to increase the communication bandwidth and expressiveness between user and system. While cross-modal systems support multiple usage variants via different modalities, multimodal systems support the simultaneous usage of multiple modalities.

The alteration of the used modalities at runtime allows a very flexible usage of multiple modalities by turning modalities on and off and adding and removing interaction capabilities at any time. This reflects context changes, making e.g. one modality useless like an increased noise level, increased distance to a touch-screen or having the hands busy while the system expects gestures. In combination with the capability to distribute the user interfaces across devices, modality (re-)configuration can be addressed through cross-modal or multimodal distribution.

Definition 1.3: *Multimodality identifies the capability of the user interface to support more than one modality.* (A more detailed definition is given in 3.1.4.)

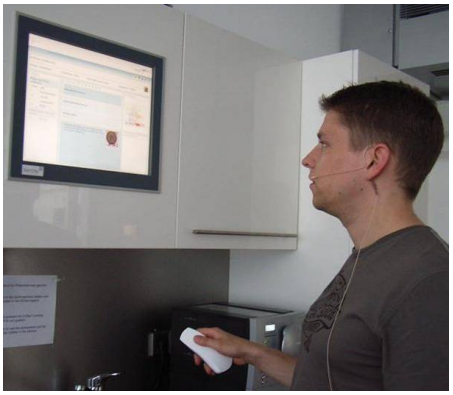

Figure 2.5.: Multimodal Interaction Example: The user is able to utilize multiple interaction resources and modalities including voice, touch and gesture simultaneously.

2.3.5. Shareability

The simultaneous presence of multiple users in smart environments leads to a demand for the shared and collaborative usage of information, applications and resources (e.g. light and temperature control of a room, collaborative work or shared resource planning). Shareable applications can support the usage by multiple users either sequentially or simultaneously by providing (synchronized) common elements within shared user interfaces. This can be realized via personal input devices of different users being connected to a single application or via shared interaction resources, like e.g. multi-touch-screens. Figure 2.6 shows two users using a single application simultaneously. Users on different locations can also be using shared applications through the replication of content on multiple locations. In the latter case, each user still has an own private interaction and information space, but exposes selected information to other users.

At runtime, shareability denotes the fact that a system can be switched to a collaborative mode, allowing a user to share an application with another (possibly distant) user. Making this feature configurable gives the user (as well as the application) control over when to share which information with whom. This allows to improve cooperative usage of applications and pays tribute to the multi-user aspects of smart environments.

Definition 1.4: *Shareability denotes the capability of a user interface to be used by more than one user (simultaneously or sequential) while sharing (partial) application data and (partial) interaction state.*

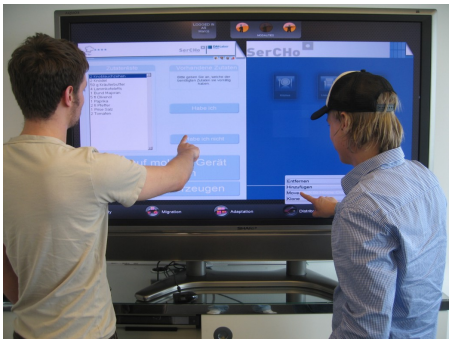

Figure 2.6.: Shareability Example. Two users sharing applications.

2.3.6. Mergeability

Mergeability identifies the capability to utilize one interaction resource to interact with multiple user interfaces simultaneously. To realize this, user interfaces can either split the available interaction bandwidth (e.g. split screen) or merge to provide an optimized usage. Merging can either happen through combination of the perceivable presentation i.e. shaping (e.g. in terms of the layout of the graphical user interface) or semantically where the underlying meaning is matched e.g. in terms of common tasks. Figure 2.7 shows an application user interface embedded into a Meta-UI controlling different parameters of the user interface. In combination with distribution capabilities, merging also allows to combine only parts of an application in a given modality or on a given IR. This is especially important when multiple applications are to be controlled via a modality that provides only a single channel (e.g. voice or gesture). In this case establishing another channel is impossible or impractical (e.g. using another microphone) thus, a virtual channel can be established e.g. by requiring prefixes to directing commands or commands need to be merged and otherwise adapted (e.g. commands like “App A: do X”, “App B: do Y”).

While merging applications at design-time allows the creation of new applications (e.g. mash-ups), which can be fine tuned by a developer, merging applications at runtime allows to use an unknown combination of applications simultaneously. In current PC-based systems this is done by overlapping or split screen and data exchange is performed via drag and drop or copy and paste. In smart environments, merging applications on a higher level of abstraction helps to better integrate multiple applications, which aims at better usability at runtime. Runtime merging also comes into play as applications in smart environments are dynamic and can appear and disappear (similar to users and

devices).

Definition 1.5: *Mergeability defines the capability of a user interface to be combined either partly or completely with another user interface to create combined views and input possibilities.*

Figure 2.7.: Mergeability Example: The user interface of a cooking assistant is embedded in the user interface of a meta user interface controlling different parameters of the interaction.

2.3.7. A Definition of UUIs

Based on the five described features, Ubiquitous User Interfaces address the challenges, that the realization of the ubiquitous computing in smart environments poses on user interfaces. However, besides the static aspects of the features - an application that can be executed in different variants or with different configurations - the dynamic aspects of the features are of great importance. This denotes the capability of the user interface to alter its feature configuration at runtime and switch seamlessly between different variations. This is also referred to as (runtime) adaptation and allows the dynamic adjustment of the application as usage situation and context change at runtime. Similar to Grolaux (2007) adaptation is defined as follows.

Definition 1.6: *Adaptation of user interfaces defines the alteration of the configuration of the user interface features at runtime, in order to adapt the presented user interface to changing needs.*

With these features, interaction possibilities are expanded to the multiplicity of users, modalities, devices, situations and applications. This also aims at bridging between the invisibility of technology and choice and control by the user through the user interface. Transparency of the complex environment is gained by conveying the system state to the

user and the removal of the boundaries of single devices. Based on the features and their runtime configuration, Ubiquitous User Interfaces are defined as follows.

Definition 1.7: *Ubiquitous User Interfaces are user interfaces that support the configuration of shapeability, distribution, multimodality, shareability, mergeability and the alteration of this configuration according to the context of use at runtime.*

2.4. Summary

In this chapter smart environments and the implications they pose on the usage of interactive applications have been analyzed with a special focus on their user interfaces. Five features (shapeability, distribution, multimodality, shareability, mergeability) have been presented to address the multiple dimensions (users, modalities, devices, applications and situations) of such user interfaces. They can be used to classify user interfaces for smart environments as well as to influence their development. The term Ubiquitous User Interfaces has been coined for user interfaces supporting these features. An important factor within its definition is the possibility to (re-)configure the features at runtime to dynamically adapt the user interface and the characterization of the features according to the covered runtime aspects.

While all five features are strongly interconnected, the ability to shape the user interface and the possibility to distribute it across interaction resources can be identified as basis for the application of the other features. While multimodality, shareability and mergeability provide important aspects, they build upon shapeability and distribution. The remainder of this work thus focuses on multi-device and multi-situation systems and thus the application and configuration of shapeability and distribution at runtime to adapt the user interface. Additionally, multimodality is considered, as a main factor facilitating flexible and robust interaction. Multi-user and multi-application scenarios are only briefly considered where suitable, as these aspects strongly rely on the availability of the more basic features. However, some information about how these are integrated into the approach are provided throughout the work.

Having defined Ubiquitous User Interfaces and their main features raises the need to describe such user interfaces at design-time and to handle interaction with them at runtime. The goal in this work is the creation of a runtime system, that allows the provisioning of UUIs for smart environments and that handles system output as well as user input. Based on the described features, it should be able to control the dialog flow and mediate

between the user and the system by providing shaped, distributed, multimodal interaction possibilities that adapt at runtime. Important related aspects are the handling of context information and the available interaction resources, as well as the utilization of these resources to provide Ubiquitous User Interfaces. Before the approach is described, the current state of the art is analyzed with respect to these aspects in the next section.

3. Fundamentals

In this chapter the state of the art is evaluated according to the goal to provide Ubiquitous User Interfaces for smart environments and the fundamentals to address this goal are discussed. Beginning with an illustration of the state of the art in user interface shaping, distribution, multimodality and adaptive interfaces, model-based development is described as an approach to handle user interface complexity and address user interface description languages to define the underlying user interface model. Afterwards selected user interface description languages as well as architectures and frameworks are discussed, to provide the desired flexibility for the next generation of user interface at runtime. The chapter concludes with a summary and identification of the shortcomings of the existing approaches.

3.1. Adaptive, Shapeable, Distributed & Multimodal Interaction

Multimodality as well as the distribution, shaping and the dynamic configuration of these features for the adaptation of user interfaces has been of increasing interest with the advance of research in ubiquitous computing and smart environments. In the following the current state of the art in these areas is analyzed, and common understandings of the problems, available approaches and open issues are discussed. Starting with the adaptation of UIs, the dynamic (re-)configuration of user interface features and the alteration of characteristics is discussed. Afterwards, shaping, distribution and multimodality are described as features that define a user interfaces and that can be configured at runtime for adaptation purposes.

3.1.1. Adaptation

Recently developers are confronted with the need to dynamically adapt their applications to changing conditions. This includes users (e.g. capabilities and preferences), device

capabilities and usage situations (e.g. on the move, at home). Within the application logic and the user interface, there is thus a strong need to reflect knowledge about the current context (user, device and environment).

Context has been specified by Dey and Abowd (Abowd et al., 1999) as 'any information that can be used to characterize the situation of entities (i.e. whether a person, place or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves'. The overall goal of the adaptation to context information is obviously not the adaptation itself, but the improving of the interaction and thus of usability, efficiency and effectiveness of the user interface (even at runtime). While context changes at runtime often require open adaptation to unknown contexts during a running session, closed adaptations can be defined and performed at design-time or application startup. The latter includes adaptations to predictive contexts that are known in advance like e.g. different user groups that individuals can be assigned to or a set of supported interaction devices. Depending on who performs the adaptation, different types can be distinguished:

- *system triggered self-adaptation*: the systems adapts itself according to the sensed context information
- *user triggered self-adaptation*: the systems adapts itself according to the sensed context information on request of the user
- *system triggered adaptation*: the systems initiates an adaptation process, involving the user to control the adaptation
- *user triggered adaptation*: the systems provides adaptation capabilities that are triggered and controlled by the user
- *configuration:* the user configures the system according to his needs

Context information relevant for the adaptation of a user interface is manifold. Park and Kwon (2007) proposed a generic context model that captures information about users, the environment, devices, and applications, Calvary et al. (2003) define context of use as a combination of user, platform and environment. Independent of the considered context information or the type of adaptation, any adaptation process can be structured in the following phases (see also Calvary et al., 2001a):

- sense the context information
- interpret the sensed information (detect and understand context changes)
- select or define a suitable adaptation strategy

- execute/apply the adaptation strategy

While the sensing of context information can usually be performed application independently the interpretation of the context information strongly depends on the needs of the application. Similarly, the selection of a suitable adaptation and the application of that adaptation heavily depend on the architecture of the underlying system and the needs of the application. Based on this process, adaptation can happen in two variants. On the one hand adaptation can "simply" (cosmetically) (re-)shape the perceivable user interface e.g. through rearrangement of interactive elements (surface restructuring). On the other hand adaptation can also be performed directly on the user interface description (functionally) altering the underlying concepts, according to adaptation rules defined by a third party (model restructuring). In any way, the defined interaction concepts provide an outline of the anticipated interaction and define the boundaries for possible adaptations. Different systems have been proposed to handle context information and to address software and user interface adaptation.

The Context Toolkit (Salber et al., 1999; Dey, 2000) introduced the notion of context widgets, which hide the specifics of devices and sensors to provide abstract context information and reusable building blocks. Encapsulating a state and behavior, the widgets can be queried by applications to receive context information. A context interpreter is used to create context information based on interpretation of available information and a context aggregator is used to collect and aggregate context information.

Huebscher and McCann (2005) present the Adaptive Middleware Framework for Context-Aware Applications, which abstracts from the raw sensor information using a 4-tier architecture. While the bottom layer consists of sensors, providing raw data, the second layer provides context providers, encapsulating this sensor information into context information. The third layer then allows the encapsulation of the context information by providing services hiding the particular context providers from the application on layer four. Applying this approach also allows the middleware itself to adapt to changing needs, e.g. by exchanging a context provider on layer three transparently for the application.

The Mobility and Adaptation Enabling Middleware (MADAM) (Mikalsen et al., 2006) comprises a context manager, an adaptation manager and a configurator to support the development of adaptive applications. Based on architecture models of applications the properties of each component are analyzed by the adaptation manager to identify relevant context information that it has to subscribe to at the context manager. Adaptation is supported at startup as well as at runtime (reconfiguration), by selecting the most feasible application variant according to the monitored context information.

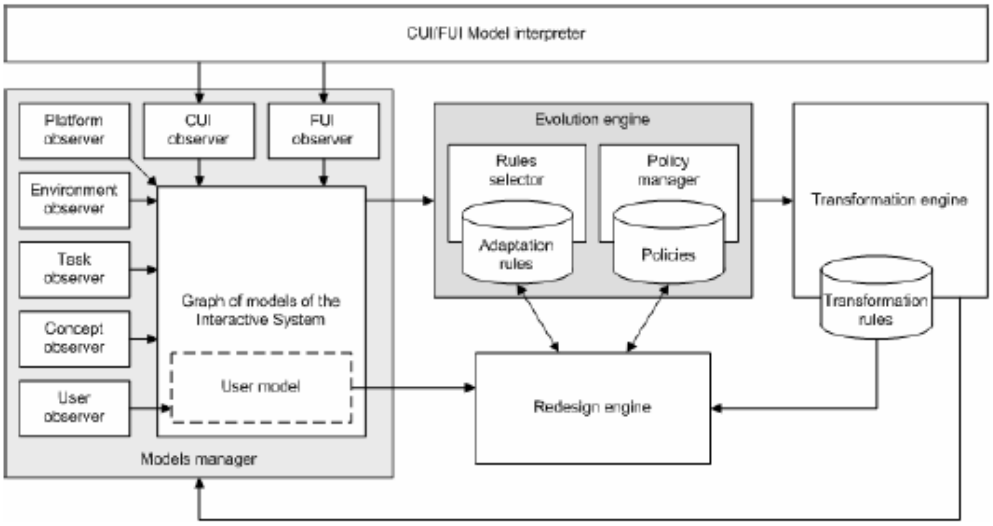

Figure 3.1.: Runtime infrastructure for open model-driven adaptation from Sottet et al. (2007b).

Rossi et al. (2005) present patterns providing different methods to apply for context adaptation. They mainly distinguish context objects, that encapsulate context information and perform the related adaptations and a rule-based approach, defining rules for fixed context information. Garlan (2004) presents the Rainbow framework that addresses adaptation by providing an external, reusable adaptation component. Similarly to MADAM it addresses the adaptation of an existing software architecture based on the architectural model of the application.

Focusing specifically on the adaptation of user interfaces, SUPPLE, presented by Gajos and Weld (2004) provides an optimization based approach to user interface adaptation according to the device capabilities and user information. The performed adaptation aims at computing an optimal layout and choosing a set of widgets for the rendering of an adapted user interface.

The Dynamo-AID runtime architecture (Clerckx et al., 2004) integrates a context control unit that utilizes abstract and concrete context objects to encapsulate context information. While abstract context objects can be directly linked to nodes of a task tree to affect the dialog flow, concrete context objects encapsulate the actual context information and are linked to the abstract objects. Based on the detected context information the system is able to change the dialog flow and recalculate the set of active tasks based on the available context information.

Sottet et al. (2007b) present a runtime infrastructure allowing the model-driven adaptation of user interfaces based on the models manager, the evolution engine, the transformation engine, the redesign engine and the CUI/FUI model interpreter. Figure 3.1 shows the components of the approach.

The model manager maintains the graph-based models at runtime and monitors changes e.g. new interaction resources or user input. It sends notifications to the redesign engine or the evolution engine in case of changes to the user model, to trigger adaptations of the UI. Adaptation is based on rules, consisting of a trigger event, a condition and an action part which can be defined in ATL (Atlas Transformation Language - http://www.eclipse.org/m2m/atl/). Each adaptation rule comprises a triggering event, a condition as well as an action to execute. Adaptation rules are defined as adaptation model, complying to a metamodel. The transformation engine then applies the selected transformation rules to the target models.

FAME provides a model-based Framework for Adaptive Multimodal Environments (Duarte and Carriço, 2006). An adaptation module comprises a platform and device model, an environment model, a user model and an interaction model and defines an adaptation engine and adaptation rules to provide adaptive multimodal user interfaces (more detail will be given in section 3.4.6). Duarte (2008) also identifies several problems of UI adaptations, that can confuse the user and lead to a drop of usability and efficiency of the user interface.

- **Hunting** refers to the fact that the user builds up a mental model of the system and has to adapt this mental model whenever the user interface of the application adapts.
- **Loss of Control** denotes the problem of adaptations that are not transparent and predictable for the user, which results in the feel of loosing control over the application.
- **Consistency** refers to the fact that adaptation should always produce similar results for similar contexts and that keeping the user interface constant might countervail the adaptation from the user's perspective.
- **Reliability** is required to ensure that no incorrect adaptations occur, which is likely to happen based on incorrect adaptation rules, wrong context information, or false user models.
- **Privacy** is an important factor, when monitored user behavior influences the user interface adaptation. Keeping information, that lead to the adaptation, private is

an important factor, especially if information from other users is also considered for the adaptation.

- **Persistence** refers to the fact, that performed adaptations have to be kept even through a system restart. This holds especially true for manual adaptations by the user.

Based on these problems several solutions have been proposed, including the active confirmation or initiation of adaptations by the user, the application of thresholds to reduce the number of adaptations, the limitation of adaptations during one session to a minimum and the scheduling of extensive adaptation in between two session, the minimization of the adaptation time, the direct influence of the user to system assumptions, context information and user data, leading to adaptations as well as the direct influence on the adaptation results. Additionally, the separation of fixed and adaptive parts and the consistency of adaptations can help to prevent the user interface from confusing the user more than supporting him or her.

While the described adaptation of user interfaces (and applications) affects all different aspects of the user interface and influences all identified features, the initially perceivable feedback is the alteration of the shape of the user interface, discussed in the next section.

3.1.2. Shaping

With the development of graphical user interfaces and the increasing diversity of computing resources, the idea of flexible user interface layouts began to rise. While user interface adaptation addresses the dynamic alteration of any aspects of the user interface at runtime, shaping addresses the possibility of the user interface to support multiple interaction devices.

While voice user interfaces can be shaped, according to temporal information, tone and intonation, the more obvious application of shape is for graphical interfaces. Shaping a graphical user interface refers to the possibility of the user interface to gain maximal usability on any interaction resource. Current approaches to realize this are layout managers, as they are used in Java Swing or AWT for example, as well as interpreted UI descriptions like HTML. Java layout managers, provide the means to define *Flow*, *Border*, *Grid*, *Gridbag*, *Card* or *Box* layouts and also to nest these layouts. The goal of the utilization of a these layout managers is to move and resize the UI elements on the screen to best fit the available screen space. Using HTML, the browser addresses the same problem, by moving around the defined elements to best fit the available space.

However, HTML does not provide a layout manager, but elements with defined properties, that remain constant and elements, that can be adjusted along their unspecified properties. HTML aims to separate shape and context via the utilization of Cascading Style Sheets (CSS).

An important aspect in the SUPPLE system Gajos and Weld (2004) was the computing of an optimal layout to render the user interface. It focuses on minimizing the user's effort when controlling the interface by adjusting size and position of elements on the screen. Constraints are used to describe device and interactor capabilities and influence their presentation.

Florins et al. (2006) apply graceful degradation to adjust user interfaces to the available screen space. Their work aims at the transfer of user interfaces designed for large screen to platforms with significantly less screen space. The underlying technique applies the pagination of interaction elements (e.g., windows, dialog boxes, web pages) based on a specification in the UsiXML user interface description language.

Other approaches focus on the calculation of a layout based on user interface constraints. A recent approach by Luyten et al. (2006b) focused on the implementation of a layouting algorithm based on the Cassowary algorithm (Badros et al., 2001), a weak constraint satisfaction algorithm. Hosobe (2001) presents the geometrical constraint solver Chorus, supporting "soft" constraints with hierarchical strengths or preferences. The same author later presents the DuPlex algorithm (Hosobe, 2005), that solves hybrid systems of linear constraints and one-way constraints and aims at the handling of Web document layout methods.

Richter (2006) proposes several criteria that need to be maintained to ensure consistency and usability when (re-)layouting a user interface. Nichols et al. (2002) also describe a set of requirements that need to be addressed in order to generate high-quality graphical and speech user interfaces. Based on these requirements, they present the Personal Universal Controller (PUC, http://www.pebbles.hcii.cmu.edu/puc/), aiming at the utilization of handheld devices to control appliances in smart environments. The PUC system aims at automatically generating multiple graphical as well as speech interfaces from the same user interface description. Similar approaches are also pursued by TERESA (Mori et al., 2004) and UsiXML (Limbourg et al., 2004b).

While shaping or layouting of user interfaces has been widely discussed and applied in different systems, supporting multiple interaction devices by the same user interface is still an open issue. However, it is the basic requirement to support multiple interaction

resources, possibly unknown at design-time, as well as unforeseen combinations of interaction resources. While the adaptation of the user interface shape at runtime is an important aspect and has great influence on the perception of the user interface and its usability, the shape can also be influenced by other aspects, like the combination of used interaction resources described as distribution in the next section.

3.1.3. Distribution

Distribution has its roots in the ubiquitous computing domain, realizing the fact that human-computer interaction in smart environments is likely to happen via multiple devices simultaneously and describes the process of splitting a user interface across multiple interaction resources. With the availability of multiple (networked) interaction devices with different capabilities, human-computer interaction is not longer bound to a single interaction device. Using a set of interaction devices simultaneously allows each device to convey a part of the user interface via its output resources (e.g. a screen or voice capabilities) and allow interaction via its input resources (e.g. keyboard, mouse, touch pad or voice input). Altering a distribution configuration at runtime is often referred to as migration (Berti et al., 2005) and describes the fact, that the user interface or a part of it move to another device. However, providing the capabilities for a device to expose its interaction resource (IRs) also allows the distribution (and migration) of user interface elements on an IR-basis. This allows the flexible assembly of the optimal set of interaction resources (even across multiple modalities). Such an independent addressing of resources also poses challenges to interaction devices and user interface components. UIs now have to have input and output separated and synchronized across the network. Within a smart environment such a distribution of a user interface can be realized within the following steps:

- discover & manage the available resources
- select a set of resources to distribute the UI to
- split the UI into the different segments and assign the segments to resources
- deliver the different segments to the selected resources
- receive input from input resources
- update output resources according to the system state

The discovery and management of interaction devices, their resources as well as their capabilities is an important prerequisite for the distribution or migration of user inter-

faces. Only if the system is aware of the available resources it can consider them in the distribution process. Basically two methods to include devices can be identified: (1) the active registration of the device either automatically or by a user or (2) the automatic discovery of the available resources e.g. via UPnP (Luyten et al., 2005). Once the available devices and their interaction resource are known, the system can calculate a distribution configuration, based on e.g. the resources used in previous interaction steps, the requirements of the current interaction step, the overall application state, context information or any other relevant data.

The user interface then has to be split to match the selected interaction resources (also known as fission). The main problem here is the partitioning of the user interface across interaction resources and modalities, without breaking the conveyed meaning. This requires to derive a presentation from semantic information by (1) content selection and structuring, (2) modality selection and the specification of the presentation details in each available modality and (3) spatial and temporal output coordination (Foster, 2002).

Content selection and structuring denote the definition of the overall goal of the communication act and the identification of the information to be presented to the user. Computational linguistics distinguishes at least three types of structure:

- intentional structure, relates the utterances to the desired effect,
- information structure, defines the semantic relationships between the information,
- attentional structure, denotes the information currently in the focus of the attention.

Additional types like the rhetorical and the information structure have also been discussed in the literature. Based on these types the presentable content can be structured for the presentation.

The selection of modalities to convey the information can be defined as the goal to find a set of modalities that effectively presents the given set of data in the current situation. It comprises the selection of the presentation modality for each information chunk as well as the specification of the presentation details according to the selected modality. The selection process can be influenced by characteristics of the available output modalities (see e.g. Bernsen, 1997b) and of the information to be presented, communicative goals of the system, user characteristics, current tasks of the user, and any limitations of the available interaction resources.

The coordination of the output finally comprises the task of arranging the presentation. This includes physical/spatial and temporal arrangement. Additionally, coordinating

the output includes the ability to reference information across modalities and also to reference the modalities themselves. The spatial arrangement of the presentation mainly addresses screen layout like the partitioning of the screen or the positioning of images in relation to textual information. Temporal coordination is required to provide coherent output especially in combination with speech or animations. This is also related with references across modalities ("as you can see on the screen now ..."). A problem here is the exact timing, as the duration of a speech output might e.g. not be determinable by the system. Referencing modalities and across modalities finally denotes the need to point the users attention to information presented in a different modality. This can e.g. be the highlighting of graphical information the system currently talks about or the referring to it ("Please select one from the highlighted list.").

Depending on the application and the fission process, application elements can either be presented each on a single resource or be duplicated (cloned) to multiple resources simultaneously. The following properties of a distribution can be distinguished:

Static and dynamic: While a static distribution defines the utilized interaction resources, a dynamic distribution changes at runtime and requires the migration of user interfaces between interaction resources.

Replication and migration: While replication refers to the duplication of parts or the complete user interface on another device, migration refers to the fact that user interfaces (or parts) are removed from the source device and transferred to the target device.

Complete and partial: Altering a distribution configuration at runtime, complete and partial migration is distinguished. Complete migration refers to the complete user interface presented on one device migrating to another device. Partial migration in contrast identifies the separation of multiple parts of the user interface and the migration of at least one of these parts to another device.

Clone and copy: In case of the duplication of (parts of) the user interface, a clone and a copy of the user interface can be distinguished. A clone completely imitates the behavior of the original user interface (parts) and is thus completely synchronized. A copy of the user interface presents the same elements, but an interaction with the copy would not affect the original user interface.

While several of the adaptation dimensions and the distribution aspects have been identified in the literature (Demeure et al., 2005; Berti et al., 2005; Molina et al., 2006), no general approach or common understanding has been achieved yet. In the following

some selected approaches, addressing the distribution of user interfaces, are introduced and some detail problems are illustrated.

Demeure et al. (2005) proposes a reference model for the classification of different types of distributed user interfaces and distinguishes mouldable, distributable, and migratable UIs. Based on the approach, the 4C reference framework is defined in (Demeure et al., 2008) and identifies four general dimensions for distributed UIs: computation (*what* is distributed?), communication (*when* is it distributed?), coordination (*who* is distributing?), and configuration (*from where and to where* is the distribution performed?). Molina et al. (2006) also present a definition of a (limited) problem space of distributed user interfaces, but with a focus on the prototyping of distributed interaction at runtime. Grolaux et al. (2005) presents detachable interfaces, Bandelloni and Paternò (2004) present migratable user interfaces and sketch a runtime system for their realization.

The DynAMITE (Dynamic Adaptive Multimodal IT-Ensembles) project (Kirste, 2004) aims at the development of software-infrastructures that use self-organizing ad-hoc appliance ensembles, based on the SODAPOP system (Encarnação and Kirste, 2005). It aims at the utilization of different devices and modalities that can be dynamically assembled to support user interaction at runtime. In the same context, Elting and Hellenschmidt (2004) present conflict resolution strategies when distributing output across graphical user interfaces, speech syntheses and virtual characters. A channel connects multiple output devices and can apply different strategies to support the distribution of output.

- the all-strategy forwards output to all connected devices supporting the type of output
- the random-strategy forwards output to only one of the devices
- the best-strategy selects the devices based on a utility value function
- the multimodal output coordination strategy applies a hierarchical AI planning approach to output distribution

The system models the IO devices, services and resources which allows the provisioning of its own properties by each entity (Ding et al., 2006). Devices provide input and output components (screen, speaker, mouse, ...), which can be either graphically or acoustic, and are classified according to Bernsen's taxonomy of unimodal input and output (Bernsen, 2001). Additionally, the communication goal and content are considered for the presentation strategy.

The I-AM (Interaction Abstract Machine) system (Barralon et al., 2007) presents a software infrastructure for distributed migratable user interfaces. It provides a middleware

for the dynamic coupling of interaction resources (screens, keyboards and mice) to form a unified interactive space. Interactive spaces in this context are assembled by coupling devices (private & public interaction resources) to access services within the global environment, which is an interesting aspect in contrast to other approaches. Coutaz et al. (2003) identify limiting factors for current tools and approaches as: the concept of windows over that of a physical surface, the poorly modeled geometrical relationships between surfaces, the limitations of the number of parallel supported input interaction resources, the limited focus on single workstations, and the absence of a dynamic discovery of interaction resources. I-AM supports a physical level, where an *IAMPlatformManager* manages the interaction resources that are connected to the platform it runs on and provides the basic means to access these. A logical level provides an abstraction of this available infrastructure to the application and an interactor level implements the basic graphical interaction concepts (windows & widgets) as *IAMInteractors*. *IAMInteractors* can then be created, moved, destroyed, etc. in the logical space which is then mapped to the physical resources.

Berti et al. (2005) present a taxonomy for migratory user interfaces comprising 11 dimensions (Activation Type, Type of Migration, Number/Combinations of Migration Modalities, Type of Interface Activated, Granularity of Adaptation, How the UI Is Adapted, The Impact of Migration on Tasks, Implementation Environment, Architecture, Usability Issues) of the migration process. They identify session persistence and user interfaces able to adapt to the changing context (device, user, and environment) as crucial requirements for their realization and describe a migration process involving two main entities: the interactive system model, defining the application and the context model, defining the outside world. Earlier approaches, focusing of the migration of interfaces between multiple devices are e.g. I-Land (Streitz et al., 1999) and Seescoa (Luyten et al., 2002). Web-based distribution has e.g. been described in (Vandervelpen et al., 2005).

3.1.4. Multimodal Interaction

Multimodal interaction aims at the utilization of multiple modalities to interact with the human user. Main goals of multimodal systems are to increase the available interaction bandwidth, to achieve an interaction closer to natural human-human communication and to increase the robustness of the interaction by using redundant and complementary information. The area of multimodal interaction is multidisciplinary, comprising (at least) signal processing, pattern recognition, computer vision, dialog management, agent architectures, knowledge representation strategies, psychology, user interface design, and

an understanding of how humans use their different senses and motor skills (Fang Chen, 2005). This multidisciplinarity makes the development of multimodal user interfaces a very complex problem, leading to various definitions of terms and approaches depending on the underlying perspective (i.e. user or system view).

While such an approach can help to increase the communication bandwidth and make communication more robust and natural, efficient multimodal interaction poses numerous challenges on computing systems. This section describes the main terms and concepts of the modality theory in the following and presents selected framework illustrating the current state of the art afterwards. Finally, fusion and fission are described as major aspects of multimodal user interfaces.

Modality Theory

The concepts of multimodal interaction are based on the understanding of the terms mode, media, modality and channel.

While mode and modality are often used synonymously, this work follows the definition given in (Stanciulescu, 2008), defining (communication) *mode* correspondent to the human senses (sight, hearing, smell, taste, and touch). The author accordingly defines four types of input communication modes: graphical, vocal, tactile and gesture and six output communication modes: graphical, vocal, tactile, olfactory, gustatory, and gesture. A communication mode thus determines the interaction type between the user and the system and refers to the communication channel used by the two interacting entities.

Media is defined in (Blattner and Glinert, 1996) as a physical device that allows storing, retrieving or communicating information. This includes input devices like mouse, keyboard, touchscreen or microphone, as well as output devices like screen or speakers. Additionally, entities storing information like CDs or DVDs are often referred to as media.

From a system perspective, the communication mode is determined by the employed physical interaction resources, allowing acquiring/transmitting information from/to the environment. In this sense, a *communication channel* defines "the temporal, virtual, or physical link that makes the exchange of information possible between communicating entities" Coutaz et al. (1993). It ties the (physical) media to a mode of utilization.

In this sense, this work also follows the definition of a *modality* given in (Nigay and Coutaz, 1993) and defines a modality as a tuple, comprising a device d and an interaction

language L: M = <d, L>. A modality thus combines a (physical) interaction device to acquire input or deliver output (e.g. keyboard, mouse, microphone, screen, speakers) with an interaction language, defining a set of symbols that convey meaning (e.g. natural language or a set of gestures).

Based on these definitions, the terms multimode, multimedia and multimodal can be defined as follows (see also Stanciulescu, 2008). A *multimode* system is a system, that relies on multiple modes of communication and thus addresses multiple human senses. A *multimedia* system involves multiple media for interaction and thus utilizes multiple interaction devices. A *multimodal* system finally supports multiple input or output modalities and thus combinations of interaction devices and interaction languages.

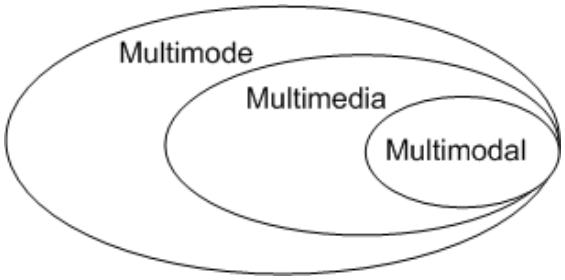

Figure 3.2.: Relation of multimode, multimedia and multimodal systems.

With these definitions, multimode includes multimedia and multimodal systems as a multimodal system is also a multimode system, because it exploits at least two different communication modes. A multimedia system also involves multiple modes and thus is also multimode. The main difference between a multimedia and a multimodal system is the capability of the multimodal system to also semantically process ("understand") the exchanged interaction data. Nigay and Coutaz (1993) thus also define multimodality as the "capacity of the system to communicate with a user along different types of communication channels and to extract and convey meaning automatically". Figure 3.2 shows the relation between the different systems. Additionally, Maes and Saraswat (2003) introduce the following terms to classify the use of multiple modalities, with respect to their temporal relation and usage.

- Sequential Multimodal Input, to identify the sequential usage of different modalities one after another.
- Simultaneous Multimodal Input, to identify the simultaneous usage of multiple modalities but their independent interpretation.

- Composite Multimodal Input, to identify the complementary usage of multiple modalities.

Multiple attempts to define the properties of modalities and devices can be found in the literature. The two most discussed are probably Bernsen (1994), that focuses on the definition of a taxonomy of output devices and Card et al. (1991), that defines a taxonomy of input devices.

Bernsen describes in his work a generative taxonomy of output modalities, distinguishing between representational modalities and the 'sensory modalities' of psychology (vision, hearing, smell, taste, and touch) to provide a theory generated from basic principles rather than based on empirical data. He identified three output media, namely graphics, acoustics and haptics that will be mainly used to represent information to humans or machines in a "physically realized intersubjective form". A modality is defined by properties allowing to distinguish modalities from one another: "linguistic/non-linguistic, analogue/non-analogue, arbitrary/non-arbitrary and static-dynamic", focusing on the types (or modalities) of information to be exchanged between user and system during task performance. Selection of appropriate modalities (as required for distribution or fission) is thereby based on the following variables: generic task, speech act type, user group, interaction mode, work environment, generic system, performance parameter, learning parameter and cognitive property (Bernsen, 1995, 1997a).

Card et al. (1991) classified input devices by the combination of linear/rotary, position/force, and absolute/relative. This results in a six tuple of manipulation operator, input domain, state of the device, a resolution function that maps from the input domain set to the output domain set, the output domain set, and a general purpose set of device properties that describe additional aspects of how a device works. The goal of this definition is to support the combination of individual input devices into complex input controls.

Nigay and Coutaz (1993) describe a design space for multimodal systems, defining both, the input and output attributes of an interface comprising three dimensions: Levels of Abstraction, Use of Modalities and Fusion. While the Level of Abstraction describes the different abstraction levels input and output are described at, when human and computer are exchanging data, Use of Modalities refers to the temporal relationship of the used modalities ranging from parallel usage of multiple modalities to sequential usage. Fusion finally describes the combination of data received from different modalities, resulting in the distinction between "Independent" and "Combined" modalities.

With these results as background information, different frameworks to understand multi-

modal interaction have been proposed. In the following two frameworks are introduced, that are relevant for this work and also gained much attention in the scientific community.

Theoretical Frameworks

Theoretical frameworks are used to understand the implications of the use of multiple modalities and to characterize the relations between the modalities. In the following the TYCOON Framework and the CARE Properties are introduced, which both provide similar means to characterize multimodal interaction.

The TYCOON Framework is a theoretical framework, based on the notion of TYpes and goals of COOperatioN between modalities (Martin, 1998). The framework comprises the notion of five basic types of cooperation between modalities and defines modality as a computational process.

1. **Transfer** describes the usage of one information chunk created in one modality by another modality, e.g. a mouse click that provokes the display of an image.
2. **Cooperation by equivalence** denotes the processing of an information chunk by one of several alternative modalities, e.g. the invocation of a command via speech or a mouse click.
3. **Specialization** means that a specific chunk of information is always processed by the same modality, e.g. input to text fields is always provided through the keyboard.
4. Modalities that **cooperate by redundancy**, process the same information by each modality, e.g. typing and uttering the same command.
5. **Complementary** denotes the processing of information by different modalities but not independently. This means the processed information has to be merged, e.g. uttering "put that there" in combination with pointing gestures.

The CARE properties (Coutaz et al., 1995), similarly to TYCOON, provide a framework for reasoning about multimodal interaction. The CARE properties comprise Complementarity, Assignment, Redundancy, and Equivalence as relationships.

1. **Complementarity** describes the supplemental processing of user input. This means an utterance in one modality can only be interpreted in conjunction with another utterance in another modality.
2. **Assignment** denotes the assignment of a specific modality to a given task or of a specific interaction device to a given modality.

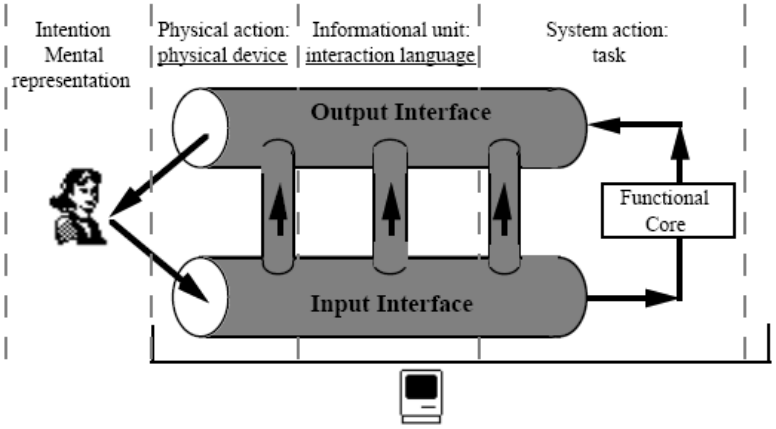

Figure 3.3.: Pipe-Lines Model from Nigay and Coutaz (1997).

3. **Redundancy** denotes the usage of multiple equivalent modalities simultaneously.

4. **Equivalence** of modalities defines that two modalities have the same expressiveness for a given task.

CARE Relationships are established between devices and interaction languages and between interaction languages and tasks, following the three levels (physical action in relation with a physical device, informational unit in relation with an interaction language, system action in relation with a task) of the Pipe-Lines model (Nigay and Coutaz, 1997) depicted in figure 3.3.

The ICARE (Interaction-CARE) system (Bouchet et al., 2004) provides a component-based approach for the development of multimodal interfaces and specifies elementary components, describing pure input modalities and composition components, allowing the combination of modalities according to the CARE properties.

While CARE and TYCOON define similar properties for the relation of multiple modalities in a multimodal system, some differences can be identified. Besides additional *transfer* property in TYCOON, it also focuses on the description of various types of cooperation among modalities and the view of a modality as a process that is analyzed to produce information. In contrast, CARE defines relationships among devices and interaction languages, interaction languages and tasks, or among different modalities and vies a modality as a tuple of device and language. Finally, TYCOON is completely defined from a system perspective, while CARE can be utilized from a user or a system perspective. Despite these differences, both frameworks describe the relations between multiple modalities and can be used to classify multimodal systems according to these

properties. Due to the widespread usage in the literature and the similarity of the two approaches the CARE properties will be used as classification scheme in the remainder of this work.

In the following fusion and fission are discussed, as basic technologies for handling multiple modalities and building multimodal user interfaces.

Fusion and Fission

Fusion and fission address the need to combine (fuse) multimodal user input to derive meaning and separate different building blocks to be presented via different modalities. While Fission is part of the distribution process, that has been discussed in section 3.1.3, a specific fission approach for multimodal systems has e.g. be realized in the SmartKom system (Wahlster, 2002; Müller et al., 2003). It is briefly introduced in the following to illustrate the concepts of fission from a multimodal perspective. Figure 3.4 shows the presentation pipeline of the system. A presentation planner receives a M3L encoded communication goal in a modality free representation and can be adapted via presentation parameters, e.g. to encode user preferences or capabilities of output devices. It decomposes the communication goal into primitive presentation tasks, according to context information and the current presentation parameters. Afterwards different output modalities are allocated to the presentation tasks and unimodal presentation goals are formulated accordingly. The system is able to synchronize speech output with the lip movement of an animated agent as well as deictic pointing gestures of the agent with graphical information.

Fusion technologies address the problem of combining user input received via multiple modalities to derive meaning and the user intention. One example would be the well known "put that there" approach by Bolt (1980). It combines speech and gesture input to e.g. select objects and move them between locations. The problem arises from the need to integrate continuous streams of signals from multiple modalities. These streams then need to be segmented and related events need to be combined. In contrast to these streams, common GUI based approaches create distinct events that can be interpreted directly. This paradigm also holds for GUIs distributed across multiple devices and thus fusion is mainly relevant for multimodal systems. The handling of contradictory input, often also handled in the fusion process, is relevant for all types of distributed systems, too.

Multimodal integration as part of modality fusion has been widely discussed in the liter-

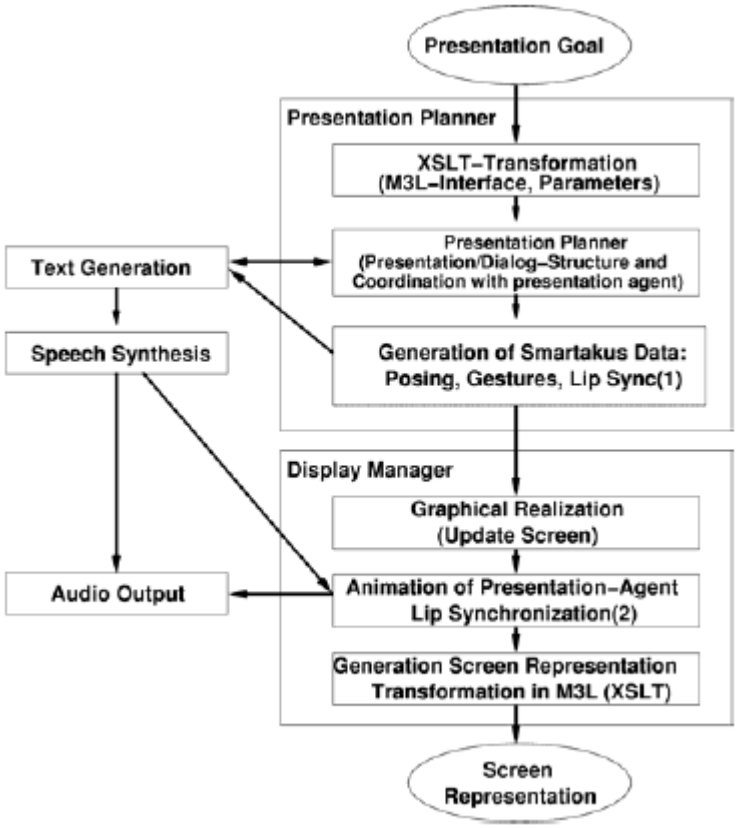

Figure 3.4.: The presentation pipeline of the SmartKom system (Reithinger et al., 2003).

ature. The reported systems utilize early fusion on the feature level to directly integrate the received signals without any semantic interpretation (Stork and Hennecke, 1996; Rubin et al., 1998) and late fusion to integrate the received signals on a higher semantic level (Bolt, 1980; Neal and Shapiro, 1988; Cohen et al., 1989; Wang, 1995; Cohen et al., 1997; Johnston et al., 2002; Holzapfel et al., 2004) and incorporate different integration mechanisms to realize input parsing and modality merging like melting pots (Nigay and Coutaz, 1995), Typed Feature Structures (Johnston et al., 1997), Unification Grammars (Johnston, 1998), Finite State Machines (Johnston and Bangalore, 2000, 2005), neural networks (Vo and Waibel, 1993; Waibel et al., 1995), rule-based approaches (Holzapfel et al., 2004), agent-based systems (Cohen et al., 1994), context-based fusion (Pfleger, 2004) as well as a biological-motivated approach (Coen, 2001). All approaches address the processing and understanding of natural speech input in combination with additional modalities e.g. gesture or pen-based input.

In **early fusion**, the recognition process of one modality usually influences or constrains the recognition process in another modality. This approach is considered more appropriate for temporally related modalities, such as speech and lip movement (Stork and Hennecke, 1996; Rubin et al., 1998). Early Fusion can e.g. be realized through the creation of feature vectors, combining the information and the utilization of hidden Markov models (HMM) for classification. However, the required training data and the close coupling to recognition technologies makes this approach less flexible than late fusion. Early Fusion also fails to handle imperfect input data and asynchronous input streams. Example systems based on early fusion can be found in (Bregler and Konig, 1994; Vo et al., 1995; Pavlovic, 1998).

Late fusion approaches use individual recognizers working unimodally and support the fusion of multiple recognition results on a semantic level. This allows the incorporation of available recognizers that can be trained with unimodal data. Late Fusion is better suited for modalities that are less coupled temporally or have different response times. It usually takes place in a two step process, integrating the different recognition results into a combined final representation first and deriving the meaning from the combined representation in the second step. The interpretation is based on the partial meaning of the different information chunks and a combined common meaning. Late fusion thus requires a common representation of meaning for all modalities.

In contrast to early fusion approaches, late (semantic) fusion approaches can more easily consider additional information like previously observed data or context information. Similar to early fusion approaches, time still plays an important role to determine the temporal relation of the chunks to each other and is thus often annotated to the chunks in form of a time stamp.

Early examples for late fusion can be found in (Bolt, 1980; Cohen et al., 1989, 1997; Neal and Shapiro, 1988; Wang, 1995). More recent approaches are presented e.g. in (Holzapfel et al., 2004; Johnston et al., 2002). Selected mechanisms for the required modality integration are described in the following and selected implementations are described afterwards.

Integration Mechanisms

A well known method to represent multimodal inputs is based on so-called *frames (Minsky, 1975)*. A frame represents objects and their relations, where each object consists of a set of attributes representing its characteristic properties. Frames can be organized in a net of nodes, where each node is a frame with attributes called slots. In multi-

modal applications, slots are filled with data, extracted from the user input. Delgado and Araki (2006) distinguish three types of frames: input frames, that hold the input of each modality independently, integration, integration frames, that are created by combining multiple input frames during the fusion process and output frames, that can be utilized to create system output. The filling of the slots of each frame can be carried out incrementally, as data is received. Additionally, slots can be derived from available information in other slots or filled with information from the previous interaction step.

Another approach to store the information representation are the so-called *melting pots* Nigay and Coutaz (1995). A melting pot collects the information received for a time slot and combines it with a time stamp. This allows the fusion based on time, context and the matching of the different melting pots. Three types of fusion are supported: microtemporal fusion combines chunks that are received in parallel; macrotemporal fusion combines chunks that are complementary and fit in the same temporal analysis window, contextual fusion combines chunks without considering time restrictions.

Feature Structures (Kay, 1984) and *Typed Feature Structures* (TFSs) (Carpenter, 1992) allow the representation of multimodal inputs using shared variables to indicate common structures. Typed feature structures are often used in natural language processing and aim at the unification of representation structures. TFSs support the specification of partial information chunks that are represented by sub-specified structures containing features that have not been instantiated and support unification by determining the consistency of two representation structures and their combination, if they are consistent. Typed feature structures have been used e.g. in Quickset (Johnston et al., 1997) to integrate input from speech and gesture recognition. *Multidimensional TFSs* have been used in (Denecke and Yang, 2000) to represent different aspects of multimodal input and also by Holzapfel and Fuegen (2002) to integrate emotions of the user and provide additional information like the used modality or confidence score for each information chunk. *Unification Grammars* are used to interpret typed feature structures and have been used e.g. by Johnston (1998). This allowed to state strategies for multimodal integration declaratively and to utilize a multidimensional chart parser to compose received inputs. *Finite State Machines* have been used by Johnston and Bangalore (2000, 2005) to parse multiple input streams and combine their content into a single semantic representation. In their approach a multimodal context-free grammar is used to specify possible interactions.

Other approaches use partial action frames and neural networks (Vo and Waibel, 1993; Waibel et al., 1995), rule-based approaches (Holzapfel et al., 2004) or agent-based systems (Cohen et al., 1994) to process the multimodal input. Coen (2001) presents a biological

view to the problem, arguing that there is a strong interconnection between our senses and thus handling multimodal integration should happen as early as possible and on multiple levels of abstraction. Portillo et al. (2006) introduce a hybrid-approach, combining multimodal grammars and a dialog-driven strategy. In (Pfleger, 2004), Pfleger introduces the PATE (Production rule system based on Activation and Typed feature structure Elements) system and emphasizes that multimodal fusion is also largely related to the context of the interaction and thus the current state of the dialog. Similarly, Chai et al. (2005) describe the interpretation of multimodal input according to the conversation-, domain- and visual context of the interaction.

Selected Approaches

The Quickset system (Cohen et al., 1997) uses a fusion technology based on typed feature structures to combine speech, gesture, and direct manipulation. It combines a set of continuous speech and gesture recognizers running in parallel and provides multimodal integration via a unification mechanism working on the typed features structures as semantic representation of the interaction. The approach has been continuously extended and redeveloped exploring different grammar-based technologies e.g. in the Match system (Johnston et al., 2002) and a statistical approach in (Wu et al., 1999).

In (Vo and Wood, 1996) an application framework for speech and pen gesture input fusion in multimodal learning interfaces is proposed. The system uses a frame merging algorithm with different modes to create partial filled frames and a continuous merging with scores. A poor recognition rate of ~80% was reported.

Holzapfel et al. (2004) present a system for multimodal human-robot interaction. The system is based on input events, represented as tokens which are transformed into semantic typed feature structures in a first step. Based on this feature structure an input set is created, providing the input for the fusion process. During fusion the tokens are read and the parser determines whether a subset of tokens can be merged. Different to other approaches, the order of the tokens is not critical and the approach supports the asynchronous retrieval of tokens. It is able to cope with processing delays by delaying the addition of new tokens until the parser finishes. Application specific fusion rules are used to configure the parser.

The SmartKom system realizes fusion based on adaptive confidence measures from recognizers that produce time-stamped and scored hypotheses (Wahlster, 2003). SmartKom provides a unification of all hypothesis graphs and the application of mutual constraints to reduce the ambiguity and uncertainty of the analysis results, similar to what has been

realized in the Quickset system. An additional discourse model is used to rank the fusion result within the context of previous interaction and to resolve deictic references.

Duarte proposes a Fusion process based on a Behavioral Matrix and related rules (Duarte and Carriço, 2006), Flippo et al. (2003) propose a Framework for Rapid Development of Multimodal Interfaces, supporting semantic fusion based on a parse trees generated by a natural language parser. The system produces information frames that can be processed by the dialog manager. In order to resolve references and ambiguities in the parse tree, the system incorporates resolving agents, operating on the parse tree, including resolvers for

- anaphoras and deictic references, finding referred screen objects and their locations,
- objects, finding an object according to the specified attributes
- spelling, returning spelled words
- dialog history, allowing the resolving of references based on the conversation history,
- coordinates, creates a (x, y) tuple from e.g. spoken input
- names, allowing string manipulations

The used agents are only aware of their local information and collaborate to create the complete information frame for a given parse tree. Additionally, a fusion manager combines data from multiple sources and spawns resolving agents.

TYCOON (Martin, 1998) integrates multiple modalities through the interpretation of sequences of detected events as commands that can be specified using a command language. A command specification like "*complementary_ coinc X SPEECH here MOUSE click **" allows merging the word here with a mouse click at any location in the same temporal window. Sequence interpretation is performed by a multimodal module, providing interconnected processing units, that process received signals according to the specified commands and the cooperation types.

ICARE provides generic composition components to implement the CARE properties, merge the data, and send it to the next linked CARE component, allowing a cascading of the components. Data merging is thereby performed on the task/component level of the system by the fusion mechanisms from (Nigay and Coutaz, 1995), utilizing the notion of buckets. A more detailed description can be found in section 3.4.3.

In conclusion, fusion technologies address the derivation of a common representation of meaning from the multimodal input resulting from multiple modalities creating different

information on the signal level as well as different (partial) interpretation results on the semantic level. The fusion of signals or (partially) interpreted information to derive the common meaning is the goal of the different fusion algorithms.

3.1.5. Summary

In this section the state of the art of adaptive, shaped, distributed and multimodal systems has been introduced and different approaches in these areas were presented. In summary, adaptive system are highly dependent on the notion of context and the provisioning of flexible user interface descriptions that allow dynamic adjustments of the final user interface. Shaping of user interfaces, graphical and vocal, requires the possibilities to express the relations between the elements and the boundaries for the adaptation by the system. Distribution of user interfaces is a complex problem, spanning support for multiple platforms and devices as well as support for the creation and synchronization of system output and user interfaces. Multimodal interaction requires means to semantically interpret multimodal input as well as to create multimodal output.

Interaction within smart environments provides the possibilities to combine and integrate these different aspects of user interfaces. Shaped user interfaces optimally exploit the capabilities of the used interaction resources. Multiple modalities support robust and natural interaction, and better insights into the user's intentions than traditional interfaces. Distribution can help to compensate weaknesses of interaction resources and modalities and, in combination with multimodal interaction and adaptation, enhance usability for diverse users. Supporting Ubiquitous User Interfaces in smart environments thus provides several advantages and has the potential to sustainably change the way we interact with computers and computer-based systems.

However, these aspects also raise new challenges, e.g. described in (Bourguet, 2004; Paternò, 2005; Vanderdonckt, 2005). One main problem of the provisioning of Ubiquitous User Interfaces is the orthogonality of the faced problems. While the abstract definition of UIs allows generic outlines that can be specialized (automatically), multimodal UIs require to combine various modalities that interact with each other to deliver meaning and functionality. Of major importance is thus to solve the problems of meaning, hidden within the usage of the different modalities, and the need to provide capabilities to make these meanings transparent and understandable for the computer. New user interface design, implementation, and runtime requirements burden the developer the task to define user interfaces, able to process (understand and generate) multimodal and distributed

interaction. Developing such interfaces does not seem to be possible by hand anymore today, but requires new methods of User Interface Engineering, based on the fundamentals of Software Engineering (SE), Human-Computer Interaction (HCI) and Human Factors (HF) (Vanderdonckt, 2005). One approach currently discussed in the scientific community is the development of model-based user interfaces, similar to successful model-based approaches in software engineering, which is described in the next section.

3.2. Model-Based Development

Model-Based Software Engineering (MBSE) has recently been successfully deployed as an approach to handle the increasing complexity of current software systems. It addresses the need to define reusable building blocks that abstract from the underlying system complexity and allow the developer to focus on the domain specific problem rather than to deal with implementation details. Underlying the approach is the idea, that a description of the problem domain can be formalized as a metamodel, allowing the creation of multiple models solving different domain specific problems. While the Unified Modeling Language (UML) made the idea of modeling popular by providing a common language to exchange concepts between developers, the Meta Object Facility (MOF) and Model-Driven Architectures (MDA) provide the key concepts for the utilization of model-based software engineering to derive running systems from software models. Technologies like Executable UML, UML Actions or the Business Process modeling Language (BPML) focus on the shift from static- to dynamic systems and executable models, allowing the direct utilization of the model to create executable code. While the original static models were mainly able to present snapshot views of the systems under study and could thus only provide answers to "what is" kinds of questions, dynamic models give access to information that changes over time and are thus also able to answer "what has been" or "what if" kinds of questions (see also Breton and Bézivin, 2001). Tools like the Eclipse Modeling Framework (EMF) or the Graphical Modeling Framework (GMF) provide the required infrastructure to create reusable models and building blocks.

In contrast to the field of MBSE, which is already wide spread in industrial software development, Models-Based User Interface Development (MBUID) is currently growing out of academia as an approach to deal with the increasing complexity of user interfaces. Managing the sequential or even simultaneous usage of multiple devices and modalities, adaptation to the context of use, multiple users with different preferences, etc. makes the already complex development of user interfaces even more complex. User interface

code already takes up to 50% percent of developed applications and this rate has very likely been increased since the study has been conducted by Myers and Rosson in 1992 (Myers and Rosson, 1992). Similar to MBSE, MBUID strives for a formalization of the domain of user interfaces to reduce complexity and provide reusable building blocks. It aims for the identification of the main aspects common for user interfaces and their abstraction in a user interface model to simplify the user interface development process by taking it from the code level to a higher level of abstraction. User Interface Description Languages provide basic information and concepts to express such models. Current model-based approaches support the developer by providing mechanisms to decompose complex applications into smaller manageable parts, modeling different aspects of the application independently. They usually combine a formal modeling language for user interfaces, an engineering methodology and tool support. Utilizing formal user interface models also takes the design process to a computer-processable level, that makes design decisions understandable for automatic systems. The main approaches can be separated in design-time approaches, providing development tools required to support the development of user interface models e.g. (Paternò and Santoro, 2002) or the tools related to the UsiXML Language (Vanderdonckt, 2005) and model interpretation at runtime (Clerckx et al., 2004; Balme et al., 2004; Sottet et al., 2006b). The latter gains more interest recently and aims at the possibility to exploit the information contained in the model for the adaptation of the user interface to multiple and changing contexts of use. All recent approaches aim at the integration of several models, describing different aspects of the user interface on different levels of abstraction in a common user interface model. Which models are required as basis for these approaches and which information is required in these models is still in discussion, although some basic types of models (tasks and concepts, abstract user interface, concrete user interface and final user interface) have recently been proven to be suitable descriptions of basic concepts (Calvary et al., 2003).

In the following a description of the basics of model-based development, introducing concepts like metamodels, transformations and mappings is provided. Afterwards the different levels of abstraction to consider within model-based user interface development are described. Finally, the utilization of user interface models at runtime is discussed.

3.2.1. Fundamental Concepts

Model-based development aims at the definition of Domain Specific Languages (DSL) allowing to form models of the addressed domain. Similar to the object-oriented pro-

gramming paradigm *"Everything is an Object"* in model-based systems the paradigm can be noted as *"Everything is a Model"*. The utilization of models for software development has gained great attention with the introduction of the Model Driven Architecture (MDA) by the Object Management Group (OMG). Its specification classifies Computing Independent Model (CIM), Platform Independent Models (PIM) and Platform Specific Models (PSM), aiming at the provisioning of a reification process. Starting with the colloquial description of the system in the CIM, a definition of abstract PIMs is derived. These are reified into more concrete PSMs using model transformations. The process finally leads to the creation of platform specific executable code.

In this sense, MDA (Miller and Mukerji, 2001) defines a model as *a representation of part of the function, structure and/or behavior of a system.* The system is in this case also noted as the system under study and the model aims at providing answers about questions concerning the system without having to study it directly. In this sense, *descriptive* and *prescriptive* models can be distinguished, with the former describing an existing system and the latter a system to be built. Additionally, *dynamic* and *static* models can be defined as models that do or do not change over time (Bézivin, 2005). Static models thus provide (only) a snapshot of the system under study at a given point in time, while dynamic models can also describe how a system evolves.

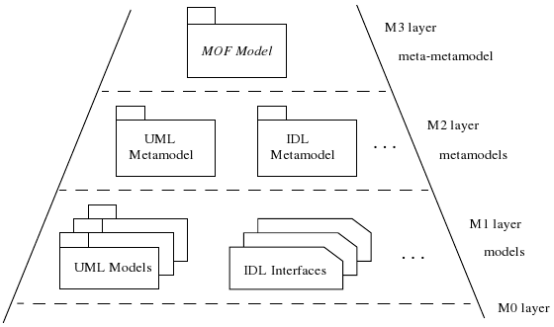

Figure 3.5.: The MOF Metadata Architecture (Obj, 2002).

An important concept in the domain of models is the concept of a metamodel, defined by the MDA specification as *a model of models.* In this sense a metamodel specifies *a model that defines the language for expressing a model* as defined by the Meta Object Facility (MOF) specification (Obj, 2002). In this context, MOF specifies the metadata architecture depicted in figure 3.5. The architecture defines four layers (also called levels) of abstraction. The lowest layer M0 identifies the system under study. This is described by models on the M1 layer. The next higher layer M2 then identifies the metamodels,

describing the syntax and semantics of the M1 models. The top layer M3 identifies the meta-metamodel, which is the model of all metamodels.

In systems comprising multiple models, relations between models can be either specified as mappings or transformations. *Mappings* express the relationship between two or more models and are defined as *a set of rules and techniques used to modify one model in order to get another model* (Miller and Mukerji, 2001). In this sense mappings are also said to perform transformations between PIM and PSM. *Transformations* are thus used to transform one or more models into new models, which refine the aspects of the original models. The difference between mappings and transformations can be seen as the former being an extension of the latter. When transformations create a model, the created model is not bound to the source model anymore. A mapping between two models can also be continuously maintained and thus relate source and target. In this work the term mapping is mainly utilized to describe links between existing models, while the term transformation is used to identify the creation of a new model from existing information.

Based on these understandings of the fundamentals of model-based software development the next sections introduce fundamentals and state of the art concepts of model-based user interface development.

3.2.2. Levels of Abstraction

Earlier approaches in model-based user interface design and development discussed a broad variety of possible models and abstraction levels, user interfaces could be defined with. However, recently there has been a consensus that four main levels of abstraction can be distinguished as proposed e.g. in (Szekely, 1996) or (Calvary et al., 2003):

1. definition of tasks and concepts,
2. definition of the modality independent Abstract User Interface (AUI),
3. definition of the platform independent Concrete User Interface (CUI),
4. definition of the platform specific Final User Interface (FUI).

These levels are defined and supported by the Cameleon Reference Framework, which presents a unifying framework that structures the development process of plastic user interfaces, capable of adapting to variations of the context of use (Calvary et al., 2001b) and multi-target user interfaces (Calvary et al., 2003). The framework is built on a set of models, comprising the domain concept model, defining the domain data, the task model, modeling the user tasks, the platform model, defining the target platform of the

context of use, the environment model, defining additional context of use information, the interactor model, comprising the available widgets for the concrete interface, and the evolution model, specifying the transition between states. Based on these basic models, the framework aims at the derivation of transient models by the developer or system. Transient models exist on the four layers of abstraction: Task and Concepts, Abstract UI (AUI), Concrete UI (CUI) and Final UI (FUI). The task and concepts model describes the interaction tasks and the related domain concepts. Based on these basic information, the AUI Model provides a modality independent definition of the interaction, serving as basis for the modality dependent, but platform independent interaction of the CUI Model. The FUI represents the platform dependent user interface code, based on the CUI Model. The framework additionally defines three types of transformations between the models. Reification allows the incorporation of information from a higher level of abstraction into a more concrete model, while abstraction defines the opposite transformation, incorporating concrete information into a more abstract model. Translation finally defines a transformation of the model on the same level of abstraction, e.g. to incorporate a different context of use. The complete model is depicted in figure 3.6.

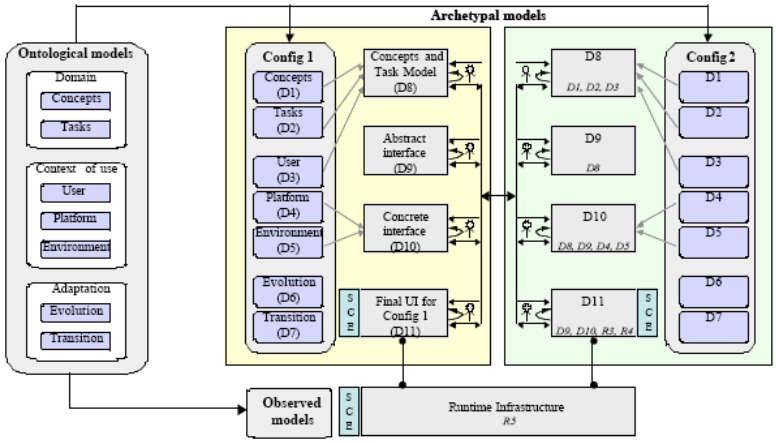

Figure 3.6.: The Cameleon Unifying Reference Framework for Multi-Target User Interfaces from Calvary et al. (2003).

The original framework has been revised in (Calvary et al., 2002) to support additional bottom-up development and reverse engineering approaches as well as multiple entry points for the development process and support for runtime computation of UIs. It has also been extended by the runtime life cycle depicted in figure 3.7. In Calvary et al. (2003) the framework has been extended to support the development of multi-target user interfaces. During this extension, three types of the underlying models have been

identified: ontological, archetypal and observed models. Ontological models thereby give rise to archetypal models, comprising the design information and observed models, guiding the runtime process. Additionally, a runtime infrastructure, following the three-step process of figure 3.7 based on the observed models has been introduced. The reaction of the system based on the identified situation may thereby comprise switching to another platform (migration), switching the underlying executable code to better suit the needs, perform an adaptation of the UI without code switching or execute specific tasks, to alter the trigger situation. One important factor during these adaptations is the conservation of the state of the interaction in the prologue (persistence) which might however not always be possible to any extend and the restoring of the state in the epilogue.

The Cameleon Reference Framework targets the development of multi-platform user interfaces and also addresses the adaptation to platform and environment at design- and at runtime. The structure of the used models has been referenced often in the literature and can be considered as a common set of abstraction levels. However, while the framework provides the means to derive multiple user interfaces from a set of models, as a classification framework, it does not provide details about how this is done.

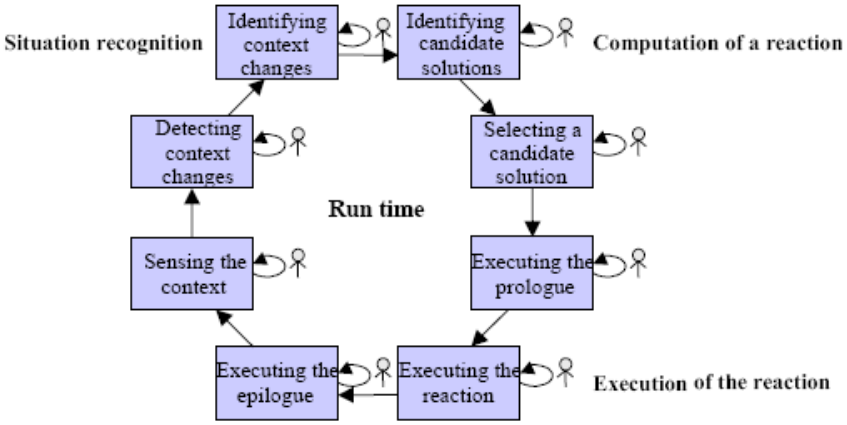

Figure 3.7.: Cameleon Runtime Lifecycle from Calvary et al. (2002).

3.2.3. Models at Runtime

While user interface development tools usually aim at the generation of executable code from the model, rendering the underlying model useless at runtime, there is also an urgent need to handle interaction complexity at runtime. In the 1980 User Interface Management Systems (UIMS) (for an overview see Szekely, 1996; da Silva, 2001) were heavily

researched as approach to cope with the complexity of user interface programming. A typical UIMS included "a presentation layer for input-output handling and device communication; a dialog manager for controlling the semantic interpretation between the system and the user; and an application interface for interpreting the application commands to the dialog manager and the user" (Blattner and Glinert, 1996). However, the approach failed as the resulting systems were large, complex, difficult to learn, and incapable of doing everything the users wanted (Blattner and Glinert, 1996).

Recently, the new challenges in the area of multimodal, distributed and adaptive user interfaces and the increasing complexity of user interface development makes the ideas of early UIMS appear quite appealing again today. The interpretation of XML-based user interface description languages is an important factor for this. With XHTML and VoiceXML being major interpreted languages to create user interfaces today, the question for a comprehensive multimodal user interface description language (besides the limited XHTML+Voice) remains yet unsolved. While some promising languages are discussed in section 3.3, approaches for the interpretation of user interface models at runtime are discussed in this section.

The runtime interpretation of models provides several advantages. While the usual approach to transform models into executable code in the final development step can lead to a loss and scattering of design information, the continuous utilization of the design models at runtime, keeps this information available and accessible. Additionally, the reflection of the state of the system within the model makes this state explicitly accessible and allows its easy monitoring. Supporting the utilization of the models for adaptation means as well as their evolution over time allow the creation of dynamic systems, completely described at runtime by the underlying design models. Decision processes at runtime are supported by design information and thus formal descriptions of meaning of the application elements, or the interaction in case of user interfaces.

However, utilizing models at runtime requires an architecture and a process for the interpretation of the models. Addressing this issue, Sanchez et al. (2008) propose a platform to execute runtime models, that aims at the provisioning of reusable monitoring and control tools. The presented Cumbia platform mainly aims at supporting extensible work flow-based applications but is reported sufficiently generic to be valuable in other contexts. It uses executable models, which keep their structural information during execution. This allows the platform to offer runtime information about every object in the models. Executable models are realized by open objects, combining an entity, a state machine, and a set of actions. While the entity holds a traditional object with attributes

and methods, the state machine defines the life cycle of the entity, which allows access to its state and to react to its changes. Actions are pieces of behavior, executed when the state machine performs a transition. Based on these meta-elements of all models, the platform is capable of handling any type of model with a corresponding metamodel.

Maoz (2008) describes model-based traces, which reflect the runtime state of a system using design models. The approach traces behavioral models of a system design during its execution and follows their activation and progress as they come to life at runtime. This allows to reconstruct and replay a system execution at the abstraction level defined by its models, which makes the execution of the program visible at this abstraction level. Additionally, the live synchronization makes the state of the system perceivable within the model. This enables the strong coupling of dynamic analysis and model-driven engineering. Rohr et al. (2006) describe an approach for the model-driven realization of self-management, based on meta-models that define constraints, monitoring and reconfiguration. A runtime model reflects the current state of the system with respect to its architectural entities.

One major application of models at runtime is the adaptation of systems at runtime. Here, the availability of the underlying design decisions is a major aspect, aiming at the semantic understanding of the performed adaptations. While architectural properties are usually reflected through runtime artifacts, which is suitable as long as the software system remains fixed at runtime. This approach is likely to fail if user needs and operating conditions vary dynamically. Floch et al. (2006) utilize architectural models for the runtime adaptation of applications within the MADAM project. Another approach for runtime adaptation based on models is discussed in (Schneider and Becker, 2008). The authors utilize runtime models for self-adaptation in the ambient assisted living domain in the context of the BelAmI project. They equip interactive components with an adaptation and a configuration model and describe an infrastructure, where an adaptation manager monitors the context to decide upon possible system changes, which are then performed by a configurator component.

In the domain of user interfaces, Klug and Kangasharju (2005) present executable task models to create applications adapting to the actions of the user. They extend the ConcurTaskTree notation to allow the dynamic execution of a task model at runtime by keeping the active task state for leaf tasks. Additionally, input and output ports extend the temporal operators to facilitate information exchange. Another approach utilizing models at runtime is the DynaMo-AID runtime architecture. The system presented in (Clerckx et al., 2004) utilizes a task and a context model to generate context-sensitive

user interfaces.

Sottet et al. (2007a) propose the utilization of user interface models at runtime to sustain user interface plasticity and perform UI adaptation. Traditional user interface models are enhanced with transformations and new models to support reasoning about their own adaptation when the context of use changes. The used models form a net of models, that comprises user-, platform- and environment model, a property model, as well as concept- and task-model. A workspace-, interactor- and program model provide different perspectives to the UI at design-time. Transformations are able to transform information from one model to another one, mappings keep the models connected at runtime.

These approaches show the growing utilization of models at runtime for software and user interface development, to cope with limitations of current approaches and to stronger interconnect design and runtime information.

3.2.4. Summary

In this section, the idea to utilize models to describe user interfaces has been introduced and recent work in this are has been presented. Utilizing the idea to describe everything as a model, this software engineering approach can also be applied to user interface development. However, while the application of model-based software development lead to various successful approaches, model-based user interface development still faces the gap between freely creative UI design and UI engineering, but the increasing complexity of future UIs are likely to render current techniques insufficient.

Utilizing user interface design models at runtime keeps design information available to allow the revision of design decisions and the adaptation of the user interface. It unveils the meaning behind the UI components and facilitates a better semantic understanding of the interaction. The underlying user interface model is interpreted and likely to evolve over time as the state of the interaction changes. It can dynamically express its state within the model to make it explicit to any observers.

The main requirements for the utilization of such an approach are (1) the availability of a set of models in form of a user interface description language, as well as (2) the availability of an architecture, providing the means to interpret the models and handle interaction with the user, incorporation of context information and the connection of backend services. Both aspects are further analyzed in the following.

3.3. User Interface Description Languages

Multi-platform and multimodal user interfaces raise the need for a more declarative description of human-machine interaction on a higher level of abstraction. This has lead to the definition of multiple User Interface Description Languages (UIDLs). Initial challenges for UIDLs in the context of smart environments are especially the lack of a dominating interaction paradigm, the needed interaction flexibility and the goal to define one common description for all possible instances of a user interface. In the following a selection of User Interface Description Languages (UIDLs) is analyzed within the context of the goals of this thesis. Main requirements to these languages from the perspective of Ubiquitous User Interfaces are:

- support for the definition of multi-platform and multimodal user interfaces
- information modeling at higher levels of abstraction and the possibility to create device-dependent and multimodal presentations from the generic information
- support for the synchronization of the different (parts of the) user interfaces
- runtime support and the integration of representation of the state of the system
- the possibility to adapt the resulting user interface or even the underlying model dynamically to context information

In this section three selected approaches are presenting.

3.3.1. UIML

The User Interface Markup Language (UIML) (Abrams et al., 1999) is an XML-based meta-language, aiming at the definition of a canonical representation of any user interface for multi-platform, multi-lingual, and multimodal purposes. It provides a limited number of tags and and allows the definition of vocabularies for different purposes. The meta-language consists of five parts:

- descriptions, that specify how elements are rendered and what their functions are;
- structures, that specify which elements from a description are presented for a given device and how they are structured;
- data elements, that collect all application specific data;
- style elements, that contain device dependent style and data;

- events, that define runtime events between interface elements or interface elements and the backend.

Using UIML to develop user interfaces, the developer has to define an UIML vocabulary that specifies the allowed UI elements (similar to a DTD for XML documents) and their mappings to a user interface toolkit. Different vocabularies supporting HTML, WML, VoiceXML or Java have been developed. Additionally, generic vocabularies (Plomp and Mayora-Ibarra, 2002), addressing multiple renderings and UIML interpreters have been defined (Luyten et al., 2006a). A main goal of UIML was also the separation of user interface and backend. For this purpose, UIML hides calculations and backend calls by aiming at a runtime engine, that monitors all events and provides the communication with the backend.

Different extensions to UIML have been defined to address support for multimodal and distributed user interfaces. The Dialog and Interface Specification Language (DISL) (Bleul et al., 2004; Robbie Schaefer, 2006) and the Cooperative User Interfaces Markup Language (CUIML) (Sandor et al., 2001) extend UIML with generic widgets, new behavioral aspects and address the need to synchronize multiple generated views. The main focus of DISL is the creation of generic and modality independent dialog descriptions. It defines an extension mechanism and the following generic widgets:

- output: *variablefield* and *textfield*
- input: *command*, *confirmation*, *variablebox* and *textbox*
- structuring and selection of structured elements: *choicegroup* and *widgetlist*
- extension elements: *genericfield*, *genericcommand* and *genericbox*

The underlying dialog model provides a new separate layer for the integration of different modalities and the behavior section is extended with possibilities to specify variables, events, new rules and transitions.

In contrast to DISL, CUIML targets the development of user interfaces for wearable computers consisting of numerous devices like head mounted displays, palm-size devices, and speech recognition systems that support multimodal interaction. To synchronize these different UI parts, CUIML suggests a MVC-based (Model-View-Controller) framework, allowing fast information exchange and the application of manipulators that directly change parts of a view instead of rerendering the view as a whole.

3.3.2. TERESA XML

The Transformation Environment for inteRactivE Systems representAtions (TERESA) is a model-based tool to support the authoring of multi-device user interfaces based on the TERESA XML (Berti et al., 2004) language. It has been recently extended to also support multimodal user interfaces (Paterno et al., 2008) and supports the four CARE properties (*Complementarity*, *Assignment*, *Redundancy* and *Equivalence*). It also structures interaction into three phases: Prompt, Input, Feedback. TERESA XML distinguishes three levels of abstraction: task definition, abstract user interface and concrete user interface.

The task model is based on the Concurrent Task Tree (CTT) Notation (Paternò, 1999) and defines user, system and interaction tasks. The tasks are hierarchically ordered and temporal operators define their relations. The CTT notation allows the definition of the model in XML.

The abstract user interface definition is derived from the task model and aims at describing the device and modality independent semantics of the user interface. It reflects the defined temporal relations between tasks and provides multiple presentations. Each presentation combines the set of interactors (at a given time) with connections, defining the dynamic behavior of the user interface. Considered presentation elements are output (text, object, description, feedback) and input interactors (*selection*, *control* (*activator*, *navigator*), *edit*, *interactive description*) as well as composition operators (*grouping*, *relation*, *ordering*, *hierarchy*). The abstract level is complemented by an interaction model, adding dynamic behavior for each presentation. The model defines presentation states and related transitions between the states. Each transition is associated with an initial state, an interactor to trigger the transition, an abstract event (*selection*, *editing*, *activation*) and a target state. Each event can specify additional consequences (*change state*, *modify interactor*, *modify another interactor*, *conditional consequence*, *set variable*, *generic function*).

The concrete user interface definition refines the abstract elements for specific devices and modalities. It defines the rendering possibilities for each abstract interactor and reifies the abstract events (selection can e.g. be mapped to a 'onClick'-event). Figure 3.8 shows the abstract and concrete interactors in an excerpt from the TERESA user interface metamodel for the graphical desktop.

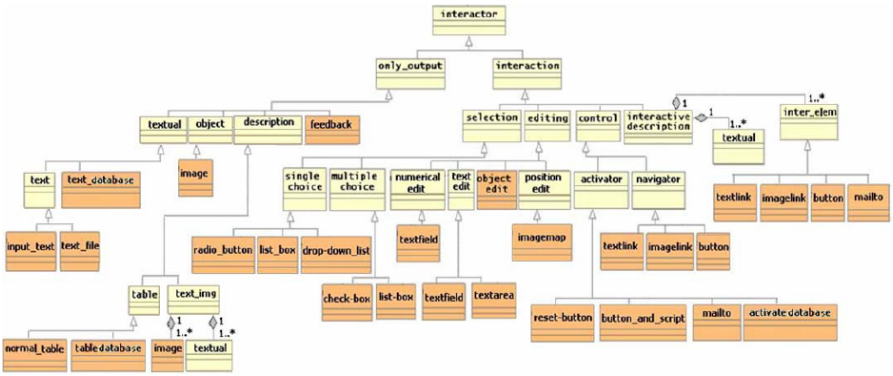

Figure 3.8.: Excerpt from the TERESA concrete user interface metamodel for the graphical desktop from Paterno et al. (2008)

The underlying model-based transformational approach of TERESA allows the transformation of the concrete user interface description into a final user interface noted in a supported target language (currently XHTML, VoiceXML, X+V, SVG, Xlet, and a gesture library) and is combined with comprehensive tool support.

3.3.3. USer Interface eXtensible Markup Language (UsiXML)

The USer Interface eXtensible Markup Language (UsiXML) (Limbourg et al., 2004b) is an XML-compliant markup language that addresses platform-, modality-, and context-independence and -sensitivity of user interfaces. Providing multiple levels of abstraction (task and concept, abstract UI, concrete UI and final UI) as defined by the Cameleon Reference Framework (Calvary et al., 2003) and a set of different models it aims at the realization of multi-path development of user interfaces. A UsiXML UI consists of three initial model: task, domain, and context. They describe the tasks to accomplish in a CTT-based notation, the Unified Modeling Language (UML)-based definition of domain objects and a definition of the relevant context information (user, platform, environment). Based on this information, transformations allow the creation of an abstract user interface and, based on this, a concrete user interface model.

The abstract user interface addresses the device independent meaning of the user interface. It contains abstract interaction objects distinguished in abstract individual components with different facets (input, output, navigation, control) and abstract containers as well as abstract user interface relationships (decomposition, abstract adjacency, spatio-temporal, dialog control, mutual emphasis).

Based on the abstract user interface definition, the concrete user interface model provides the means to specify modality specific (but platform independent) properties of the final user interface. It uses concrete interaction objects to provide an abstraction of the final widget sets (e.g. HTML, Java, Flash). The layout of the concrete interaction objects is defined via relative relations between them. The concrete UI also aims at the provisioning of a navigation definition and events/actions providing the dynamic behavior as part of the dialog. Stanciulescu (2008) introduces a set of voice- and multimodal concrete interaction objects to extend UsiXML with support for multimodal user interfaces. Figure 3.9 shows the concrete voice interaction objects of UsiXML.

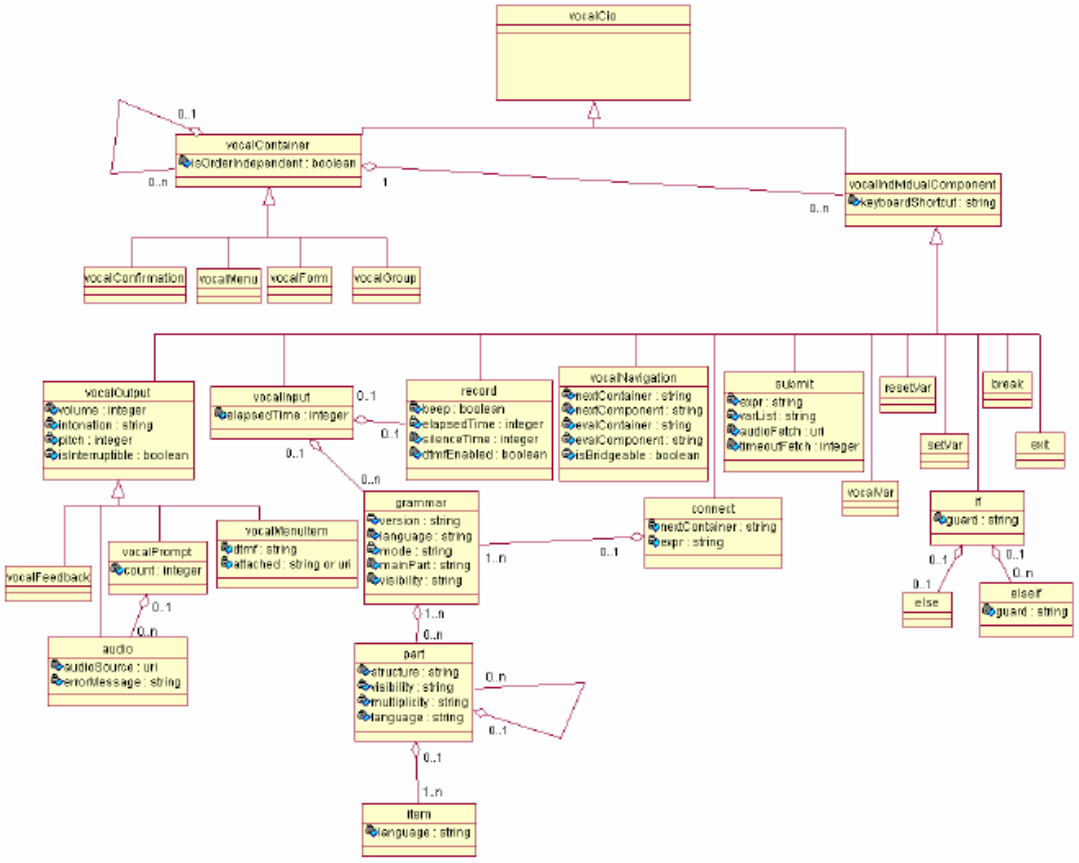

Figure 3.9.: UsiXML voice concrete interaction objects taken from Stanciulescu (2008)

The mapping model defines a set of relationships (*triggers*, *observes*, *updates*, *isReifiedBy*, *isAbstractedInto*, *isGraftedOn*, *isExecutedIn*, *isTranslatedInto*, *isAllocatedTo*, *manipulates*, *hasContext*, *isDelegatedTo*, *isShapedFor*) between the elements of the different

model. It also specifies the information exchange between the different parts of the user interface description.

The combination of the different models with comprehensive tool support addresses the development of multi-platform user interfaces via various development methodologies. The widespread use of the language and its continuous extensions make it a promising approach to handle the upcoming problems of future user interfaces.

3.3.4. Other

Besides the languages that have been introduced above, a variety of UIDLs have been created for different purposes. In the following some interesting aspects of other languages are emphasized to complete the overview of relevant languages.

The most common and well known example for an interpreted (UI-)language is probably (X)HTML (eXtended Hypertext Markup Language) available in version 5 today. It is interpreted by web browsers which are available for various fixed and mobile devices now, to provide web-based graphical user interfaces. It can be enriched with dynamic information via Javascript and allows the separation of style and content via CSS. Similarly, VoiceXML allows the creation of voice-based user interfaces, interpreted by a voice browser, that provides the means to output voice and interpret vocal user input. A first approach to provide multimodal UIs has been provided by XHTML+Voice, which aims at combining XHTML and VoiceXML in a single document to enrich graphical user interfaces with voice output. In the same language family, XForms has been developed to provide a stronger separation of content and presentation. The Synchronized Multimedia Integration Language (SMIL) has been developed to synchronize multimedia content in time. Finally, the Extensible MultiModal Annotation markup language (EMMA) aims at the standardized representation of interpreted multimodal content for text, speech and handwriting. All languages are standardized by the World Wide Web Consortium (W3C).

The eXtensible Interaction-Sheet Language (XISL) (Katsurada et al., 2003) provides means to build multimodal UIs based on interaction scenarios. It controls dialog flow and transitions similar to VoiceXML and supports the synchronization of input/output similar to SMIL. Unique features of this approach are the separation of content and interaction (HTML elements are referenced from a XISL definition via XPath expressions) and its definition of parallel, sequential and alternative input.

The eXtensible Interface Markup Language (XIML) (Puerta and Eisenstein, 2001) aims

at the creation of a universal specification of interaction data. Based on the concepts of components, each consisting of a set of interface elements, attributes and relations, developers can define their own languages or use a predefined set of components. Similarly to UsiXML, XIML follows a model based approach and allows the definition of a task-, domain-, user-, dialog-, and presentation model. It supports relations between the different models like "Data of type A is displayed with presentation element B or C", which allows the definition of structures that can be examined by a rendering software.

The Personal Universal Controller (PUC) Specification Language aims at the automatic generation of remote control interfaces and has an interesting approach to support state variables to explicitly incorporate and manipulate the interaction state during execution (Nichols et al., 2006).

The Multimodal Markup Language (M3L) (Wahlster, 2003) has been developed within the SmartKom project and serves as exchange format for the different modules of the system. It combines concepts from frame languages, concept languages and terminological linguistics as well as web languages to provide object oriented modeling, formal semantics and inference capabilities based on an XML/RDF syntax. It is e.g used to express temporal relations between gestures and voice and allows the mapping of typed feature structures to M3L constructs.

3.3.5. Summary

In this section User Interface Description Languages that address the definition of multimodal interaction and multi-platform user interfaces have been presented. The described approaches mainly address the derivation of multiple (multimodal) user interfaces, supporting different devices. However, from the perspective of Ubiquitous User Interfaces, it has to be noted, that none of the UIDLs incorporates means to integrate all identified features of UUIs. Considering these features, it also remains unclear, how the final UI code is created from the models, as the consideration of information only available at runtime plays an important role for shaping, user interface distribution, the dynamic usage of modalities or adaptation. None of the presented approaches directly aims at the utilization of the UI description at runtime. Thus, in the next section different architectures are presented, that put a strong focus on addressing the various runtime needs of Ubiquitous User Interfaces.

3.4. Architectures

The definitions of architectures allowing the creation of multimodal as well as distributed and adaptive user interfaces have been widely discussed in the literature. In this section selected architectures are presented, with a focus on architectures that facilitate these features.

3.4.1. W3C Multimodal Interaction Framework

The Multimodal Interaction Framework of the W3C (Larson et al., 2003) aims at an identification of the major components of multimodal systems. It identifies markup languages to describe information (for components and to support data flow) and supports current and future input and output modes. It is explicitly defined as a framework being one level above a concrete architecture.

The main components identified by the framework are depicted in figure 3.10 and described in the following.

- The Human User is an actor, using the system by entering input and perceiving output.
- Input provides a representation for multiple input modes (defined as audio, speech, handwriting, keyboarding, and other input modes).
- Output provides a representation for modes of output (defined as e.g. speech, text, graphics, audio files, and animation).
- An Interaction Manager is the component that coordinates data and execution flow from various input and output components. It maintains the interaction state and context of the application and responds to inputs from components. It also reacts to changes in the system and environment and coordinates input and output.
- Application Functions connect to the functional core and the application logic.
- The Session Component provides state management and (temporary and persistent) sessions.
- A System and Environment Component connects the interaction component and the actual environment information.

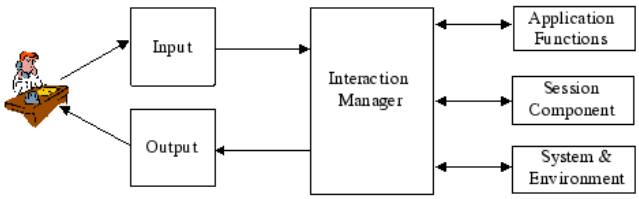

Figure 3.10.: General components of the Multimodal Interaction Framework from Larson et al. (2003)

Input is supported by the cooperation of a recognition component, capturing input from the user (e.g. based on a grammar, e.g. W3C Speech Recognition Grammar for speech), an interpretation component, processing the results of recognition components in terms of meaning and semantics and an integration component, combining the input from multiple interpretation components. EMMA (Baggia et al., 2008) is used to define information exchanged between the components.

Output is realized by the cooperation of a generation component, selecting the used output modalities, a styling component, adding layout information, and a rendering component, rendering the final e.g. graphical or speech user interface. The output components use e.g. Speech Synthesis Markup Language, Cascading Style Sheets or the Synchronized Multimedia Integration Language for styling and create XHTML or Scalable Vector Graphics as output.

3.4.2. MultiModal Dialog System

The conceptual structure of the MultiModal Dialog System (MMDS) of Delgado and Araki (Delgado and Araki, 2006) provides a system architecture consisting of several modules organized in three main components: input interface, multimodal processing and output interface. Input and output are individually addressed and a central model handles the dialog flow.

Similar to the Multimodal Interaction Framework the concept identifies multiple input and output modalities. While input is integrated by a multimodal fusion engine, output is coordinated by a response generator. The main difference to the W3C framework is the more detailed approach for the interaction management in form of the multimodal processing. The integration of a dialog manager, based on a task description and a database, containing domain objects, allows a more detailed description of the multimodal dialog.

3.4.3. ICARE

Based on the definition of the CARE properties (see section 3.1.4), ICARE (Interaction-CARE) (Bouchet et al., 2004) provides a component-based approach for the development of multimodal interfaces. It specifies elementary components, describing pure modalities, and composition components, allowing the combination of modalities according to the CARE properties. Elementary components comprise device components, allowing the translation of device specific input into recognizable utterances, and language components, allowing the translation of the utterances recognized by the device component into higher level commands. Four generic composition components finally implement the CARE properties, merge the data (supporting the Bucket-fusion mechanism from Nigay and Coutaz, 1995) and send it to the next linked care component, allowing a cascading of the components. The hierarchical combination of the components allows the flexible combination of multiple modalities and the expression of the interrelation of the elements. This allows the interpretation of multimodal input on multiple levels of abstraction, based on the consideration of the CARE properties. The system focuses on the fusion of multimodal input and does not yet support fission or multimodal output. Several systems have been developed based on the ICARE system and the underlying fusion mechanism has been tested and enhanced in (Dupuy-Chessa et al., 2005).

3.4.4. Cameleon-RT

Cameleon-RT (Balme et al., 2004) provides an architecture reference model supporting distributed, migratable plastic user interfaces. It defines three basic layers of abstraction: the Interactive Systems Layer, the Distribution, Migration and Plasticity (DMP) Layer and the Platform layer, shown in figure 3.11. While the DMP Layer is the central layer of the approach, the Interactive Systems Layer comprises the applications currently running, and the platform layer forms the basis of the system and includes hardware and operating system.

The DMP Layer provides mechanisms and services for DMP UIs. It therefore comprises context infrastructure to incorporate context information as well as a platform manager and interactors toolkit to support resource discovery, hide the platform heterogeneity and support distribution and migration of the UI. It also hosts the Open Adaptation Manager, which is one of the key components of the architecture. It incorporates the location-, user- and platform-information as well as information about the interactive system through observers, enabling the situation synthesizer to compute the current

situation from the information. If a new situation is discovered, the evolution engine identifies the components of the UI that must be adapted and provides a configurator with a plan of actions. The configurator finally executes the plan and performs the required adaptations. It can access a component manager to retrieve required components. The configurator is also able to retrieve the state of a running component and transfer it to a new component, replacing the first one.

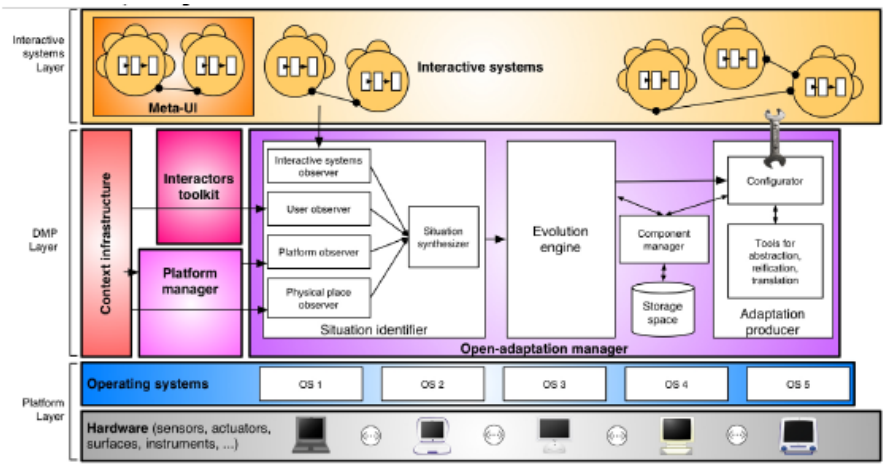

Figure 3.11.: The Cameleon-RT architecture. (Balme et al., 2004)

3.4.5. DynaMo-AID

DynaMo-AID (Dynamic Model-bAsed user Interface Development) allows to develop context-sensitive user interfaces for pervasive systems that support dynamic context changes (Clerckx et al., 2004, 2006). It comprises a task-centric design process as well as a runtime system and is part of the Dygimes User Interface Creation Framework (Coninx et al., 2003). The design process consists of seven steps:

1. the creation of a task model,
2. attaching abstract descriptions to each task,
3. calculating the collection of task trees relevant for different contexts,
4. automatic extraction of the dialog model,
5. dialog construction by the designer,
6. linking of context to task and dialog model and
7. service modeling.

Based on this approach an application model is created, providing multiple variations of a task tree according to different supported context variants.

To put these design principles into practice, the DynaMo-AID approach comprises a runtime system depicted in figure 3.12, consisting of three distinct modules: (1) the application, that manages the functional core and the state of the application, (2) context, provides context information to the dialog controller and the functional core, and (3) presentation, provides the actual presentation of the UI. The three modules are connected by the dialog controller as the central component of the system. The dialog controller controls the communication between the modules, manages the dialog models of the application and maintains the state of the user interface. Based on the detected context information the active variant of the task-model is selected. Additionally, whenever a service (dis-)appears the corresponding task tree can be attached to a currently active task tree if there is a reserved spot. Each service is therefore associated with the task tree to provide a high-level description of the required interaction.

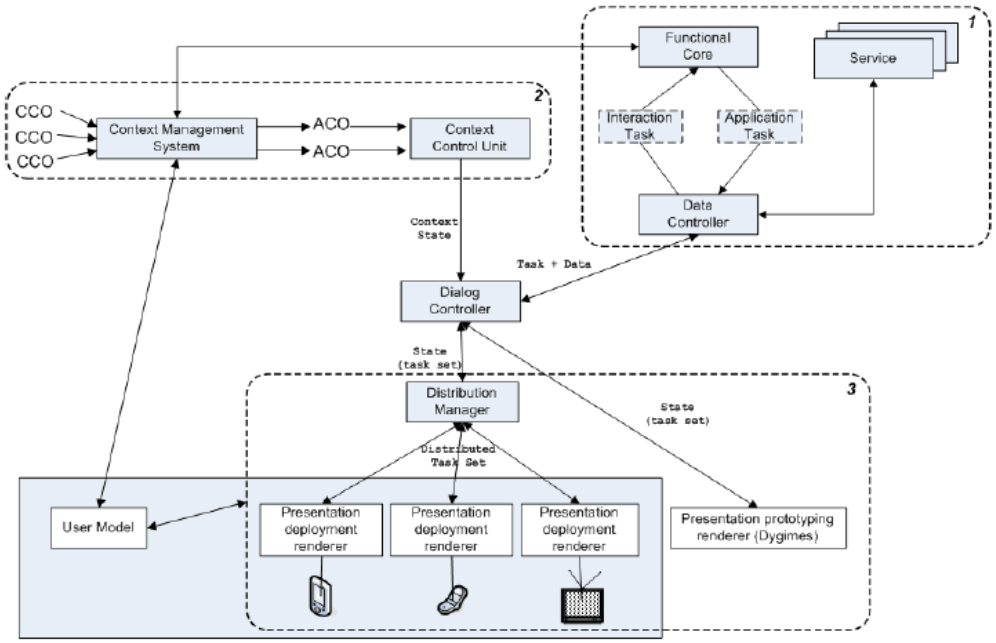

Figure 3.12.: The DynaMo-AID Runtime Architecture, adopted from (Clerckx et al., 2006).

The distribution manager holds a distribution model, describing to which type of device a task or dialog element can be distributed and distributes the elements of a dialog across devices accordingly. It keeps track of the available interaction devices via UPnP-

based device discovery. The available devices are stored in the environment model. The final rendering of the user interfaces is handled by the interaction devices, running UIML renderers (Abrams et al., 1999) or a presentation manager to support XHTML or XHTML+Voice (Clerckx et al., 2007).

3.4.6. FAME

The Framework for Adaptive Multimodal Environments (FAME) (Duarte and Carriço, 2006) provides a model-based approach that comprises an architecture, a representation of adaptation rules and a set of guidelines for the development process. The architecture, depicted in figure 3.13, uses several models for controlling multimodal outputs and presentation layout as well as for the interpretation of user input.

The models used in the architecture, store context information (user-, platform & devices-, and environment model) and describe the UI elements available for presentation and interaction (interaction model). The user model stores user preferences, goals, and past interaction history as well as general user characteristics. The platform and devices model stores information about available interaction devices, platform characteristics and application specific events. It is relevant to the choice of what modalities to use for input and output and the selection information presented to the user. The environment model stores information about the surrounding environment like noise level of lighting conditions. The models are continuously updated by a set of observers for device and environmental changes with the possibility to process application generated events and support the fusion of input events. The additional interaction model stores templates, defining the components that are used to generate the actual interface, according to the context of use. Atomic templates define the presentation of a component for a single modality; composite templates express relations between templates and allow their grouping, even across modalities.

The underlying adaptation knowledge is expressed in the Behavioral Matrix, introduced to reduce the complexity of expressing the adaptation rules. One Behavioral Matrix is defined for each component in the interaction model and encodes the behaviors and transitions between them. The adaptation module is the core of the architecture and combines the different models with the Behavioral Matrix to process user inputs, determine context changes, and adjust the parameters controlling the multimodal operation of fusion and fission.

The outer layer of the depicted architecture provides means for multimodal input pro-

cessing and output generation. The input processing elements translate and forward information from outside the application boundaries to the adaptation module, distinguishing user inputs, environmental changes, device changes and application generated events. User inputs are processed by a multimodal fusion component, determining the intended user action from the input. Weights of modalities and underlying integration patterns of the fusion engine are thereby controlled by the adaptation module. The output generation process combines a multimodal fission and an adaptive arrangement component. The fission process separates the data received from the adaptation module and coordinates the presentation of the different modalities. The arrangement component finally deals with the arrangement of the presentation elements of each single modality. The parameters of both components are controlled by the adaptation module.

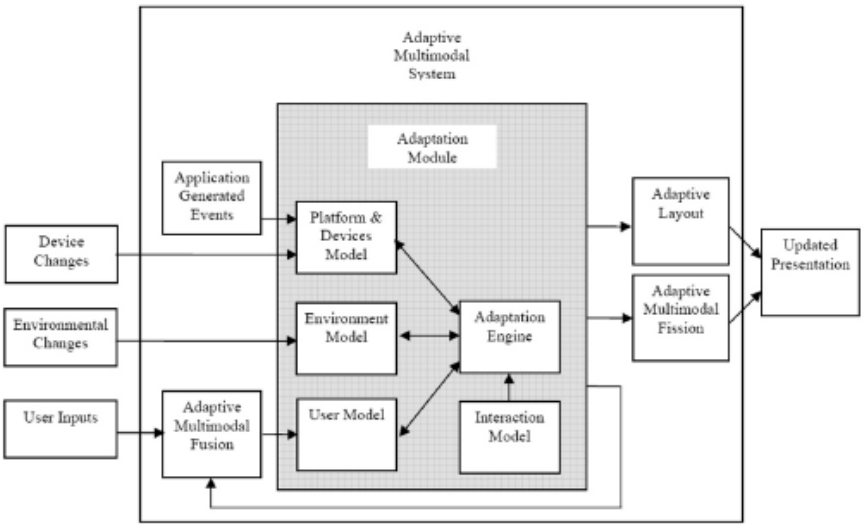

Figure 3.13.: FAME Architecture taken from Duarte and Carriço (2006)

3.4.7. DynAMITE

DynAMITE aims at "intelligent multimodal interaction with distributed networked devices within dynamical changeable ensembles" to realize assistive systems and services (www.dynamite-project.org, last visited March 2009). Its goal is to empower devices and software components (of different software producers) to interact spontaneously. This should allow to analyze the user's interaction in terms of intentions and goals and to act accordingly across multiple devices in a so called ensemble. The approach comprises three main aspects:

1. A Software-Infrastructure to realize the communication between distributed components.
2. A Topological Model to define the required components for the realization of reasonable behavior of smart environments and for the self-organizing management of the communication flow.
3. Semantic Models to define common ontologies for the different components within smart environments.

The DynAMITE-Software-Infrastructure (Heider and Kirste, 2002) is based on the SodaPop-model, that distinguishes two fundamental elements: channels and transducers. Transducers send and receive messages and communicate with each other via channels. Channels deliver messages and use conflict resolution strategies to determine addressees. It realizes the essential communication patterns of data-flow based multi-component architectures.

The topological model consists of five components, depicted in figure 3.14.

1. Input components that monitor user interactions.
2. Output components rendering information from Dialog components in a user friendly manner
3. Dialog components that collect and interpret input information and control the dialog with the user to identify the user goals.
4. Strategy components analyze the detected user goals and to map them to executable functions.
5. Actuator components finally execute the called functions.

These components allow the definition of multimodal interfaces that separate input and output and allow the management of human-computer dialogs to derive an action strategy and finally execute the appropriate actions.

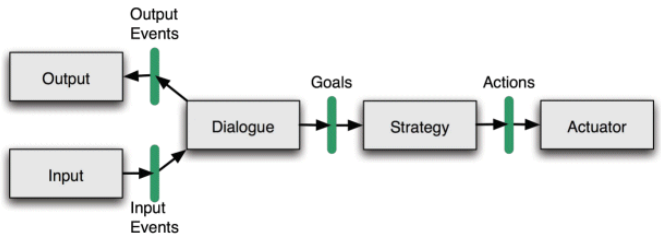

Figure 3.14.: Components of the topological model of DynAMITE. Taken from (Kirste, 2004).

The semantic model of the approach finally distinguishes user and preferences, devices and the environment and allows the description of the context of the interaction.

The DynAMITE approach addresses the needs of interaction within distributed systems and explicitly focuses on the semantic processing and understanding of user input to derive interaction strategies and finally identify the required function calls to match the users goals and needs.

The same authors also propose a conceptual architecture based on common basic architectural building blocks for situation-aware multimodal interactive assistance systems, derived from the evaluation of multiple systems: EMBASSI, MAP, and SmartKom with rather comprehensive approaches, and ARVIKA, INVITE, and MORPHA (Kirste and Rapp, 2001). The main building blocks of the architecture are:

1. The **Multimodal UI** combines input recognition and fusion as well as output generation and presentation planning with an appropriate dialog management.
2. The **Autonomous Control** elements maps user requests to action sequences, controlling the external world.
3. The **Context Knowledge** components contain knowledge about the external world in terms of domain, discourse, user, resources and environment information and provide this to the UI and the control elements.

The proposed infrastructure explicitly addresses smart environments and distributed systems. It goes beyond the pure definition of user interfaces and again also considers the control and monitoring of the environment (the external world).

3.4.8. SmartKom

SmartKom (Wahlster, 2006) is a multimodal dialog system that symmetrically combines speech, gestures and facial expressions for input and output. It uses an animated agent to realize multimodal interaction reflecting the communication habits of the human user. It features the processing of imprecise, incomplete or ambiguous input and provides presentations, coordinating multiple modalities. Based on shared knowledge services, the system is able to reason about user input and to create multimodal output. An intention analysis interprets the fusion results and initiates the action planning, to create an appropriate answer. Backend services are integrated via a function modeling component. The information exchange between the components is realized via the M3L language, allowing the representation and exchange of multimodal content with a single language.

SmartKom realizes three different scenarios, namely a multimodal portal to information services at home/office, a communication kiosk for public places, and mobile services on a PDA.

3.4.9. Other Systems

Besides the presented systems, several other projects also provide interesting approaches and solutions. Some are briefly introduced in the following.

The Open Agent Architecture (OAA) (Cohen et al., 1994; Moran et al., 1997; Cheyer and Martin, 2001) is an agent-based system, allowing the distributed execution of user requests through a multimodal interface supporting pen, voice and direct manipulation. It is based on a hierarchical blackboard approach that utilizes facilitator agents to store global data as well as schedule and maintain information flow. Requests that can not be handled are passed to the next higher hierarchical layer. Agents can be distributed across different devices and communicate via the network. User Interfaces are implemented by a set of agents coordinated by a central User Interface Agent that hides the currently used modality from the underlying application implementation. Micro-agents, handling different modalities independently, comprise gesture, handwriting and speech recognition. A Modality Coordination Agent combines the inputs from different modalities and resolves references to derive the user's intention. Application Agents provide application specific services for specific tasks (e.g. speech recognition).

The Quickset system (Cohen et al., 1997) is based on the Open Agent Architecture and runs on desktop and handheld PCs, communicating via (wireless) LAN. It employs speech, gesture, and direct manipulation to formulate a military scenario that allows creating, positioning, and editing units as well as supplying them with additional information. It allows the user to gesture and draw directly on a map while uttering spoken commands. Speech recognition, gesture recognition and natural language agents perform the basic input processing. A multimodal integration agent provides a unification mechanism based on typed features structures to integrate multimodal input. Output can be created via a text-to-speech or a web-display agent as well as through the coupling of other systems via additional agents. The Quickset system offers capabilities for collaborative user interfaces, where multiple users can have different interfaces, while working with the same system.

The MATCH (Multimodal Access To City Help) system (Johnston et al., 2002) is based on a multimodal application architecture that combines finite-state multimodal language

processing, a multimodal dialog manager using on speech-acts, a multimodal output generation, and text planning algorithms. The implementation is built on a hub-and-spoke architecture similar to MITREs Galaxy System. MATCH itself is a multimodal speech and pen based interface to restaurant and subway information for New York City for mobile devices.

The i-Land system (Streitz et al., 1999) aims at the creation of dynamic workspaces for future cooperative work of dynamic teams by enhancing existing roomware with additional electronic functionalities. i-Land is based on the BEACH software infrastructure (Tandler, 2004), supporting collaboration with heterogeneous devices. An application of the BEACH infrastructure is the ConnecTables system (Tandler et al., 2001) that, similar to I-AM, aims at connecting multiple surfaces for interaction.

Another approach to address multimodal issues are multimodal widgets. Blattner et al. (1992) present MetaWidgets as part of the PolyMestra System (Glinert and Wise, 1996) to create multimodal widgets that abstract information from the application to the user. Each widget contains a set of representations in different (combinations of) modalities and methods of selecting among them to determine the optimal presentation. In the background of the PolyMestra system a resource monitor calculates the total cognitive load of the user and a presentation manager determines which information to present based on the cognitive load, user preferences and other system data. Kobayashi et al. (2005) also present an approach identifying *Trigger*, *Delta*, *BoundedValue* and *TextEntry* as input types on a meta-level, that can be utilized to connect widgets supporting different modalities to a service component of an interactive system. Crease et al. (2000) present a toolkit for the development of widgets that are capable of presenting themselves in multiple modalities and adapting to the context in which they are used.

Aiming at bridging the gap between model-driven engineering and these interactor toolkits, COMET defines a software architecture for task-based plastic interactors (called COMETs). COMETs can be rendered in different variations and support multiple interaction technologies. Each COMET is composed of three facets: Logical Consistency, to represent the user tasks, a Logical Model, to realize the semantics, a Physical Model to realize physical and functional properties and implement the semantics. COMETs are structured in a graph of multiple COMETs to realize complex user interfaces. A semantic network links together different concepts ranging from final UI to task and concept level.

3.4.10. Discussion

In this section, eight selected architectures for the creation of multimodal, distributed and adaptive user interfaces have been presented and common aspect have been identified. The architectures have been developed with different foci and different applications in mind and cover various abstraction levels. Compared to the features identified in chapter 2, none of the architectures completely covers all of the features, but each approach addresses some of the features and each of the features has been covered by at least one of the architectures. However, if one would set out to implement a Ubiquitous User Interface there would be no framework, architecture, tool or reference implementation, covering all the needs.

The following tables provide a comparison of different aspects of the presented architectures with respect to the features of UUIs. Table 3.1 compares the aspects related to multimodality:

- **Multimodality**: Modalities covered by the approach.
- **Fusion**: Describes how the combination of user input from multiple modalities is supported.
- **Fission**: Identifies the means to separate output across modalities.
- **Separation of input and output**: Denotes the capability of the system to separate input and output on the UI level.

Framework	Multi-modality	Fusion	Fission	I/O Separation
W3C MIF	voice, handwriting, keyboard, graphic	multimodal integration component	output generation component that selects the used modalities	explicitly separates input processing and output generation

Framework	Multi-modality	Fusion	Fission	I/O Separation
MMDS	voice, handwriting, gesture, face, gaze, lip reading, keyboard, graphic, haptic output	fusion component	response generator component	input and output interface are distinguished
ICARE	input only, e.g. voice, mouse, location/ orientation tracker, graphics	composition components (implement the CARE properties)	fission is not addressed	considers only input
Cameleon-RT	not multimodal	fusion is not addressed	fission across modalities is not addressed	does not separate input and output
DynaMo-AID	input and output modalities depend on the available UIML renderers	fusion is not explicitly addressed	fission across modalities is not addressed	does not separate input and output
FAME	input is abstracted as observers, output depends on available/ supported devices	directed by the Behavioral Matrix and controlled by the adaptation module	directed by the Behavioral Matrix and controlled by the adaptation module	distinguishes user input and presentation updates

Framework	Multi-modality	Fusion	Fission	I/O Separation
DynAMITE	addresses e.g. avatars, speech, gesture, position recognition, haptics	fusion component aims at deriving user intentions	presentation planning and generation components are proposed	perception and rendition are distinguished
SmartKom	supports gesture, speech, graphics and a character agent	time-stamped hypotheses and unification grammar	presentation pipeline with presentation planner	separates intention analysis and presentation planning

Table 3.1.: Comparison of the architectures part 1.

Table 3.2 compares distribution and adaptation as additional aspects, directly related to the features of UUIs as well as main aspects like the functional core and the underlying model:

- **UI Distribution**: Identifies the capabilities of the system to distribute a UI across multiple interaction resources and to dynamically change that distribution at run-time.
- **Adaptation**: Denotes the capabilities to adapt the UI to the context of use.
- **Functional Core**: Identifies the capabilities of the system to connect to external application functions and services
- **Modeling Approach**: Lists the models supported by the approach.

Framework	Distribution	Adaptation	Functional Core	Modeling Approach
W3C MIF	focus on multimodal interaction	does not focus on adaptation	provides a component representing the available application functions	not model-based

Framework	Distribution	Adaptation	Functional Core	Modeling Approach
MMDS	focus on multimodal interaction	no focus on adaptation	integration of the functional core is not explicitly addressed but tasks and a database are considered	not model-based
ICARE	considers only input	no focus on adaptation	based on arch and integrates a functional core adapter	not model-based
Cameleon-RT	provides a distribution layer, supporting the handling of distributed components	provides an open adaptation manager	integration of the functional core is not explicitly addressed	models can be considered, but are not explicitly addressed
DynaMo-AID	provides a distribution manager	context adaptation is considered (e.g. for task selection)	the functional core is integrated via a data controller, making service calls based on the task model	considers a task-based application model with multiple variants for different contexts

Framework	Distribution	Adaptation	Functional Core	Modeling Approach
FAME	does not explicitly focus on distribution, but provides multimodal fission	based on the Behavioral Matrix	integration of the functional core is not explicitly addressed	platform& devices-, environment-, user- and interaction model are considered
DynAMITE	addresses interaction within distributed environments	context is considered, but dynamic adaptation is not explicitly discussed	considers the control of functions of the "external world"	domain-, discourse-, user-, resources- and environment model are considered at runtime
SmartKom	does not focus on distribution	a dynamic action planning can consider context information	a function model connects external services	interaction-, discourse- context- and function model are considered

Table 3.2.: Comparison of the architectures part 2.

In summary, a major drawback of the presented approaches is the lack of distribution and adaptation support within the MIF, MMDS and ICARE architectures. ICARE is additionally limited to multimodal input and does not support the creation of multimodal output yet. While aiming at UI adaptation, Cameleon-RT and DynaMo-AID lack support for multimodal interaction. The DynAMITE system aims at the creation of multimodal systems with a focus on distributed interaction in smart environments. While the system is able to incorporate context information into the interaction, it does not focus on the provisioning of adaptive user interfaces. SmartKom provides a very interesting approach to create symmetric multimodal systems, with a focus on speech and gesture modalities, but does not address the dynamic combination and alteration of these modalities. Adaptation is considered only by the action planning component.

The most interesting approach from the perspective of this work is the FAME framework and architecture. It presents a model-based approach to generate adaptive multimodal user interfaces. The framework uses an interaction model, comprising multiple templates for different modalities and modality combinations and facilitates a behavioral matrix, to select the most appropriate template for the current interaction context. Although this approach provides means to adapt to predefined contexts, it does not facilitate open adaptation and the semantic understanding of the templates by the system. It is also unclear, how the templates for the different modalities are defined and how they are synchronized at runtime. Additionally, the approach lacks any means to integrate a functional core or backend services within the developed application.

In summary, the presented frameworks and architectures cover a broad range of topics and provide various capabilities to create innovative user interfaces for different purposes. However, none of the approaches covers all identified features and thus, none of the architectures is suitable for the creation of Ubiquitous User Interfaces for smart environments in the current state.

3.5. Conclusion

In this chapter different aspects of the current state of the art have been evaluated and described with respect to the goals of this thesis. Starting with the analysis of adaptation, shapeability, distribution and multimodality as primary features of Ubiquitous User Interfaces, model-based development has been introduced as approach to cope with the increasing complexity of such user interfaces. The need to define the underlying models leads to the analysis of selected UIDLs, providing the means to express different aspects of the user interface on different levels of abstraction. However, while models and languages (and the appropriate tools) can greatly simplify the development process of user interfaces, there is also a need to address the runtime issues arising with the new challenges posed by Ubiquitous User Interfaces. In terms of the Cameleon Reference Framework this would be the runtime infrastructure, handling the final user interface. Thus, different architectures for the development of user interfaces with a broad range of characteristics have been evaluated. These architectures address various issues related to the creation of Ubiquitous User Interfaces, ranging from the identification of the basic components of multimodal user interfaces as in the Multimodal Interaction Framework of the W3C to the integration of multiple models as part of different architectures like e.g. in FAME and the definition of adaptive widgets and concrete implementations of

different systems. However, up to now, the development of a Ubiquitous User Interface requires a high and usually very complex implementation effort as there are little reusable components for the runtime handling of interaction. None of the approaches addresses all required features and it is yet unclear how a language to express such interfaces and interaction could look like.

3.5.1. Shortcomings

To successfully create applications and user interfaces for smart environments it is necessary, to provide the means to take the dynamic nature of such environments into account. Multiple devices facilitate user interface distribution and migration, different device capabilities require shaping, various usage situations demand for flexible and multimodal interfaces, context variations require adaptations. The need to set up such an environment and maintain it makes it impractical to also maintain a set of stand alone applications. An integrated environment, providing basic services and hosting capabilities would also open possibilities for inter-application cooperation. Integrating such a hosting environment with flexible application implementations would be a major goal here. Directly utilizing application and user interface models at this point would make design details, that might be crucial e.g. for adaptation purposes, available at runtime, instead of having them lost and scattered in the code. The UI models should allow the expression of the interaction means, while also providing leeway for the interpretation process. From this perspective of the development of Ubiquitous User Interfaces, three major shortcomings of the presented approaches have been identified:

Missing runtime concepts in UIDLs can be identified, considering the fact that the models are not only required to describe a static snapshot of the user interface, but represent its dynamics as well as its evolution over time. Concepts for the representation of the interaction state are needed, as well as concepts to configure components that manage fusion, fission or handle interaction resources. Assuming, that the user interface and the underlying interaction are completely expressed within the user interface description, these concepts are mandatory.

No comprehensive approach that reflects the close interconnection of all features has been found. While each of the features is addressed by at least one of the approaches, none of the evaluated approaches addresses all features. Considering the interconnection of all features, it seems however necessary to not only consider their isolated application,

but also their interrelations. Examples for this are the reflection of user interface distribution and modalities within the layout of an interface, the need to reflect supported input modalities also in the output of a UI distribution, or the possibility to reference distributed output interactors and the used devices within the user input.

A low level of integration between architectures and UIDLs can be identified. While the described architectures facilitate the provisioning of multimodal interaction at runtime, they mainly work as blueprints for the development of specific multimodal systems. However, the idea of UIDLs is to automatically derive the final user interface from the description in a transformation process. Currently, there is a gap between the concepts available in the user interface description and their implications at runtime. A possibility to bridge this gap, is the runtime interpretation of UIDLs, which has gained little attention yet.

3.5.2. Requirements

From the perspective of the identified shortcomings, addressing the creation and handling of UUIs raises the need to describe UUIs at design-time and the need to handle interaction with them at runtime. While the former is needed to express the complex interaction as well as interface structure and semantics, the latter allows the interaction with the user and ideally reflects all design-time aspects at runtime.

This leads to the concept of a runtime architecture, handling declarative user interface descriptions to flexibly create and manage Ubiquitous User Interfaces. The architecture and the underlying user interface description language are closely coupled to ensure that the created user interface model holds everything needed to express the design information as well as to allow its runtime interpretation. However, even with a (limited) focus on adaptive multimodal, multi-device and multi-situation user interfaces, the relevant development dimensions are manifold. In the following the required concepts and their relations are identified with a special focus on the adaptation of the user interface in the dimensions of shapeability, distribution and multimodality at runtime. Afterwards, the problem space is analyzed from the architectural perspective, illustrating the concepts an architecture for UUIs has to reflect and the concepts an UIDL has to provide are discussed from the runtime perspective.

The requirements in the following are structured according to the features. However, as the features are closely connected, they also share several requirements which are only introduced once and are not repeated for each feature.

Shapeability

Shapeability of the user interface is the major aspect to support multiple interaction resources and modalities or the dynamic adaptation of the user interfaces. It requires the definition of boundaries for the shaping at design-time and a possibility for flexible adjustments within these boundaries. Temporal and spatial aspects should be considered to support the optimal utilization of interaction resources. Additionally, context of use information, reflecting changes in user-, environment- and platform properties can be considered. This leads to the following requirements:

Requirement #1.1: The developed approach should support the definition of temporal and spatial boundaries at design-time and the flexible adjustment within these boundaries at runtime.

Requirement #1.2: Information about the utilized interaction resources and their capabilities are required to influence the runtime shaping.

Requirement #1.3: Additional context information should be accessible and considered.

Distribution

Distribution allows the simultaneous utilization of multiple interaction resources and, in conjunction with the possibility to dynamically alter these resources, allows the sequential usage of different interaction resources for specific tasks within the interaction process. A separation of input and output allows the independent addressing of resources, but also requires to keep a semantic connection between the two (e.g. resolve deictic references or directly address modalities: “as you can see on screen X”). Fission into and synchronization between multiple representations is required for distributed user interfaces (as well as for different modalities). Support for UI distribution entails the following requirements:

Requirement #2.1: The separation of input and output is required to independently address input and output resources.

Requirement #2.2: The semantic relation between input and output has to be considered, even if the two are technically separated.

Requirement #2.3: Continuous synchronization of different parts of the user interface has to be supported.

Requirement #2.4: Fission of output across resources and modalities is the basis for multimodal output and user interface distribution.

Requirement #2.5: Information about the available interaction resources and their capabilities are required to calculate a distribution (while shaping requires the knowledge about the actually used IRs in #1.2).

Multimodality

Multimodality aims at the interaction via multiple input modalities and the presentation of information via different output modalities to create robust and natural interaction. It is based on the distribution of input across multiple IRs (and modalities) and additionally requires the fusion of multimodal input from multiple IRs. Additionally, streams of input and output (e.g. continuous speech) have to be supported and eventually mapped to processable information chunks. Dynamic support of multiple modalities raises the need to provide a modality independent processing layer, defining abstract interaction semantics as well as the mapping of these to a concrete level addressing modality specifics.

Requirement #3.1: Fusion of user input from different modalities is important to derive the meaning of combined, complementary input.

Requirement #3.2: Continuous interaction requires the constant processing of input streams and the creation of output streams as well as their transformation into discrete information chunks.

Requirement #3.3: The provisioning of interaction concepts and adaptation facilities on multiple levels of abstraction should be considered.

Adaptation

The dynamic adaptation of user interface properties is a major requirement to enable the dynamic reflection of context information and context changes within the user interface. Using a user interface model, UI properties are the design-time information expressed by the model structure as well as the runtime information (state). Making the state explicit at runtime requires a model of the UI state. The dynamic redesign of the UI and the reconfiguration of the features require direct access to and alteration means for model structure and state. Performing these alterations at runtime requires the persistent monitoring of user interaction (e.g. as part of the UI state) to ensure its continuity. This adds the following requirements:

Requirement #4.1: An explicit interaction state is required, to make the current status of the interaction accessible and perceivable.

Requirement #4.2: Direct access to and alteration means of structure and state of the underlying UI model are needed.

Requirement #4.3: Persistence has to be supported to ensure interaction consistency and continuity during adaptation.

General Aspects

In addition to the feature related requirements, additional, more general requirements can be identified. The utilization of a UI model at runtime, underlying the approach developed in this work, raises the need to bridge model and outside world. It requires the reflection of both, design-time information and runtime information, as a model to build a complete internal representation of the system under study - the user interface. While the model provides a view of the system and allows to analyze it, a projection of the results of the analysis into the outside world in form of the created UI is required. Any interaction with this UI has to be interpreted in relation to the model and influences its runtime state.

Requirement #5.1: Utilizing a model at runtime requires to bridge the inner representation - the model - and reflect it to the outside world as well as the reflection of any changes of the outside world (interaction) within the model.

The possibilities to influence the configuration of distribution, used modalities or adaptation at runtime also lead to the demand for user control over this feature configuration. This includes the triggering of adaptations as well as full control over the configuration of the UUI features and thus the behavior of the created user interface.

Requirement #5.2: The provisioning of (meta-)control over the feature configuration of the user interface is an important requirement.

From the runtime perspective, another major aspect is the consideration of the integration of the functional core within the interaction. This is important to actually execute functions realizing the user's intentions in the "real world". This addresses the integration of services as part of the modeled domain, in contrast to the projection of the model state into the world outside of the model of #5.1.

Requirement #5.3: The integration of the functional core is required to connect user interface and backend services.

These derived requirements can be mapped to architectural concepts as well as UIDL concepts, presented in the following.

Architecture Concepts

Developing an architecture, incorporating a user interface model at runtime to handle user interaction and make it more flexible is based on the idea of having additional design information available at runtime and being able to utilize and alter these information. This requires the semantic understanding of the underlying model and the possibility to map this model to interaction with the outside world. In the case of UUIs, this outside world would be the user, interacting with the system via the created user interface. Assuming that the UI model provides the system's internal model of the interaction, a dynamic model is required. Based on this model, the system analyzes the current state and creates a UI, that projects this state to the outside world. User input received via this interface can then be fed into the model. As the dynamic model is able to model behavior over time, dependent on external stimulation, feeding the input into the model leads to a state change of the model, which is in turn reflected by the created user interface. A major aspect of any architecture utilizing models at runtime is thus the provisioning of capabilities to analyze the model and project its defined aspects to the outside world.

In the case of UUIs this requires the independent utilization of multiple interaction resources to reflect the state of the model in a distributed manner. However, as the distribution of the output might also influence the presentation, there is a feedback loop, between the used interaction resources and the information presented on these resources. An example is the utterance "as you can see on screen XY" which can only be rendered if the system is aware of this fact. As an aspect that can not be modeled at design-time (as the available devices are unknown), but influences the presentation, distribution is thus required to have information about its state affect the internal model. The same holds true for shaping, where only boundaries or constraints can be defined at design-time, but the actual shape has to be determined at runtime. Multimodality requires the reflection of the used modalities and the processing of received multimodal input according to the state of the internal model. Adaptation, altering the model to incorporate context, requires the direct and continuous projection of the model structure and state to the presented UI. Additionally, context information has to be monitored and provided on the model level, to allow its incorporation into the internal system state.

Similarly, the utilization of the model as an internal representation requires the integration of external processes into the model processing. At some point during the interaction, actions of the system, triggered by the user, are required. A home control service would e.g. have to switch on the actual light within the room. This requires the integration of external functionalities and its integration as accessible functional core within the

architecture.

The described concepts summarize the requirements for an infrastructure realizing the model-based provisioning of and interaction with UUIs at runtime. For UUIs, the discussed runtime aspects have to be supported by the runtime infrastructure, but need also to be expressed within the UI model as they are application specific information, which is addressed in the next section.

UI Concepts

Besides the architectural view, the identification of concepts that have to be covered and addressed by the underlying UI definition and thus by the used UIDL is crucial for the approach. The UIDL is required to provide means for the developer to express the anticipated interaction as well as to constrain and control the UUI features.

From the perspective of distributed and multimodal interaction, the UI description has to identify separable building blocks that can be utilized via multiple interaction resources and shaped to match resource specifics. These building blocks provide the basic interaction elements the architecture can assign to available resources. Assigning an interaction element to an IR then results in the projection of the element to the IR in form of a UI. The separation of input and output within the UIDL allows to address and combine input and output resources independently. However, the interrelation of input and output also has to be expressed to keep them connected. In combination with the goal to support multimodal interaction multiple levels of abstraction are required. This allows to express the input and output relation on one level and their separation on another. Additionally, a modality independent abstraction allows to handle the combination of different modality specific definitions on a lower abstraction level.

With the goal to utilize the created UI description at runtime, it also has to cover runtime issues like the integration of backend services and the expression of state information. The integration of backend services requires means for the developer to identify the services. At runtime the identified services then need to be called by the architecture, based on the provided design information. Utilizing the UI description as a runtime model of the anticipated interaction additionally requires state information allowing the observation of the modeled system to interconnect it with the outside world. Providing interaction states and transitions between these states as part of the model allows to convey the model state to the user and to utilize user input to stimulate the model to perform state changes according to the provided input. Integrating the means to reflect this state

supports its explicit observation and alteration. Covering not only static aspects, but also the UI behavior and evolution over time allows the modeling of dynamic systems. Context information has to be modeled to define possible adaptations. At runtime, this requires the provisioning of live context information that can be incorporated into the interaction process and reflected within the state transitions of the model.

The concepts identified in this section summarize the UUI specific requirements to the user interface description language and thus the model underlying the user interface at design and runtime. A focus has been put on the runtime aspects required for the handling of UUIs. Additional basic aspects like grouping (TERESA and UsiXML), eventing (TERESA and UIML), abstract and concrete interactor details (TERESA, UIML and UsiXML), voice interactors (UsiXML), graphical interactors and design information (TERESA, UIML and UsiXML), have already been addressed in the presented UIDLs and have thus not been explicitly discussed here.

3.5.3. Summary

While the current state of the art shows great progress in the possibilities to create flexible user interfaces for future applications, handling the identified shortcoming is crucial for the success of Ubiquitous User Interfaces and the interaction in and with smart environments. The identified requirements address the need for a framework that closes the gap between user interface description languages and runtime architectures. They serve as basis for the determination of an architecture and a suitable UIDL to express Ubiquitous User Interfaces and summarize the needs specific to Ubiquitous User Interfaces. The described concepts have to be addressed by the underlying models and supported by the developed runtime architecture. A strong focus has been put on the identification of aspects relevant for the combined integration of the UUI features, as there is a close coupling between the different features, aggravating the consideration of isolated aspects.

Based on the identified requirements, the remainder of this work focuses on the realization of the infrastructure required to realize Ubiquitous User Interfaces for smart environments. Three major building blocks are considered to reach this goal:

1. an architecture, capable of interpreting user interface models and handling the desired interaction with the user,
2. the identification of common UIDL (meta-)concepts, required to utilize model-based user interface descriptions at runtime,

3. the definition of a reference UIDL to express the concepts of UUIs, based on the underlying UIDL meta-concepts.

The close integration of the architecture, with the general concepts allows its utilization for the interpretation of various user interface models, integrating the provided concepts. However, a reference implementation of the architecture to create Ubiquitous User Interfaces, based on the defined reference models is also provided. While the integration of the development process and methodological aspects is an important factor, the remainder of this work concentrates on the runtime aspects.

Based on the identified requirements, the next section illustrates the implications of the identified concepts to the architecture, followed by the introduction of executable models, providing the foundations for the UIDL meta-concepts in chapter 4. A set of reference models, addressing the application specific issues are then presented in chapter 5.

4. Executable UI Models

Aiming at the utilization of user interface models at runtime, based on earlier work (Lehmann, 2008), this chapter identifies common building blocks to facilitate this approach. It presents a meta-metamodel, allowing the development of metamodels, which express dynamics and behavior within a stateful model and combine syntax and semantics.

Dynamic executable models allow the combination of static and dynamic elements with the related execution logic, defining the dynamic behavior in one single model. This makes the models complete in the sense that they have "everything required to produce a desired functionality of a single problem domain" (Mellor, 2004). They provide the capabilities to express static elements as well as behavior and evolution of the system in one single model. Executable models run and have similar properties as program code. In contrast to code however, executable models provide a domain-specific level of abstraction which greatly simplifies the communication with the user or customer. Combining the idea of executable models with dynamic elements as part of the model gives the model an observable and manipulable state. Besides the initial state of a system and the processing logic, dynamic executable models also make the model elements that change over time explicit and support the investigation of the state of the execution at any point in time. In summary, dynamic executable models can be described as models that provide a complete view of the system under study over time. Main advantages of the approach based on the availability of design information at runtime are:

- the direct interpretation of models, as the interpretation/execution logic is part of the metamodel,
- explicit access to dynamic runtime information as this is also part of the model,
- alteration of static and dynamic aspects of the model even at runtime,
- well-defined boundaries of the model and means to access elements outside of these boundaries,

- the possibility to connect models and support the runtime data exchange between them.

The meta-metamodel (M3 layer of the MOF Architecture), introduced in the following, describes the common building blocks of these executable models formally. The description of its syntax and semantic is complemented by the description of well-defined access to elements outside of the scope of the model. Additionally, a mapping metamodel allows the definition of relations between multiple models. Being itself an executable model, the mapping metamodel is defined as the glue between models, which allows the exchange of information between different models at runtime. A summary concludes this chapter and motivates the set of reference models illustrating the approach and its utilization in the context of Ubiquitous User Interfaces in chapter 5.

4.1. The Meta-Metamodel

The meta-metamodel of executable models conceptually describes the common building blocks shared by all executable models and identifies the concepts models have to obey to be executable at runtime. It combines the initial state of the system, the dynamic model elements that change over time and the processing logic in one model. This leads to the need to clearly distinguish definition-, situation- and executable elements as shown in table 4.1. A similar classification has also been identified by Breton and Bézivin (2001).

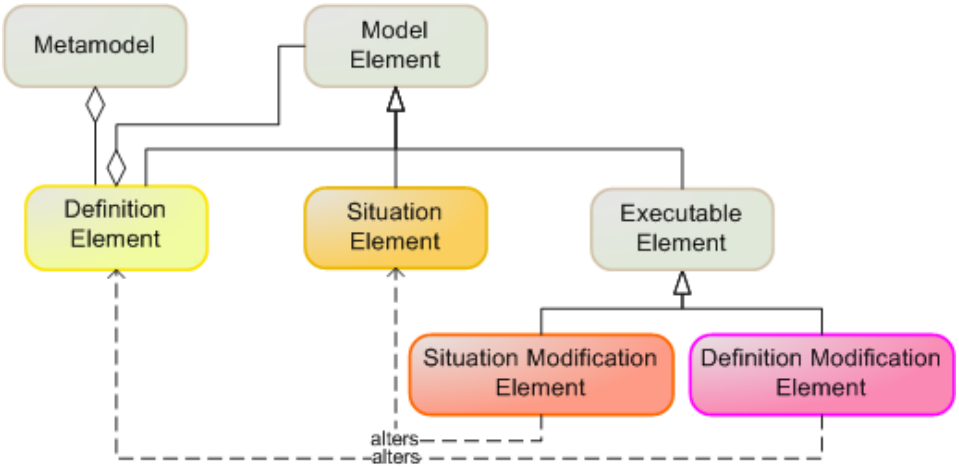

Figure 4.1.: Meta-Metamodel of Dynamic Executable Models

Element	Description
Definition Elements	Definition Elements define the static structure of the model and thus denote the constant elements that do not change over time. Definition elements are defined by the designer and represent the constants of the model, invariant over time.
Situation Elements	Situation Elements define the current state of the model and thus identify those elements that do change over time. Situation elements are changed by the processing logic of the application when making a transition from one state to another one. Any change to a situation element can trigger an execution element.
Executable Elements	Executable Elements define the execution logic and thus the semantic of the models. They specify the dynamic of the models and thus the logic altering the models at runtime. This alteration either changes the state of the model, in which case situation elements are affected, or the defined structure of the model, in which case definition elements are affected. Two types of executable elements are distinguished: Situation Modification Elements and Definition Modification Elements.
Situation Modification Elements	Situation Modification Elements (SMEs) define the interpretation process of the model, in other words the transitions from one state to another. In this sense these elements are procedures or actions altering the situation elements of a model. Situation modification elements also provide the entry points for data exchange with entities outside of the model. Defining execution elements as part of the model allows the incorporation of semantic information and the interpretation process as part of the model itself and thus ensures consistency and an unambiguous interpretation. This approach makes an executable model complete and self-contained.
Definition Modification Elements	Definition Modification Elements (DMEs) define manipulation processes altering the model and are specializations of execution elements. As executable elements they, similarly to situation modification elements, alter the model, but are not limited to situation elements. They do not define internal state transitions of the model, but changes to the structure of the model definition and can thus alter any model element. Definition modifications are part of the metamodel and have to ensure that any performed alteration is legal with respect to this metamodel.

Table 4.1.: The main building blocks of executable models.

The difference between the Situation Modification Elements and the Definition Modification Elements is important. While the former change the run time state of the model,

the latter modify any model elements and are able to create or alter any kind of model structure. DMEs are theoretically able to revise decisions of the designer and can be used for adaptation purposes. Additionally, they make it possible to define the whole construction of correct models as part of the metamodel and to ensure that tools are only able to create correct models. Distinguishing the different elements leads to the conceptual meta-metamodel of dynamic executable models depicted in figure 4.1.

In summary, the described meta-metamodel provides a formal view of executable models and summarizes the common concepts the models are based on. The separation of the elements provides clear boundaries for the designer, only modifying definition elements and the runtime system, altering situation elements. A definition element as the basic element finally aggregates situation- and executable elements that describe and change situations for the given definition element. Using such models in a prescriptive way (constructive rather than descriptive modeling) allows defining systems that evolve over time, reason about the past and predict future behavior. Thus, dynamic models can also be used to build self-adaptive applications, similar to approaches described in Rohr et al. (2006); Schneider and Becker (2008); Sottet et al. (2007b). In this context, the models monitor system and environment to calculate adaptation and behavior of the UI.

From this definition of the meta-metamodel, two issues arise. One is the integration of objects and services outside of the scope of the model into the execution logic. While this is technically possible, e.g. via a Java/EMF implementation as in our approach, there is a need for well-defined concepts to do so, which will be discussed in the next section. The other issue is the connection of multiple models, especially at runtime, which is discussed in section 4.3.

4.2. Execution Logic

Creating executable models requires to express execution logic as part of the model. Additionally, runtime execution interconnects the model with external instances outside of the scope of the model. This is required to incorporate external information and affect the outside world. This leads to three types of execution logic:

intra-model logic is part of the model and expresses the internal execution semantics of the modeled domain.

inter-model logic is related to the model, but does only indirectly affect the modeled domain. It is outside of the scope of the model, but related in a wider sense.

external model logic interconnects the model with real-world entities, that have to be integrated into the model, but are not directly part of it.

The three logic types are described in the following sections.

4.2.1. Intra-Model Logic

The integration of executable elements requires execution logic to be part of the model. While models usually aim at the derivation of code from the designed models, executable models aim at the direct execution and the specification of their internal behavior. Intra-model logic thus expresses how the model is transformed from one state into another.

Expressing this in a formally modeled manner can be suitable in some cases, e.g. to formally ensure its correctness. Some approaches thus aim at the modeling of execution logic as part of the model (see e.g. (Muller et al., 2005)). In other cases, a formal definition of the execution logic might produce some overhead. It can thus be suitable to simply integrate a programming language and apply some restrictions on its usage to express the internal logic of the model.

In this work, the algorithms and semantic to compute runtime situations and structural modifications are not formally modeled, but it is assumed, that there is a language allowing to express them in conjunction with the model. A similar approach can be observed in the connection of EMF and Java.

4.2.2. Inter-Model Logic

In contrast to the internal state transitions and the structural modifications of the model, runtime utilization of a model also raises the need to embed the model in a larger context or system. Within this larger system, dependencies and interconnections may be observable, that are not directly part of a model, but that affect the model internals. The notion of *components* allows the definition of interpretation logic that observes one (or multiple) model(s) and provides some execution logic, based upon observed changes. Interaction with models to incorporate the execution results then happens via calls to execution elements and thus the internal logic of the model.

Similarly to object oriented programming concepts that still use imperative code as part of their objects, this allows the integration of objects or imperative program code to express information beyond the scope of the model. The main contrast to the executable

elements of a model is that components are not limited by the boundaries of a single model. They can read information from other models and directly call external execution elements.

4.2.3. External Model Logic

Additionally, a strong focus has been put on the connection of the models and especially their semantics, to the outside world. While a design model aims at the creation of code that can be connected to any kind of external, non-modeled code, executable models have to directly incorporate such connections and handle them at runtime.

An example: Assuming, the definition of a context model, that models the context at runtime and thus has to continuously reflect the current context. This requires access of the model to real world context information, which is dynamically acquired via sensors in the environment. However, the model itself has to acquire the information and ensure that it is up-to-date and thus needs to be aware of the outside system. A problem is the definition of the border of this model.

Any executable model has to assure, that it does not depend on logic (or program code) outside of the model. Assuming, that the model is somehow aware of the sensors and of the information they deliver, it could also express which sensors deliver which information and how to acquire this information. However, at some point a border between what is modeled and what is outside of the model is required. In this example outside of the model is e.g. the implementation of the API of the sensor or a web service call to acquire information from this API. This raises the need to define how information can be brought into the model and how information can be requested from the outside by the execution logic of the model.

While the first issue to bring information into the model can be solved by making Executable Elements available to the outside world (e.g. context provider or modeling tools), the second issue requires referencing external code from within the model and thus blurs the models boundaries. To cope with this issue, the concept of a proxy element is introduced.

A proxy definition element identifies processes outside of the model and provides an internal representation of these processes without requiring to directly reference them. Based on this proxy, models can control the external processes and communicate with them. Some required elements of the proxy have been identified in table 4.2.

Element	Description
processIdentifier	The processIdentifier uniquely identifies the referenced process. Using Java, this would be the class name of the Java object to instantiate. As the processIdentifier is provided by the UI developer, it is stored in the definition element at design-time.
proxyConfiguration	The proxyConfiguration is an additional definition information, holding the configuration of the external process, e.g. parameters for the constructor.
startMethod	The startMethod identifies the method to initialize the process.
stopMethod	The stopMethod identifies the method to terminate the process.
callback	The callback attribute denotes the situation modification element the process is able to call to push information into the model.
proxyReference	The proxyReference stores the reference of the instantiated process at runtime. As the reference is created at runtime this is a situation element. Storing the reference allows controlling the process if it has a longer lifespan.

Table 4.2.: Main attributes of the proxy definition element, connecting model and outside world.

Based on these six elements, a proxy element can be defined as part of any metamodel allowing to identify and incorporate processes outside of the model. This allows the definition of clear boundaries of the model, while still being able to express connections to the outside (unmodeled) world. Additionally, to the proxy element, which connects unmodeled entities, the mapping metamodel presented in the next section allows to connect different models with each other and facilitate the information exchange at runtime.

4.3. The Mapping Metamodel

Building models of complex systems often requires the utilization of multiple models that describe different aspects of the system (e.g. the different abstraction layers in the Cameleon Reference Framework or the models in UML). An explicit definition of links between these models is crucial for executable models, as there should not be any implicit knowledge and thus hidden dependencies in the models. In this section a mapping metamodel is introduced, which allows to express (runtime) relations between multiple metamodels.

Located on the M2 layer, the mapping metamodel provides an extra metamodel solely for mappings, that conforms to the described meta-metamodel. The mapping metamodel

itself is executable and provides the required event hooks in the execution logic to interconnect multiple metamodels. It has been defined on the concepts of the meta-metamodel only and is thus able to connect any metamodels conforming to the meta-metamodel. This also enables to benefit from tool support and removes the problem of mappings hard-coded into the architecture, as has been advised e.g. by Puerta and Eisenstein (1999). The mapping metamodel allows the definition of the common nature of the mappings, relating metamodels elements. The mapped models do not need to be aware of their relations.

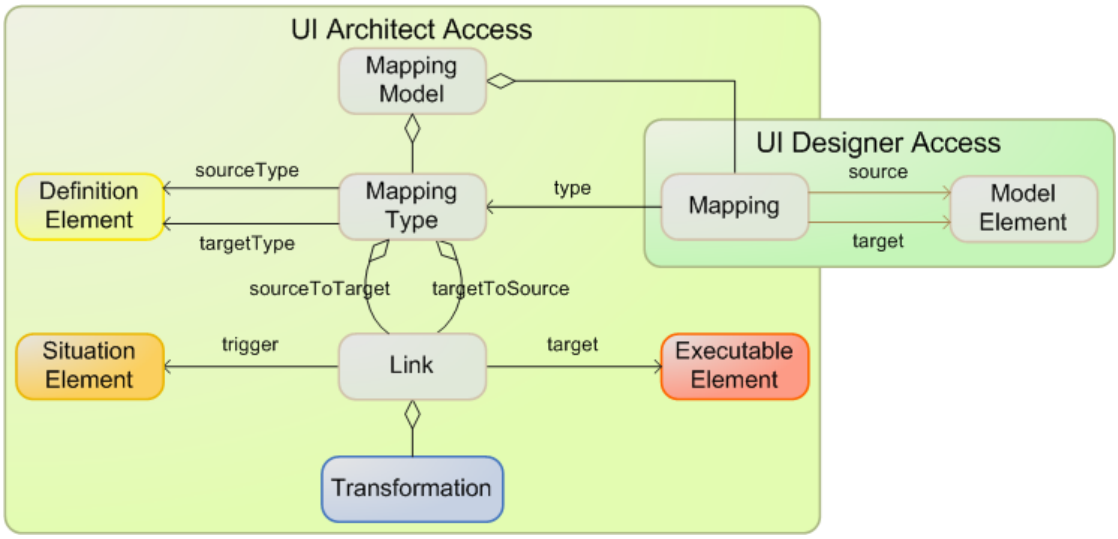

Figure 4.2.: Mapping Metamodel

An example of a mapping metamodel, consisting of a fixed set of predefined mapping types, can also be found in UsiXML described by Limbourg et al. (2004b). Sottet et al. (2006a) have defined a mapping metamodel, which can also be used to describe transformations between model elements at runtime. However, in contrast to these approaches the presented mapping model puts a stronger focus on extensibility, the specific situation at runtime and the information exchange between dynamic models. Especially interesting at runtime is the fact, that the relations can be utilized to keep models synchronized and to transport information between models. The information provided by the mappings can be used to synchronize elements if the state of the source elements changes. Mellor et al. (2004) see the main features of mappings as construction (when the target model is created from the source model) and synchronization (when data from the source model is propagated into the existing target model). The mapping model contains mappings of the latter kind. Focusing on runtime aspects, a mapping expresses the possibility to alter

an existing target model, based on changes that happen to the related source model. In contrast to the transformational view of mappings, the mappings utilized here do not transform a model into another one. Instead, they synchronize runtime data between coexisting models. Mappings connect definition elements of different models with each other. They are always triggered by situation elements and activate execution elements.

The conceptual mapping metamodel is provided in figure 4.2 and combines mapping types and mappings. It consists of the following elements:

- *Mapping types* are the main elements of the mapping metamodel, as they provide predefined types of mappings that can be used to define the actual mappings between elements on M1 layer. A mapping type is predefined by the architect of the user interface meta-models and can also serve as extension mechanism to integrate additional relations or models. It basically consists of two definition elements and well-defined links between the two.
- The *definition elements* are the source and the target of the mapping and the mapping synchronizes the runtime data between these two elements.
- The *links* consist of a situation element, an executable element and a transformation.
- The *situation element* is the trigger of a link. Whenever a situation element in a model changes, the link is triggered and the referenced execution logic is executed to synchronize the two definition elements of the mapping.
- The *executable element* is the logical target of the link, as model changes can only be performed by executable elements.
- The optional *transformation* associated with the link describes how the situation data, which activated the trigger, is transformed into (input) data needed by the target execution element in the other model. This transformation might be required, especially when models with distinct data types and structures are linked by mappings.

To simplify the usage of the model, the metamodel supports multiple links in one mapping type, as multiple situation elements (e.g. related to the same definition element) might be relevant to trigger the execution. Supporting more than one link also allows a back linking, as some mapping types might also demand two-way links.

From the designer's point of view, the initial mapping model provides a set of available mapping types with predefined logic, defined on the metamodel level. Thus to relate two

models, the user interface designer extends this initial model by creating new mappings that reference an available mapping type (see the box "UI Designer Access" in figure 4.2). To create such a mapping, the designer has to provide the specific source and target model elements to the mapping and define its type. This leads to a relation between the two elements and their synchronization according to the given execution logic.

Using the meta-metamodel the mapping metamodel could be defined independently from the concrete metamodels that mappings can be created between. Only the mapping models contain mapping types, which are not of generic nature, but specifically designed for the given metamodels. Using executable elements as target allows the definition of two variants of mappings: synchronization and construction mappings.

4.3.1. Synchronization Mappings

Synchronization mappings are (runtime relevant) mappings that have situation modification elements as their target elements. These mappings aim at the synchronization of the state of two models and facilitate the exchange of information between them. At runtime the alteration of the situation of one model can trigger a mapping and thus influence the situation of another model. E.g. could the completion of a task trigger the removal of the related presentation elements from the screen. An example is shown in figure 4.3. While the mapping type is defined on the level of the metamodel and identifies the element types, the mapping itself is defined on the model level and relates two distinct elements. As the situation element of Metamodel X is linked to the SME of Metamodel A, changing situation X in Model X triggers the execution of A -> B and thus the alteration of situation A to B.

While synchronization mappings are used to maintain a consistent runtime state, constructional mappings alter the underlying definition elements at runtime.

4.3.2. Constructional Mappings

In contrast to synchronizational mappings, the utilization of constructional mappings allows the relation of model definitions. Using this mapping ties the structure of two models together. A modification of the structure of the source model also triggers a modification of the structure of the target model. At design-time, this supports the creation of consistent systems spanning multiple models. At runtime, it addresses the need to reflect runtime alterations within multiple models. The basis for the mapping

are the definition modification elements, that allow the metamodel-conform alteration of the model definitions. Being allowed to alter any definition elements and thus any model structures of the model, DMEs are defined as part of the metamodel and operate on the metamodel structures.

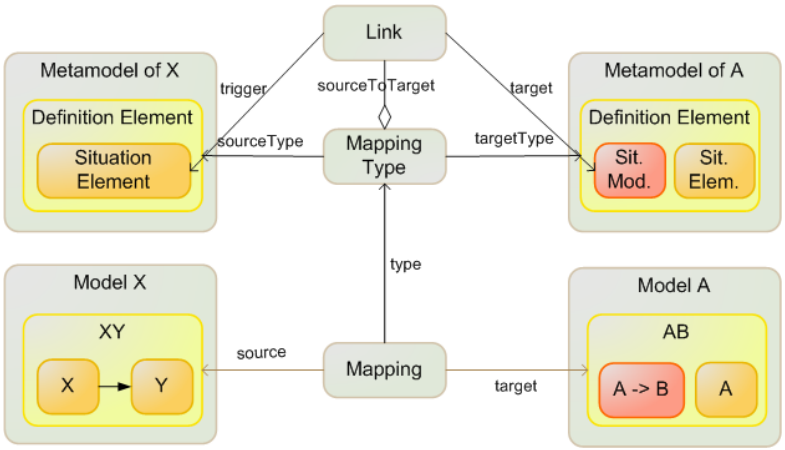

Figure 4.3.: Example for a synchronization mapping.

Taking this to the extreme, the metamodel can provide a set of definition modification elements, that allow to construct any (set of) metamodel compliant model(s), similar to production rules of grammars. At runtime, the possibility to alter the definition of the meta-model in a well-defined manner, allows to perform model adaptation according to the context of use, even across multiple models. To realize this, constructional mappings support definition modification elements as target elements. Relating a situation to a definition modification thus allows to reflect e.g. context changes, by altering the structure of the underlying model.

4.4. Summary

The executable models introduced in this section support the creation of models that define systems and their behavior over time, while also exposing all state information for manipulation and inspection. The meta-metamodel of executable models describes the building blocks of such models as definition-, situation- and executable elements. Based on these elements dynamic systems that change over time can be defined and coupled with external processes. While defining dynamic situation elements makes the state of the model explicit, the definition of execution logic as part of a metamodel also makes

the transitions between states explicit. This strongly interconnects syntax and semantic as part of the model. Bridging the model and the real world (or the derived system), external processes can be encapsulated within proxy elements to be incorporated into the execution process. More complex systems, spanning multiple models, can be built by linking multiple models together via mappings. A mapping metamodel formalizes the mechanisms to map metamodel elements and also complies to the overall meta-metamodel. Mappings between models allow the exchange of information at runtime and ensure structural consistency across models. In summary, executable models provide:

- the identification of common aspects that make (meta-)models executable
- the strong interconnection of syntax and semantics as well as runtime state and execution logic as part of a model
- well-defined external calls into the model altering the runtime state (situation) or structure (definition)
- embedding of external processes
- a realization of events between the models to exchange information via mappings

This approach sets the foundations for a UIDL, that is not only interpretable but by itself executable at runtime. Based on the defined meta-metamodel and the mapping model, a set of metamodels is introduced in the following. The metamodels specifically address identified runtime issues of multimodal user interfaces for smart environments and form the basis for the creation of a runtime architecture for UUIs.

5. User Interface Metamodels

The basis for the development of user interface descriptions and models is the availability of a modeling- or user interface description language. While the previous chapter explored common building blocks for the runtime utilization of user interface models, expressed in the meta-metamodel of executable models, this chapter applies the findings to develop a conforming set of core metamodels. These metamodels basically form a user interface description language to define Ubiquitous User Interfaces. Based on the metamodels, user interface models can be defined and executed at runtime. The goal of this chapter is not to describe another UIDL, but to illustrate concepts, suitable to extend existing approaches towards their interpretation at runtime and to illustrate the features and runtime issues of Ubiquitous User Interfaces.

A main focus is set on the interaction metamodel as the central instance for the output creation and input interpretation. Additionally, task-, domain- and service-metamodel are explored. A mapping metamodel allows to dynamically link the different models to express their interrelations. The roles of the involved metamodels can be described as follows:

1. The task metamodel (section 5.1) defines the basic workflow of the application. It separates interaction tasks, refined via the interaction model and application tasks, accomplished without user interaction by backend services via the service model.
2. The domain metamodel (section 5.2) describes the domain concepts underlying the user interface. It defines types and objects, which are referenced from tasks and thus involved in interaction tasks and service calls.
3. The service metamodel (section 5.3) results from the need to integrate backend services at runtime. It provides the means to identify the functional core, make calls to services, pass parameters, and process return values.
4. The interaction metamodel (section 5.4) refines interaction tasks and defines the actual communication with the user. The metamodel provides support for multi-modal interaction and separates the following abstraction levels:

a) An abstract interaction metamodel (section 5.4.1) aims at a modality and device independent description of interaction.

b) A concrete input metamodel (section 5.4.2) targets the definition of user input expected via specific modalities and devices.

c) A concrete output metamodel (section 5.4.3) targets the creation of output in different modalities.

5. The mapping metamodel (section 5.5) finally provides the possibility to interconnect the different models and thus ensures synchronization and information exchange between models.

In the following the different core metamodels are described in detail. Their interconnection and the runtime process, the combination of models facilitates, is described in section 5.5. For the sake of brevity the term model is used to refer to a metamodel as the whole section elaborates on the meta-level (M2).

5.1. Task Model

The task model describes the tasks the user is able to execute. It provides a high level view to the intentions of the user and the system and defines the workflow of the application. At runtime the workflow of the interaction is derived from the temporal operators of the task model. Based on the active tasks, (interactive) elements in other models are activated and deactivated to provide interaction possibilities matching the currently available tasks. Similar to TERESA and UsiXML, the task model is based on the ConcurTaskTree Notation (CTT) (Paternò, 1999), which defines interaction-, application- and user-tasks. The order of the tasks is defined by high level LOTUS operators, providing hints about the execution order of the tasks and the interdependencies between tasks. To fit the runtime approach, the CTT notation has been modified to distinguish interaction-in-, interaction-out- and application tasks (Feuerstack et al., 2007). This approach allows a clearer separation between the expected type of interaction - user input or system output - and also takes into account the need to connect to the backend to perform tasks that only the system is involved in. Additionally to the modified task types, the object references for each task have been modified allowing the distinction of objects being declared, created, read, or modified. This allows a clear classification of the object modifications performed by each task. In combination with the domain model holding the referenced

concepts, the annotation of the objects also allows the specification of an object life-cycle for the domain model. Other approaches follow a similar concept. Limbourg et al. (2004a) define actions (*start/go, stop/exit, select, choose, create, delete, modify, move, duplicate, toggle, view, monitor*) and items (*operation, container, collection, element*) the actions manipulate. Oviatt (1999) also proposes actions (*specify constraint, overlay, locate, print, scroll, control task, zoom, label, delete, query, calculate distance, modify, move, add*), which could be applied on a task level. In this work, the interaction semantics are detailed as part of the interaction model, described in section 5.4. To be able to execute the CTT-based task model the static part of the CTT metamodel has been extended with state information, needed to reflect the state of the execution in the model. The model thus introduces situation elements as attributes for each task, identifying the runtime state. Figure 5.1 shows the metamodel structure for executable task models.

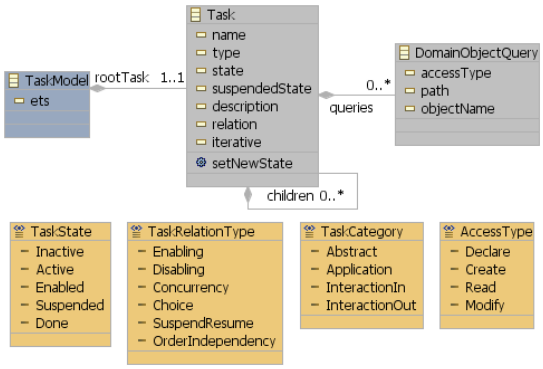

Figure 5.1.: An executable task metamodel.

As one can see in figure 5.1, every task model is comprised of a root task with a set of child-tasks. Each task is a definition element which also comprises situation elements. While name, type, description, relation (temporal relation to neighbor task) and the iterative flag are defined by the designer, state and suspended state (the last state before suspension) are annotated as situation elements as they change over time.

Based on these elements, the execution of the task model at runtime can be illustrated. The *setNewState* execution element is used to change the state of the task as well as all related child-tasks (according to their temporal relations). This allows to explicitly store the execution state of the model as part of the model. The execution starts with the root task and derives the initial Enabled Task Set (ETS). Each task in this set is at state *enabled*. Based on this set of tasks, backend services are called or user interaction is requested. Once a task from the set is completed, it is set to *done* and a new ETS

is calculated, which alters the presentation or executes additional backend services via mappings to other models.

In summary, the task model comprises the static definition of tasks, their temporal relations, the current state of each task and thus of the whole process, and the execution logic, that defines the transition from one state to the next in a single model. However, to support the creation of a user interface based on this information, additional models are needed. While the interaction model (section 5.4) and the service model (section 5.3) refine the frontend and backend interaction for each task, the domain model described next denotes the objects presented and manipulated during the interaction.

5.2. Domain Model

The domain model is required to model the classes representing the application domain from the user interface point of view. Similar to the task model, TERESA or UsiXML integrate a domain model. Stocq and Vanderdonckt (2004) show how domain model and task model can be related to create user interfaces, Pribeanu (2007) describes a set of mapping rules between task and domain model. Earlier approaches like GENIUS (Janssen et al., 1993) and TRIDENT (Bodart et al., 1995) also use entity-relationship models related to a database. As in the more recent approaches, the utilized domain model allows to link defined classes of domain objects to tasks that define actions (*declare, create, read, modify*) performed on objects of the defined classes. It additionally covers the need to handle instances of the described objects at runtime. Figure 5.2 shows the structure of this model.

At design-time, the domain model defines the structure of the objects related to the different models in form of a class diagram. Classes can reference each other and are structured in packages. Each domain object is of a certain class, combined of subclasses and attributes. This is important to check for consistency and to guarantee that the objects instantiating the classes at runtime regard the structure of the model. The model defines the known instances of each class. This guarantees that object references in other models refer to the same object. Thus all objects required by the application as well as their structure are known at design-time.

At runtime, the domain model plays a major role as storage space for domain objects. Any object created through user interaction or received as result from a service call is stored as situation within the domain model as value of the domain objects. This makes all domain data continuously available at a central location. In this sense, the definition

elements of the object define the types and known objects, the situation elements of the model hold the instances of the objects. Other models, referencing types or objects from the domain model, can retrieve instances of objects from the model or be synchronized with object instances if they are altered via mappings.

Objects stored in the domain model serve as interaction objects providing dynamic information for input and output possibilities. Additionally, the domain model holds objects that are passed to backend services during service calls or return values received from these services via mappings.

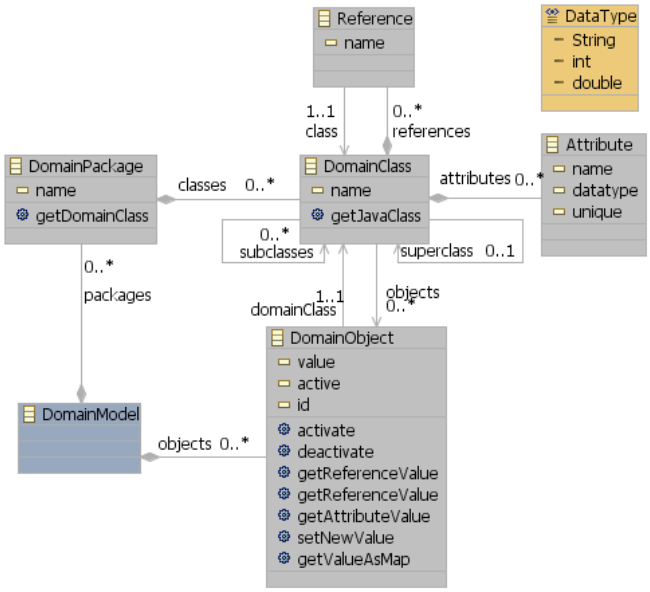

Figure 5.2.: An executable domain metamodel.

5.3. Service Model

As described in 5.1, the task model comprises application tasks identifying tasks the system is able to perform without any kind of user interaction. This kind of task is especially relevant for the interpretation of the task tree at runtime to integrate the referenced backend services. The service model, depicted in 5.3 provides the connection of the application tasks to the backend services. It defines service adapters and service calls, which allow to call any referenced backend service. Similar models have been utilized in the DynaMo-AID system or SmartKom as well as in the ICARE system, utilizing the Arch concept (Bass et al., 1992) of an adapter for the functional core. Ghorbel et al.

(2006) present an approach to model assistive services for smart environments. While the latter aims at the dynamic discovery and utilization of services, similarly to the first three approaches, the presented service model aims at the referencing of services to call directly. Dynamic discovery, composition, etc. could then occur within the called service.

In contrast to other approaches, the utilized service model does not aim at the modeling of the related services, but focuses on the possibility to make calls to existing services. An additional main difference to other approaches is the utilization of the service model as an executable model. Service adapters, realized as proxy elements, allow the connection of multiple technologies to perform the actual service calls and provide execute methods, that translate the service call to the technology the adapter represents. Technology specific adapters can be integrated by refining the general concept. Each service call references an adapter and is thus specific for a given technology, which makes it easy to change the technology underlying a service call. Parameters allow to pass elements from the domain model to the triggered service, properties allow an additional configuration e.g. of the utilized adapter. This model allows the comprises all elements needed to execute calls to external services, but does not model the services or their underlying semantics yet.

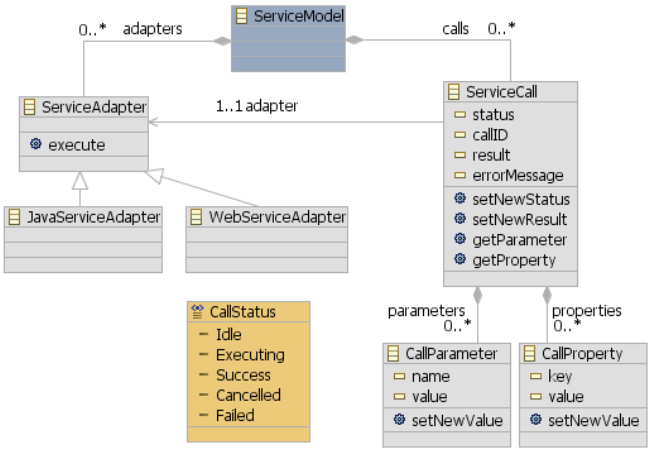

Figure 5.3.: Service Model

Service calls are referenced by application tasks in the task tree and allow to call backend services. They become active immediately when the application task becomes enabled, which allows the realization of asynchronous service calls via application tasks, parallel to interaction tasks. Required parameters are related to the domain model via mappings, which allows the exchange of domain objects between the services and the user interface.

After executing the service call, the service can also provide results updating stored domain objects accordingly. This can result in user interface updates of the interaction elements, that are related to the updated domain object. These user interface updates are handled by the interaction models as described in the next section. The sequence, a service call is realized by, is presented in figure 5.4.

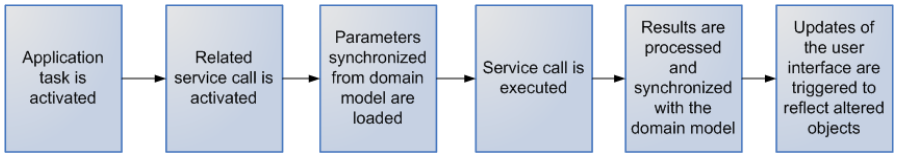

Figure 5.4.: Service Call Sequence.

5.4. Interaction Modeling

To model UUIs, the interaction model described in this section aims at the definition of modality independent abstract interaction and its refinement as modality specific concrete interaction model. It focuses on the description of the runtime utilization of the defined models.

The device and modality independent definition of the anticipated interaction in an abstract form and the specialization of the described interaction in a more concrete model have been in the focus of different UIDLs, including UIML, UsiXML and TERESA. The usual approach here is to build the abstract interaction model (e.g. based on the task model) and then to refine the abstract interaction with concrete representations for different target modalities and platforms at design-time.

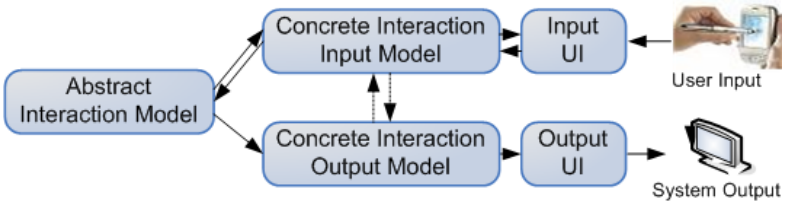

Figure 5.5.: Basic Structure of the Interaction Model, separating an abstract and a concrete level. Based on an abstract interaction definition, concrete input and output are the basis for the creation of input and output UIs. Information exchange at runtime is denoted by arrows.

In contrast, the approach presented in this section, assumes that the abstract and con-

crete model exist in conjunction at runtime, linked by mappings. Common elements of the existing UIDLs have been incorporated into executable interaction models to illustrate the approach. Figure 5.5 shows the underlying structure. The interaction model is split into three parts: an abstract interaction model, a concrete input model and a concrete output model. While the abstract model allows the modality independent definition of the anticipated interaction, the concrete models provide a definition in terms of input styles and user interface elements. Input and output elements are separated to support the dynamic combinations of multiple interaction resources. The combination of the three models describes the interaction from a system point of view, forming the mental model of the system in terms of possible output and expected user input. All three models are executable in the sense, that they encapsulate state information and allow the activation and deactivation of the interaction elements. Input can be processed and output can be defined accordingly. In the following the models and their relation are described in detail.

5.4.1. Abstract Interaction Model

The abstract interaction model (figure 5.6) describes the anticipated interaction in an abstract, device and modality independent way. Its purpose is the provisioning of a common abstraction layer, that all different modalities and interaction technologies adhere to. The model does not specify the workflow of the application, but refines more abstract elements like tasks. An abstract interactor can be seen as a specification of the interaction concepts of a task. The temporal relations as well as a possible hierarchy between interactors are expressed externally in the task model. The model defines five interaction elements that determine the basic types of interaction, similar to UsiXML and TERESA. Modality independence is thereby provided, as the elements do not refer to any modality-specific elements (buttons, voice prompts, gestures, ...). Table 5.1 describes the available elements, also shown in figure 5.6. The additional *DataSlot* element provides the connection to domain data (and thus the domain model), serving as reference element for application dependent data at runtime. It provides the dynamic information, the abstract interaction is populated with (e.g. the elements of a list the user has to choose from). As Figure 5.6 visualizes some abstract interactors have relations to *DataSlots*. Each *DataSlot* represents a data object that is either read or written during the interaction. In case of the *FreeInput* interactor the *newInput DataSlot* holds the new data provided by the user, while the *oldInput DataSlot* can optionally hold the previously provided input. This allows creating and modifying domain data. Similarly,

DataSlots allow populating lists of alternatives the user has to choose from with domain data as well as the storage of the user's selections. The *OutputOnly* interactor can be connected to multiple *DataSlots* to make domain data perceivable to the user. Finally, *Command* interactions are not connected to any data slots, as they are not related to application data, but provide access to fixed application functions.

Element	Description
OutputOnly	OutputOnly, identifies interaction where the computer presents the user information without requiring feedback or input. For example information as images or text.
FreeInput	FreeInput, denotes that the user inputs (unstructured) data into the system. Examples of FreeInput interactions are providing a name or a password.
Command	Command, is used when the user sends a signal to the system ordering it to perform an action. Command interactions are often referred to as navigation, since they typically cause a change of state in the dialog between the human and the system. Examples of Command interactions are "OK" and "Cancel" buttons or menus.
Choice	Choice, provides a selection to the user and denotes the possibility to choose one or several options from a list of elements.
ComplexInteractor	ComplexInteractor, allows the aggregation of multiple abstract interaction objects into a more complex type. This provides logical grouping facilities and allows the definition of temporal relations.

Table 5.1.: The abstract interaction elements of the abstract interaction model.

From the system perspective, these interactors can be identified as the basic interaction elements behind a broad range of widgets like menus, buttons, lists of any form, free text input, text and image output, etc. The definition of the listed elements is modality independent and covers the utilization at runtime by providing the basic interaction elements ready to receive input from the user and adapt the related output accordingly. This ensures the availability of information about the interaction possibilities as well as the currently expected interaction, e.g. for better adaptation and multimodal input processing.

At runtime, the state of each abstract interactor defines the currently available input and output capabilities of the system. The underlying precondition is that the set of currently active interactors (the workflow) is determined by another instance, i.e. another model

(e.g. the task model). This allows the selection of the currently active interaction elements and provides the means to process user input and to create the system. Activating and deactivating interactors is possible using the state attribute of each interactor. Four states are supported as shown in table 5.2

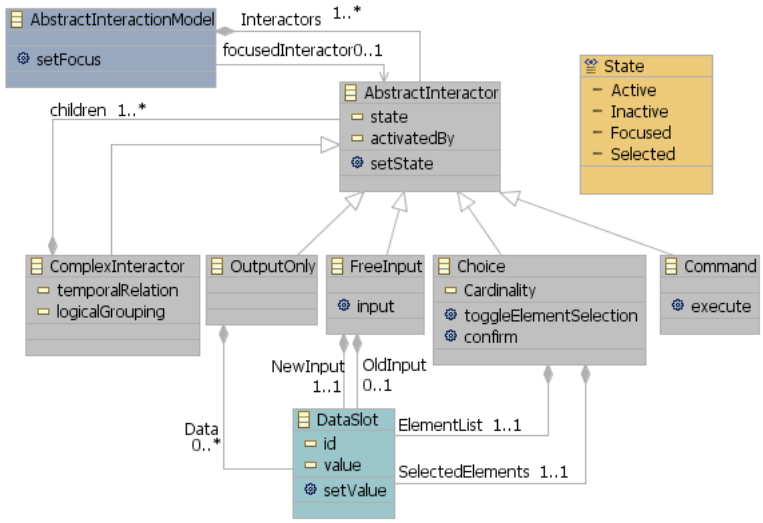

Figure 5.6.: Abstract Interactors of the Interaction Model

State	Description
Inactive	The interactor is not activated and cannot be interacted with.
Active	The interactor is perceivable/usable by the user.
Focused	An interactor reaches the state *Focused* when the user directs attention to it (e.g. by moving the mouse pointer over the element).
Selected	A focused interactor becomes *Selected* when the user explicitly interacts with it. The state *Selected* is only momentary, just as the user's interaction is. Therefore, after being selected the interactor instantly returns into the state *Focused*.

Table 5.2.: The four runtime states of the abstract interactors.

All interactors provide hooks in form of execution elements to influence their state. Each object provides the *setState* execution element, that allows the alteration of the state of the element (and the *state* situation element). Additionally, the *execute* method of

a *Command*, *confirm* and *toggleElementSelection* of a *List* or the *input* element of a *FreeInput* element provide hooks to trigger additional actions within the model.

This approach allows the selection of the active interaction elements based on the current state of the application and explicitly holds the state of the interaction. Additional execution elements allow further interaction with the model. The combination with the concrete input and output models allow the relation of abstract elements to more concrete interaction definitions as described in the next section.

5.4.2. Concrete Input Model

In contrast to the abstract interaction model, the concrete interaction model provides a more specific description of the interaction. While the abstract interactors combine input and output, the concrete representation separates the two. The concrete input model allows the definition of possible inputs related to the abstract interaction objects, independently from the presentation. This approach supports the distribution of the interactors to different devices and the free combination of multiple input resources to enable different modalities and interaction styles. The utilized concrete interactors are designed to conceptually match the supported interaction technologies and simplify the connection of new interaction resources by keeping the implementation efforts as low as possible.

Figure 5.7 shows the classification of *ConcreteInputInteractors*. The interaction elements aim to support as many interaction resources as possible and therefore still abstract from the specific properties of the interaction resources. The model distinguishes the input types listed in table 5.3.

The additional definition of the CARE properties for complex interactors allows to express complementarity, assignment, redundancy or equivalence of different input definitions. This way, the relations of multimodal input can be defined by interaction elements. A similar approach has been proposed within the ICARE platform (Bouchet et al., 2004) in form of the ICARE software components. In contrast to ICARE however, the components here are embedded in an interaction model and integrated in a model-based approach. Table 5.4 explains the supported properties.

In combination with these properties, the described elements provide the modality specific input elements to define multimodal input, retrievable from the different interaction resources. Additional predefined input components help to structure reoccurring interaction, that is especially interesting for a generic control of graphical user interfaces.

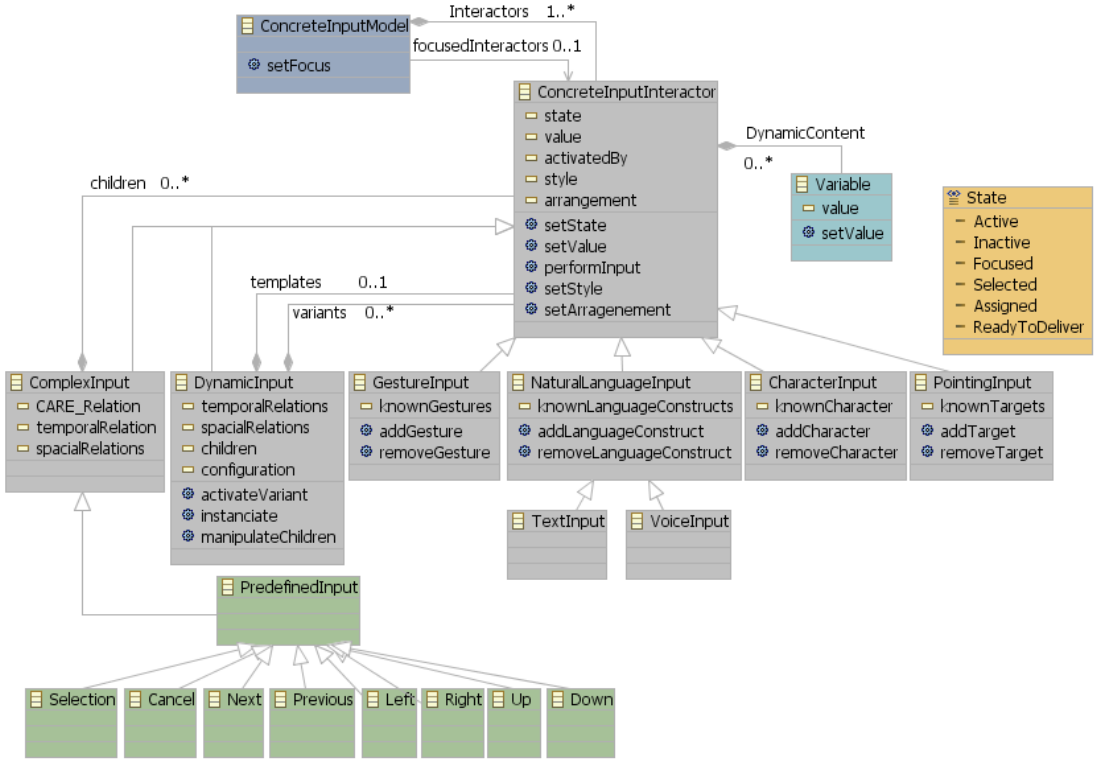

Figure 5.7.: The concrete input interactors that are considered for the definition of multimodal input.

Element	Description
GestureInput	*GestureInput*, supports the definition of gestures that can be performed to interact with an abstract interaction object. Gestures can e.g. be pen drawn gestures as well as a mouse gesture or even a mouse click.
NaturalLanguageInput	*NaturalLanguageInput*, allows the incorporation of natural language in form of *Voice-* or *TextInput* which can be used to assign utterances or keywords to the related abstract interaction elements.
CharacterInput	*CharacterInput*, provides a stream of characters produced by a keyboard or a similar device. Each character presents a single input event that can be related to an abstract interaction element.

Element	Description
PointingInput	*PointingInput*, allows the incorporation of continuous pointing information. While a stream of gestures can be segmented into distinguishable gestures, continuous pointing creates a stream of coordinates with no indication, how to segment it. An extra input type is defined, supporting continuous pointing, which provides the ability to query the pointing modality at a given moment e.g. to resolve a deictic reference.
DynamicInteractor	*DynamicInteractors*, allow the utilization of any of the above interactors as a template to dynamically create interactors at runtime. This addresses cases, in which the user interface and the underlying interaction are dynamically determined by content of the domain model. On the one hand the utilization of templates addresses the need to create input elements that vary in number and type (e.g. three vs five voice commands with key words dynamically taken from the domain model). On the other hand the activation of different variants based on information from other models allows to consider multiple situations known at design-time (e.g. a gesture instead of voice). In this case, the designer creates multiple variants of the user interface at design-time and the most appropriate variant is activated via a mapping at runtime. Selection and creation rules can be defined as configuration for the specific interactor.
ComplexInput	*ComplexInput*, finally allows the combination of any of the simple input elements described above. A complex element thereby allows the specification of temporal or spatial constraints to relate the aggregated elements. This allows the creation of input interactors defining a combination of multiple input elements that can even be assigned to different modalities.

Table 5.3.: The interactors supported by the concrete input model.

Property	Description
Complementarity	Complementarity of modalities is expressed by the utilization of complex input elements, that combine multiple inputs from different modalities. The connected simple input interactors then have to be used in combination to provide the expected input. Their temporal (or spatial) relations and fusion constraints can be specified by the complex interactor.
Assignment	Assignment of an interaction to a distinct modality is addressed by the provisioning of a single concrete input element. This is thus the standard case, even if no complex interactor has been specified and only one modality is supported for the interaction.
Redundancy	Redundancy can be realized by providing multiple concrete interaction elements for different modalities, that support the same interaction means. The given elements are then used in parallel, to provide redundant input possibilities, which means all expected input has to be provided by the user. This allows e.g. the provisioning of more robust interaction techniques in critical applications.
Equivalence	Equivalence allows the alternative utilization of different interaction techniques, that all serve the same interaction purpose and provide the same interaction means. In contrast to redundancy, only one of the available interactions has to be performed to complete the interaction.

Table 5.4.: The CARE properties supported by the concrete input model.

The following predefined input has been distinguished and can be configured via various modalities:

- *Select*, to perform the selection of any element.
- *Cancel*, to abort any action or e.g. unselect an element.
- *Next and Previous*, to allow 1D navigation e.g. within a list.
- *Up, Down, Left, Right*, to allow 2D navigation e.g. between spatial arranged widgets.

The predefined interactions provide reusable elements that can be either generically defined or overridden with specific interactions, depending on the application. However, the main advantage of the open and device independent definition of the input capabilities for each interaction is the maximized support for various types of devices and modalities. While for output the application developer decides which widgets to use for which modalities, for input this decision is made by the user. The goal is thus to support a maximum of input capabilities that can be used in conjunction (parallel or alternatively) with a minimum effort for the application developer.

At runtime, the selection of the currently active interaction elements is determined by the activated abstract interactors which are mapped to the input interactors. Activation is again performed via the setState execution element. While *inactive*, *active* (triggered by the activation of task and thus abstract interactor), *focused* (if the interactor is focused by the user) and *selected* (if the user interacts with the interactor) are used similar to the abstract model, *assigned* and *readyToDeliver* have been introduced as additional states. They have a strong focus on the runtime utilization of the model and are explained in table 5.5.

State	Description
assigned	The state assigned is reached after the interactor has been assigned to an interaction resource, e.g. by the distribution. In this state, interaction resource specific adaptations can be performed.
readyToDeliver	The readyToDeliver state is assigned after any adaptations have been completed and the interactor is ready to be delivered to the assigned interaction resource.

Table 5.5.: Additional states of concrete input elements.

At runtime, each concrete interaction element (and thus the state of the model) can be manipulated using the specified execution elements to reflect user input or changes to the system state. The main elements all input interactors support are:

- *setState*, to set the state of the input element. While this is mainly used to synchronize the states of abstract and concrete interaction objects, setting the state also occurs to signal interaction. Selecting an element e.g. sets the state attribute to *Selected*. This also applies for focus and unfocus, which is also signaled through the state.
- *setValue*, to alter the *value* situation element to set the value produced by the cur-

rent input. Based on the produced value, different mappings to execution elements of the other models can be triggered.

Based on the described elements, the concrete input model allows the specification and processing of various types of multimodal input expected from the user. After introducing the concrete output model in the next section, the relation of abstract to concrete interaction elements and the interrelation of concrete input and output is described in detail in section 5.4.4.

5.4.3. Concrete Output Model

Similarly to the concrete input model, the concrete output model refines the abstract interaction model in terms of output presentation means. Each abstract interactor can thus be mapped to input capabilities (*InputInteractors*) as well as to a perceivable representation (*OutputInteractor*s). An exception is OutputOnly which is only related to output interactors. For each abstract interactor there is a separate type of presentation to address the distinct properties and functionalities of the interactor via specific mappings.

Figure 5.8 shows the concrete output model with graphic and voice as main modalities. Additionally, more limited modalities like blinking lights or haptic feedback (vibration) can be considered using signal output. The entailed types of output interactors are described in table 5.6.

The combination and structuring of modalities in the complex output interactors is again supported by CARE-based relations as listed in table 5.4. In the case of output, complementarity is used to combine multiple output modalities, assignment again relates a distinct modality to the output, redundancy of output has the goal to present the information in as many of the specified modalities as possible, and equivalence leads to the selection of the currently most suitable of the available output interactors.

To realize scenarios of any complexity, complex interactors can be nested and hierarchically structured allowing e.g. to define a presentation combining multiple complementary modalities that is equivalent to a presentation only using voice output. An additional structure to the otherwise flat collection of concrete output elements can be given by the *group* element. Expressing the interrelation of interactors, that are not otherwise connected, it can be used to combine e.g. graphical elements to a group or ensure that related elements are not scattered across interaction resources. The meaning of the group depends on its properties, with the separable attribute explicitly configuring if the group can be broken up or not.

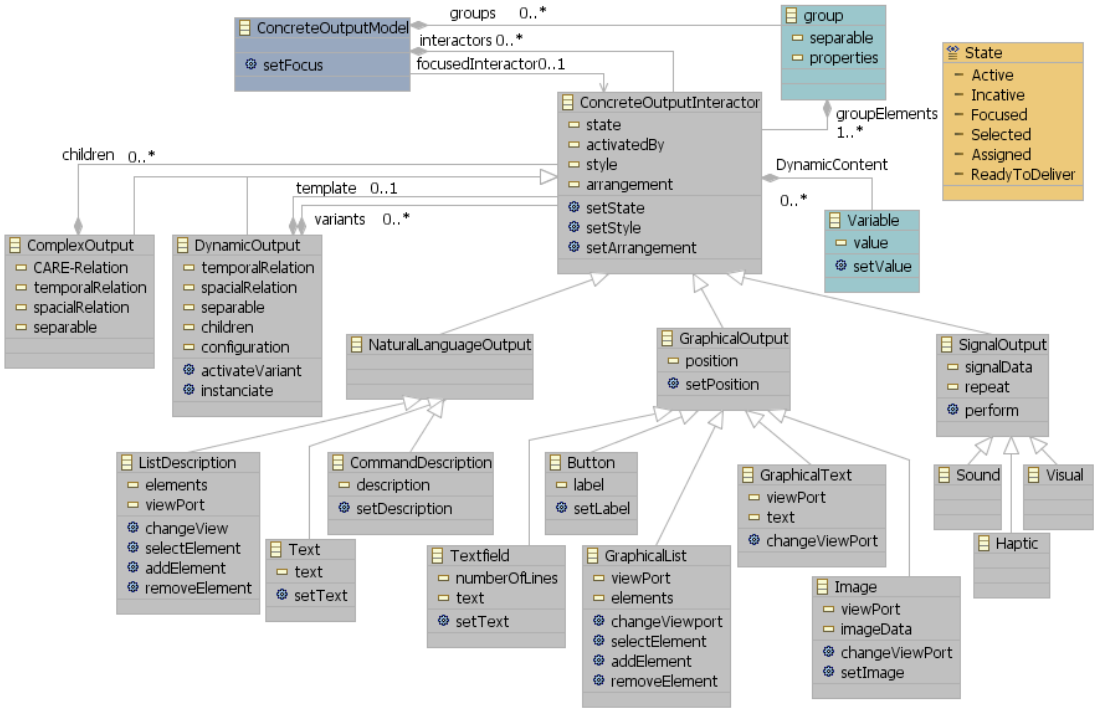

Figure 5.8.: The concrete output model with interactors separated into NaturalLanguageOutput, GraphicalOutput and more simple Signals.

At runtime the state of the model is again expressed via the state of the interactors. This allows the selection of the currently active output elements by activating and deactivating interactors to alter the presentation. The utilization of executable models allows the realization of execution logic for each of the output interactors. This is e.g. reflected by the view port specified for some graphical elements. Altering the view port aims at an update of the presentation and can e.g. be utilized to realize scrolling within a list. However, as the manipulation of the viewport is an output specific interaction it is usually not defined within the abstract interaction model. Realizing the desired scrolling effect thus requires the specification of additional input elements that are only necessary in combination with specific output elements. More details about this connection between concrete input and concrete output as well as their relation to the abstract interaction model are given in the next section.

Element	Description
NaturalLanguageOutput	*NaturalLanguageOutput*, allows to define one-dimensional voice or text output. This allow e.g. to specify a voice description that is read out, when an abstract interaction element changes its state. It can be a plain text that is read out or a description of a command that is incorporated into the voice dialog accordingly. Additionally, complex voice output can be created combining multiple concrete elements. While figure 5.8 only shows a basic set of interactors, a more detailed specification as e.g. provided in (Stanciulescu, 2008) can be easily integrated to support more fine grained voice interaction.
GraphicalOutput	*GraphicalOutput* addresses two dimensional modalities providing graphic capabilities. It provides the possibility to assign graphical presentations in form of images, buttons, text and text fields to abstract interaction objects. These objects are also intended to guide input in a related GUI, by showing the required interactors.
SignalOutput	*SignalOutput* summarizes three interaction modalities with a rather low bandwidth. While still important e.g. for notifications or direct feedback, sounds, haptic feedback like e.g. vibrations and and limited visual notifications are not capable of transporting a large amount of information.
DynamicInteractor	*DynamicInteractors* address the dynamic creation of multiple output (as has also been described for input). While e.g. a graphical list output has to take care of the presentation of all related list items, the DynamicInteractor again serves as container for other dynamically created elements.
ComplexOutput	*ComplexOutput*, allows the combination of any of the simple output elements described above. The complex element specifies temporal or spatial constraints to relate the aggregated elements, which allows the creation of output interactors structuring multiple elements to a complex construct. While spatial constraints allow the structuring of graphical objects, temporal constraints could be added for voice coordination or animations. Complex elements can also complementary span multiple modalities, which is again realized by incorporating the CARE properties. Additionally, the separable attribute identifies if and how the included elements can be split across interaction resources.

Table 5.6.: The interactors supported by the concrete output model.

5.4.4. Interrelations between Interaction Elements

The separation of the three parts of the interaction model addresses the need to define interaction on multiple levels of abstraction. While this is usually utilized to simplify the development process, it can also be used at runtime to facilitate a loose coupling of multimodal interaction and keep a more abstract definition of the interaction goals. This allows to separate interaction that is only relevant for the handling of the user interface (e.g. scrolling) and interaction that is application relevant (e.g. activating a command). Additionally, the separation of input and output allows to freely combine interaction resources as needed.

However, the combination of abstract interaction, concrete input and concrete output entails several relations between the three models and the overall state of the interaction is only expressed by the combination of all three models. This also requires the interconnection of the models to facilitate information exchange between them. After discussing the relation of abstract to concrete and concrete input to concrete output in the following, the dependency of the UI to content information and the resulting mapping types are introduced.

Abstract-Concrete Relation

The abstract and concrete representations of the interaction are related to support the reification of the abstract interaction towards an actual user interface. While the abstract interaction model describes the semantics of the interaction (e.g. trigger a command, select an element), the concrete interaction model provides modality-specific interaction means (button/keyword, list). The two abstraction levels are connected via the state of the interactors. This means, if an abstract interactor is activated, the related concrete interactors also become active. An important point here is, that an abstract element combines input and output. Its state is mapped to multiple elements to simultaneously activate input and output means for the interaction (e.g. show the form field that typed characters are put in). The utilization of the data slot and related mappings ensures, that values to present are properly passed to the concrete output objects and received input is properly stored within the data slot and passed to the abstract level.

Besides the activation of concrete elements, abstract elements can also trigger the alteration of the presentation of an element. Focusing the graphical representation of a command (i.e. a button) via an input interactor, triggers a state change to *Focused* of the abstract interactor, which in turn changes the state of the related concrete interactor

to *Focused*, resulting e.g. in a highlighted button. This behavior is realized by mappings between the *state* attributes and the *setState* execution elements. Similarly, the selection of a button triggers the execution of the related *command* via a mapping targeting the execute method. In combination with multiple models, this creates a multi-level event propagation, that allows the incorporation of information from multiple models to interpret a received input event or to create the related output.

In contrast to the abstract-concrete relations that span two levels of abstraction, the concrete-concrete relations address the need to integrate input and output means not directly relevant for the interaction with the application.

Concrete-Concrete Relation

While interaction at the abstract level is application driven and focuses on the goal of the interaction, interaction on the concrete level is driven by the capabilities of the modalities. A long list on a screen e.g. requires scrolling, even if the anticipated abstract interaction is "only" the selection of a list element. Thus, there is another level of interaction that is not directly required for the completion of the task but important for comfort and clarity reasons. This interaction level is expressed as relation between the input and the output elements on the concrete level. Using the list involves the selection of a viewport and thus the number of items to display in parallel as well as the change of that viewport and thus the scrolling through that list. Such interaction is not directly related to the underlying user task, but crucial for the graphical modality. Additionally, scrolling through that list can also be realized via multiple modalities, making it a multimodal interaction directly related to a specific output modality. This example is illustrated in figure 5.9.

This example shows that input capabilities vary depending on the current form of the presentation and thus also depend on the currently active presentation objects. The direct relation of input elements to output elements is thus supported to facilitate controlling the modality specific representation. This way, the activation of a concrete output object can also activate additional concrete input elements, not related to the abstract interaction. Interaction with these input interactors is then directly mapped to the concrete output to influence its presentation. Figure 5.9 also shows that input elements can require specific output elements they rely on. In case of an active voice command, there may e.g. be an additional hint explaining the available voice commands.

The two different types of relation between interaction elements show that different levels of interaction means exist, that have to be addressed, when modeling multimodal user

interfaces. While this is often addressed by widgets, smart enough to announce and provide their required interaction means, full control of this behavior within the development process can be achieved by making this behavior explicit.

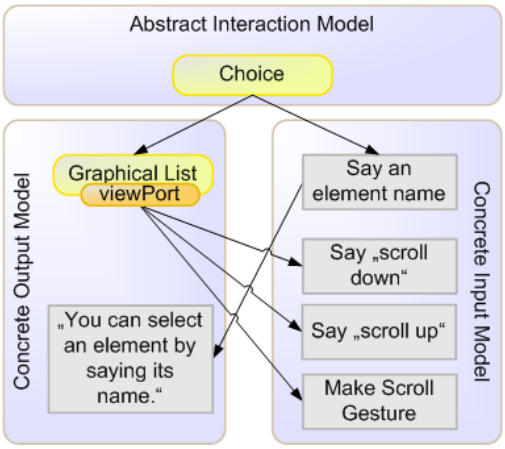

Figure 5.9.: Abstract example of a graphical list, that can be controlled via voice.

Content Dependent Interaction

Besides the interconnection of the different interaction models, with the utilization of interaction models at runtime the influence of dynamic (domain) data on user interface and interaction is another issue. This rises from the need to reflect dynamic information in the user interface. Three cases can be distinguished, that have to be addressed:

1. The most obvious case is probably a list, showing dynamic content (a list of available services, a list of values from a database). In this case, list elements have to be dynamically created. As this is a standard case in graphical user interfaces, it becomes more difficult with multimodal interaction. An example are list items displayed on a screen, that can also be directly addressed via voice. In this case there is also a need to create the required voice commands.

2. The dynamic creation of elements which are not part of a list. In this case, the type and number of needed interaction elements varies with dynamically acquired information (e.g. three vs five buttons with dynamic captures taken from a complex object, stored in the domain model).

3. Dynamic content can also play a role if multiple situations known at design-time should be considered (e.g. different numbers of buttons or an image instead of

text). This way multiple variants of the user interface can be created and the most appropriate variant is selected at runtime.

While the first case is addressed by the list interactor itself, which takes care of the creation of the items to display, the second case is handled by the provisioning of templates within the *DynamicInput* and *DynamicOutput* interactors. The actors handle the creation of interaction elements and take care of their presentation or input processing. In the third case the dynamic interactors allow the definition of different variants, that can be activated according to other model information. Variants and template utilization are handled via mappings to the execution elements and the activated elements are stored as situation (child-) elements. Storing these elements as situation elements is crucial for the approach, as they are created at runtime, according to the model processing logic and are not provided by the designer.

Based on these cases and requirements for the mappings between interactors, a set of mapping types expressing the described means can be identified as described in the next section.

5.5. Connecting the Models

The models described in the previous sections provide the core models of Ubiquitous User Interfaces. While each model defines specific aspects of user interfaces, each model on its own is not very comprehensive. The power of the approach lies in their combination into a net of models, comprising all the different aspects. Information can then be fed into the network to observe the behavior and derive the interaction results.

The basic mechanism underlying the approach is the propagation of state information (*inactive, active, selected, focused*) and user input/application data across models. User input is propagated from the user interface up to the higher levels of abstraction to realize a multi-level interpretation process abstracting the concrete input to a machine processable instruction. In the other direction, the propagation of application data to the user interface across multiple levels of abstraction facilitates the reification of the application information to human readable information.

Following the concepts of the mapping metamodel, described in 4.3, various mappings between the given metamodels have been defined. They interconnect situation- and executable elements across models at runtime and facilitate the exchange of information between the models. While the mappings between the different parts of the interaction

models have already been described in section 5.4, additional mappings between task-, domain-, service- and interaction-model are discussed in this section. Figure 5.10 shows the created net, connecting the core models to handle the interaction state.

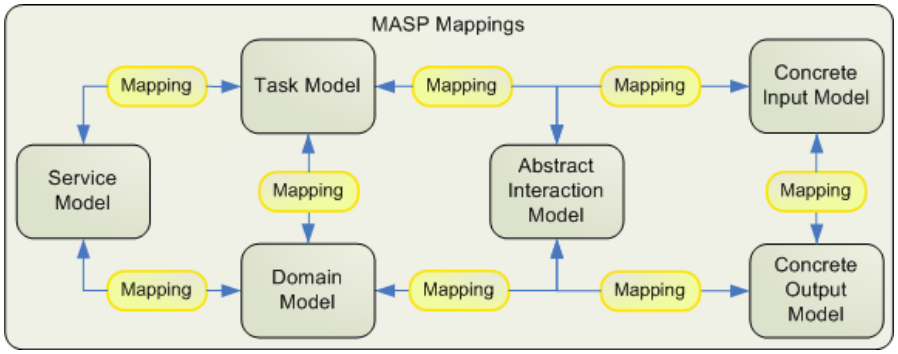

Figure 5.10.: Interconnection of and mappings between the involved Models.

Starting with the task model, the graphic illustrates that based on the workflow, defined in the task model, service calls or abstract interaction elements are (de-)activated. Mappings between the task- and domain model facilitate the specification of a life cycle and object management for the stored domain objects and allow the creation of objects if needed as well as the garbage collection of objects not needed any further. The service model utilizes information from the task and the domain model, to make service calls. Thus whenever an application task becomes active, the corresponding service call is activated as well. Mappings between service calls and domain objects take care about the passing of parameters to the service and the update of the domain object with the result of the service call. On the other hand the abstract interaction model is also related to task and domain model. In a similar way, as the activation of an application task activates a service call, the activation of an interaction task activates the corresponding interaction elements. The interaction elements are also related to domain objects, providing the dynamic information to visualize through the interaction objects as well as the information to alter through the actual user interaction. The abstract interaction model is related to more concrete representations of the interaction, separated into input and output.

Using this general structure a set of different types of mappings can be distinguished. Based on the mapping metamodel, all types relate definition elements of two different models and contain a set of links that detail the relationship. The mapping types are listed in table 5.7.

Mapping Type	Description
IsPerformedBy	Relates an application task to a service call or an interaction task to an interaction object. The mapping activates the corresponding service call or interaction element whenever the related task becomes active and also changes the task state to done, when the service call finishes or the interaction is complete.
Uses	Denotes the domain objects involved in an interaction or service call. In case of a service call the object can be a parameter or a service result, in case of a user interaction the object can be presented to the user or manipulated / created by the user.
IsStateSynchronized	Relates two interaction objects and synchronizes their state. This means that if one element becomes active, the other does as well. The relation is either direct or with a translation function redefining the dependency of the states. State synchronization from abstract to concrete or concrete to concrete elements e.g. guarantees that all related concrete elements are activated whenever an abstract interaction element becomes active and that related concrete elements are activated simultaneously.
IsValueSynchronized	Addresses the need to transport additional information between the different elements and across different levels of abstraction. It allows to communicate user input to higher levels of abstraction and to configure concrete interaction elements according to information from other elements. For the communication of input e.g. each concrete input element provides a value attribute (situation element) that can be changed during the interaction.

Mapping Type	Description
IsDynamicallyRelated	Addresses the creation of elements based on dynamic data. This is crucial to deal with input capabilities that depend on data unknown at design-time like e.g. a list populated with dynamically generated elements where each should be directly addressable via speech at runtime.The mapping is established to dynamic interactors (DynamicInput and DynamicOutput) which allow the identification of template elements and variants, that can be populated with the dynamic data. This allows the relation of dynamic data and the concrete visualization of input options related to this data. The transformation method can be utilized to incorporate the dynamic data into the input construct of a given type.
IsAccessedThrough	Relates an interaction object with an interaction resources that is used to present or manipulate the object.
IsAdaptedBy	Defines the adaptation of an interaction object according to context information.

Table 5.7.: The predefined mapping types supported by the mapping model to relate the defined elements.

While the *IsPerformedBy* and the *Uses* mapping mainly focus on the workflow and object management the *IsStateSynchronized*, *IsValueSynchronized* and *IsDynamicallyRelated* directly address the user interface management. *IsAccessedThrough* and *IsAdaptedBy* are related to context information. Linking the models together via these mappings allows the combination of the different models to a comprehensive executable system, exchanging information between the models, with the user and the service backend. The combination of the different models also allows the interpretation of the interaction on different levels of abstraction.

5.6. Discussion

The models presented in this section are strongly based on the current UIDLs presented in section 3.2.3. While UIML, UsiXML and TERESA XML provide similar elements, the presented models incorporate runtime aspects and the concepts of executable models.

They aim at the integration with a runtime architecture instead of the derivation of executable code.

While especially UsiXML and TERESA XML comprise a similar set of models there are some major differences. In case of the interaction model, UsiXMLs abstract user interface model distinguishes *components*, *containers* and *relations*. *Components* can have one or more facets of the type: *input*, *output*, *navigation*, *control*. *Containers* group *components* and *subcontainers* and define an *order type* and if the group is *splittable*. *Relationships* define *time* and *space*, *mutual exclusion*, *containment*, *adjacency*, and *dialog control* in terms of LOTOS operators. TERESA XML distinguishes output and interaction on the abstract level. Output interactors are: *textual*, *object*, *description*, and *feedback*; interaction elements are: *selection* (*single choice, multiple choice*), *editing* (*numerical edit, text edit, position edit, object edit*), *control* (*activator, navigator*), and *interactive description*. Additionally, interactor composition can be defined based on the following relations: *grouping, relation, ordering, hierarchy* and a connection to the functional core is defined as part of the model. In both approaches, the abstract model is refined by concrete elements at design-time. The concrete elements of UsiXML are the most fine grained here.

Main differences between the presented interaction models compared to UsiXML/TERESA are:

- The strong separation of abstract and concrete model to avoid redundancy, create clear boundaries and allow the incorporation of interaction means into adaptation and input interpretation at runtime. However, the creation of the models and the related mappings can become complex and requires comprehensive tool support. The different mapping types allow different views to the mappings in terms of mutual activation, information exchange and dynamic element handling.
- The separation of input and output and the definition of mappings between abstract-concrete and concrete-concrete levels provide an extended flexibility, and allow to establish two levels of interaction:
  1. task related interaction, that influences the application logic,
  2. presentation related interaction, that only controls the presentation.
- The integration of the CARE properties as attribute of complex interactors allows to express the relations between interactors across modalities.
- The incorporation of information from the task model at runtime helps to determine

grouping and temporal relations, which do not have to be redefined within the interaction models.

- Additional groups for output interactors can be created to allow a rendering specific grouping and the provisioning of group specific properties.
- The direct mapping between tasks and abstract interaction objects provides clear boundaries for the level of detail of the task model and allows to refine the task semantics as abstract interaction.
- Execution and interpretation semantics are part of the model, defined as execution elements. Embedding the execution semantic into the model allows to address the definition of the interactor behavior on the meta-level. The handling of the CARE properties and fusion e.g. can be supported by the execution logic of complex elements.
- The dynamic creation of interaction objects is supported based on templates that can be defined at design-time and a mechanism to handle these templates at runtime. Additionally, multiple variants of a user interface can be created.

The integration of the interaction model in a larger net of coexisting models at runtime instead of utilizing transient models at design-time also leads to a different utilization of mappings between the models. While a definition of different types of mappings between the models has also been provided by Limbourg et al. (2004a) in the context of UsiXML, the main difference to the their approach lies in the extension of the mappings by the inclusion of backend service access and the extended separation of tasks and manipulated objects at runtime. Using the presented mappings, the manipulation of domain objects is directly defined through the interaction objects. However, the UsiXML mappings provide valuable insights which can be transferred to runtime aspects and have been incorporated into the presented mapping types.

In summary, the presented models and especially the interaction model address several of the requirements identified for the UIDL:

- a possibility to define interaction, while not having to specify all details of the user interface to gain the needed flexibility (requirement #1.1)
- the relation and separation of input and output (requirement #2.1, #2.2) and the related support for distribution and multimodality,
- support of fusion and fission means by the provisioning of independent building blocks as well as the separation of input and output (requirement #2.4, #3.1),

- the dynamic utilization of data slots to store user input to support persistence (requirement #4.3),
- an explicit interaction state (requirement #4.1),
- different levels of abstraction (requirement #3.3) and their separation of modality and device independence and specifics.
- the possibility to integrate backend services via a service model (requirement #5.3)

Different types of mappings facilitate the synchronization of elements and their dynamic creation, and the overall approach is easily extensible if more interactors are needed. Utilizing the interaction model at runtime is strongly based on support by the runtime architecture and the interconnection of the interaction model with additional models to reflect the model state within the outside world (requirement #5.1). The next chapter illustrates the integration with a context model (requirement #1.2, #1.3, #2.5) as well as the means for input and output handling (requirement #2.3, #3.2), runtime distribution, shaping, multimodality, and adaptation (requirement #4.2).

5.7. Summary

In this chapter, a core set of metamodels for the creation of Ubiquitous User Interfaces has been presented. Based on a task metamodel, defining the workflow of an application, and a domain metamodel, defining domain specific data, service calls to the functional core can be triggered and user interaction can be initiated and processed. A major role is played by the interaction metamodel, combining abstract interaction with concrete input and output elements to provide a modality independent definition and interpretation of interaction as well as the independent support for input and output interaction resources. The abstract interaction model serves as central point to connect input and output and allows to refine interaction tasks in terms of semantics and interaction means. The concrete models support the development of multimodal interaction and allow the usage of the CARE properties to combine multiple modalities in different ways. Mappings between the different (meta-)models facilitate the synchronization of state information and the exchange of information between the models. All presented metamodels are based on the concepts of the meta-metamodel of executable models an are thus comprised of definition-, situation- and executable elements.

A major advantage of the integration of execution concepts within the metamodels is the possibility to utilize these models at runtime. The proposed linking of the models allows

the execution of the task model to trigger a chain reaction, leading to the derivation of a set of active concrete user interface elements from the defined user interface models. Stimulation from the outside via the interaction or the service model trigger the derivation of the next interaction state, resulting in new interaction means and executed services. Thus the net of models provides a dynamic view to the runtime behavior of the modeled system. It forms the internal representation of the system, that has to be conveyed to the user via the interaction resources.

However, to do so additional elements are needed that cover aspects like distribution, fusion, and adaptation. This requires bridging the internal state of the models to the outside world. Context information is added, channels convey the model state to interaction resources, and distribution, fusion and adaptation means are added to the network of models. This additional infrastructure is provided by the Multi-Access Service Platform as explained in the next chapter.

6. The Multi-Access Service Platform

After introducing the notion of executable models in chapter 4 and the set of reference metamodels in the previous chapter, this chapter deals with the integration of the concepts with an architecture for the creation and handling of Ubiquitous User Interfaces for smart environments. The Multi-Access Service Platform (MASP) is introduced as an implementation of the described concepts, that integrates additional components and models to bridge model and outside world and handle shaping, distribution, multimodal interaction and adaptation as basic UUI features.

The MASP combines executable models with these additional components and can be denoted as runtime infrastructure for executable models. However, as executable models provide their own interpretation logic, this does not fully reflect the nature of the approach. In this sense the MASP rather is a set of metamodels, combined with a set of components, to enable the creation of multimodal user interfaces for smart environments. Figure 6.1 shows the introduced meta-metamodel of executable models (M3), the described metamodels (M2) as well as the user interface models (M1) of the MASP in relation to the MOF Meta Pyramid.

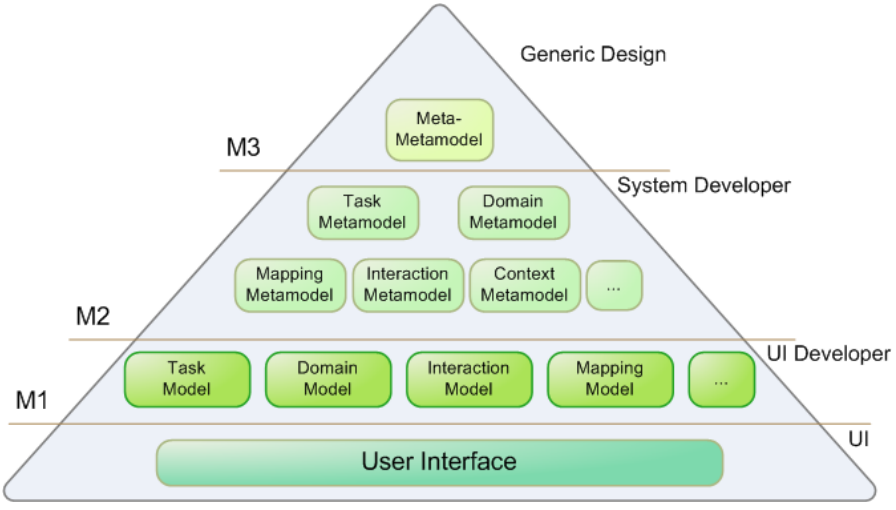

Figure 6.1.: The MASP models in relation to the MOF Meta-Pyramid.

At runtime the M1 level is the most important, as it comprises all developed UI models utilized for user interface creation (and input interpretation) by the architecture. It configures and drives the connection of the functional core as well as the user interaction and the mediation between the application and the user using the application in terms of input interpretation and output generation. From the current state of the models, the final user interface representing the interaction state, is continuously derived and distributed to the available interaction resources.

In this chapter, the overall architecture of the MASP and its different building blocks are illustrated. Starting with an overview of the architecture in the next section, the additional models and components are described. This includes the context model, making live context data accessible for the executable models, as well as the concept of interaction channels, making interaction resources directly accessible for the MASP. Additionally, the underlying concepts for UI distribution, shaping, multimodal fusion and adaptation are introduced in this chapter. Finally, a walk-through illustrates the interconnection of the different components and models and illustrates how user interfaces are created and interaction is handled within the MASP.

6.1. Architecture

The overall architecture of the MASP combines the described core models with additional models and components, to reflect the main concepts identified for the creation of UUIs in section 3.5.2. As shown in figure 6.2 it directly integrates the core models to handle the state of the interaction and integrate backend logic. They allow the derivation of multiple user interface variants for varying modalities and IR combinations. The mapping model contains all mappings defined between the various mappings. However, additional models have been added, to provide additional features and address specific problems of UUIs. The MASP Core Model configures the utilized models and component for any application and provides general access to the used models via an exposed API. It provides the means to initially load applications (sets of models) and trigger their execution. The model contains sessions for the user and application management and instantiates the external components via proxy elements. Additionally, it provides a basic API to access the models, making it easy to build software and management tools for the platform. The MASP Core Model is created by the developer and represents the configuration of the developed application. This allows the developer to directly influence which models are used and how these models are linked. It also allows the runtime system to manage

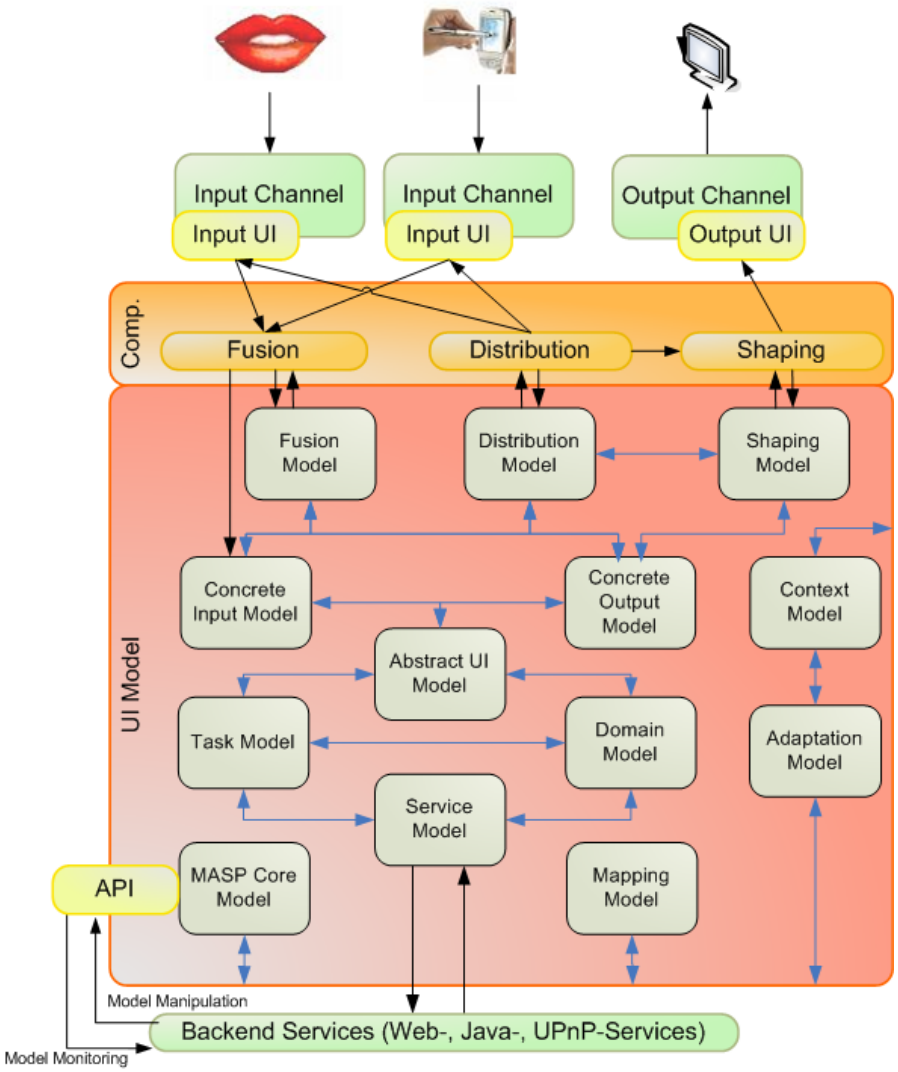

Figure 6.2.: MASP Architecture with exemplary IRs.

users accessing the application and to start and stop instances of the application accordingly. A context model provides direct access to context information for the created models. Shaping, distribution and fusion are realized by a combination of problem specific models, that provide configuration possibilities and hold calculation results, with additional components. An adaptation model addresses the adaptation of the models to the context of use. Finally, input and output channels (handled through the context model) connect interaction resources and make them available to the MASP. They handle the UI presentation and provide the means to push updates to the presentation at any point in time and to receive user input. Input is interpreted based on the current state

of the interaction and fused with input received via additional modalities.

This combination of executable models with additional components as well as external elements like context providers and channels, connected via proxies, allows the creation and handling of UUIs. The infrastructure aims at the realization as a central server within a smart environment, allowing the aggregation of multiple interaction resources and the handling of multiple applications and users. Each available interaction device registers its resources, which are then directly addressed independently by the server-side system. Additionally, the server allows the integration of central context sensing capabilities for the whole interaction environment with multiple applications and users.

The main components and the full underlying interaction mechanism are illustrated in detail in the following. Starting with the context model in the next section, distribution, shaping, channels and UI delivery are explained afterwards. Fusion and adaptation means round up the chapter which is then concluded with a complete walk-through and a summary.

6.2. Context Model

The context model provides the means to include context information as basis for adaptation and interaction configuration. It aggregates context data from various sensors and provides a central model to access the acquired data. It bridges the user interface models and the outside world, by integrating sensor information into the model level. From the design perspective, its main task is the identification of the context information relevant for the application. From the runtime perspective, its main task is the acquisition and provisioning of live context data at runtime.

At design-time, there is a need to specify the relevant context information, to ensure that the modeled concept can properly reflect these. In this sense, a design-time context model needs to specify relevant data and provide the means to define hypotheses about this data at runtime.

At runtime, the model is filled with information delivered by various sensors and triggers behavior or UI adaptations dependent on the context. At this stage, the context model reflects the actual real life context, the application is executed in. This results in the need to acquire and maintain data about the current context and provide this to the application (in a way that it understands). Modeling the context as an executable model allows to address this by the definition of dynamic elements as well as the integration of

context providers directly into the model. This is supported by the utilization of proxy elements, encapsulating the actually used providers. Considering smart environments, this also leads to the need for configurations, that reflect the static properties of the environment. In this sense, the context model contains a specific configuration for the very environment, that identifies static values (e.g. rooms) and is configured with the required parameters to incorporate the available dynamic context providers.

As depicted in figure 6.3 the used context metamodel comprises information about the user, the available interaction resources and the environment, including position information of user and devices in terms of rooms and coordinates. Additionally, details about the interaction resources are emphasized as they are most relevant for the interaction. Context providers are directly configured and integrated within the model. The different parts are discussed in the following.

6.2.1. Environment Information

The context model specifies an environment with a set of rooms and devices as basic elements. Devices consist of interaction resources, which are located in rooms and can have detailed positions. The capabilities of these resources are further specified in the IR part of the model. Users within the environment can then also be located within a room. Similar to IRs, they have a position, which can be set. For the utilization of a radio- or visual-tag based localization, tags can be assigned to any of the elements to determine their positions. Finally, rooms can have areas that identify specific parts of the room, which might be especially interesting for interaction purposes (e.g. in front of a screen).

6.2.2. User Information

User information within the model aims at the modeling of user preferences and capabilities. In the current model it is limited to the (more or less abstract) preferences and capabilities. Additionally, the preferred modalities and some configuration information (the status of the follow-me function) can be specified. The current position of the user in terms of the room and the respective coordinates is important information for a whole list of application functionality. While information can easily be added as needed, the currently available information grew from the specific needs of the developed demo applications.

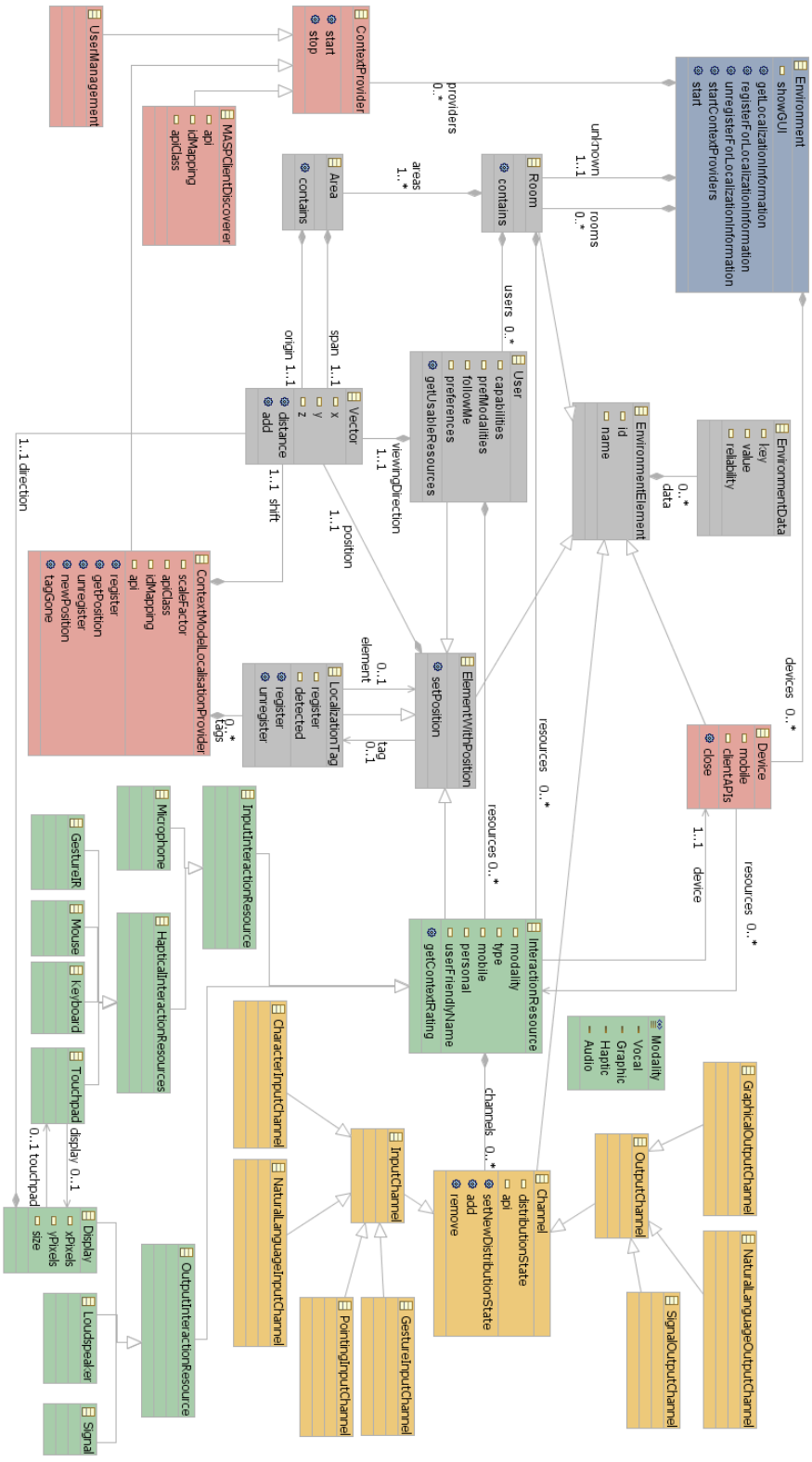

Figure 6.3.: Context Model

6.2.3. Integrating External Processes

The integration of external information is a central aspect within the context model. While the other models discussed so far are mainly self-contained, the context model has a strong connection to the outside world. It has to provide means to acquire data from as well as to interact with the outside world. As introduced in section 4.2.3, proxy elements are utilized for this purpose. Proxy elements encapsulate external processes to integrate them into the model and provide control, without having to completely specify them as part of the model. While configuring a context model for a given environment, the connections to the context providers are configured (e.g. a webservice URL) and the provided information is mapped to information of the context model by identifying the executable elements acting as callback for the process. With this method external processes can be started and provide dynamic information incorporated into the model via well-defined interfaces. This is illustrated e.g. by the localization provider, setting the locations of the known tags at runtime. Other supported context providers include a user management, providing information about the known users, and a client discoverer, providing information about available interaction resources. A similar mechanism is also used to connect the available interaction resources to the system via the channel. Here the channel provides a proxy allowing to send data (e.g. HTML code) to the external resource and to receive incoming interactions.

6.2.4. Interaction Resources

The available interaction resources are a major aspect of the interaction context within the smart environment. The presented model distinguishes input and output interaction resources with different capabilities to support the interaction model defined in section 5.4. The maintained interaction resources are mainly categorized by their type and the modality they support. Additional knowledge about mobility and personal or public character of the resource can also be considered if information is to be pushed to a user. The concept of a channel is introduced, to identify the connection to an interaction resource that can be used for information exchange and interaction with the user. A channel provides an internal representation of an interaction resource. To match the concrete interaction model concepts to a platform specific language, the channel acts as proxy object for the implementation of a resource and modality specific communication channel which is encoded as external process.

6.3. Interaction Channels

With the goal, to flexibly utilize various interaction resources, there is the need to make them directly accessible. While the IRs are managed by the context model, the actual communication with the resources is realized via interaction channels. Using proxy elements, these channels are assigned to the interaction resources by the context model and connect the physical medium to the system internals. Internally, they provide logical representations of the resources and their capabilities, abstracting from the underlying platform specifics and the platform dependent communication format (e.g. HTTP), while providing a common interface for resources of various types.

Each interaction channel connects a single interaction resource to the MASP and thus supports a single modality. This also entails a separation of input channels, conveying input user interfaces and output channels, presenting output interfaces. Input channels monitor user input, which is preprocessed based on an input UI, derived from the state of the concrete input model. Output channels translate the state of the concrete output model into a perceivable user interface presentation. The main role of the channel is to ensure, that provided information can be processed on both sides (IR and system). It thus mediates between the internal representation of the system and the presentation format of the resource. Main elements, as shown in figure 6.4 are:

- a target language processable by the resource (e.g. HTML),
- transformations between the resource specific language and the internal representation of the system,
- an API to send information to and receive information from an interaction resource,
- a meta-description of the resource capabilities.

Figure 6.4 illustrates the utilization of input and output channels. Receiving the concrete interaction elements to handle (after the distribution has been calculated, as explained in section 6.4), the channel transforms the elements into the final user interface of the specific target language. In case of input, user interaction is translated back into events handled by the fusion component (see section 6.6) and then by the concrete input model.

In the following, a set of exemplary interaction channels that have been realized to build multimodal applications with the MASP is introduced and the integration of channels and models is illustrated.

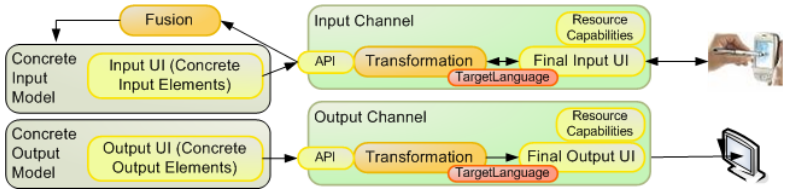

Figure 6.4.: Channels connecting a pen input IR and a graphical output IR to the MASP.

6.3.1. Channel Types

The current MASP implementation distinguishes seven types of channels, matching the types of the input and output interaction objects defined in the concrete interaction model (see section 5.4).

The **Natural Language Input Channel (NIC)** allows the processing of natural language input of the user. This includes text processing as well as Automated Speech Recognition (ASR) engines. It provides the means to encapsulate speech recognition facilities and abstracts from the underlying specifics of different engines. The encapsulated language is thus a data exchange format like e.g. VoiceXML. NICs have been used to connect Dragon Natural Speaking, a Voice Genie Platform and a Loquendo System. An additional Text Input Channel (TIC) has been created to substitute voice interaction and perform Wizard of Oz tests. It realizes keyboard input the user could enter to a chat-like window. Additionally, the TIC allows the integration of handwriting devices if the recognition software is able to produce a text string from the provided input.

The **Character Input Channel (CIC)** supports the connection of various devices providing simple discrete events. The channel can be used to directly feed single key presses into the system (e.g. the up arrow or the enter key). It also allows to connect other interaction resources providing input that can be mapped to single discrete event like e.g. the click of a mouse button, without the consideration of the mouse position as well as the usage of a remote control or of any kind of switch. User interfaces for this type of interaction device provide a mapping of single characters to discrete events and thus also limit the set of processable character.

The **Pointing Input Channel (PIC)** allows the incorporation of pointing information into the user interaction. Pointing in this sense includes the mouse position as well as any pointing gestures recognized by the system. The PIC is able to continuously monitor pointing information from the connected resource and to provide pointing information

on request. It can also send events if a pointing inside or outside of a given area has been perceived. The relevant areas would in this case be defined by the provided input user interface. This allows on the one hand the monitoring of pointing gestures and the query of the monitored gestures to resolve deictic references e.g. in the voice input. On the other hand pointings of the user to given coordinates can directly trigger processable input events.

The **Gesture Input Channel (GIC)** allows the detection of gestures performed by the user. The basic channel concept can be used to connect different modalities providing gesture-based input to the system. This includes mouse gestures, 3D gestures using accelerometers or camera-based systems as well as pen gestures. Gestures also include simple mouse clicks or touches on a touch screen, interpreted as selection gesture (point and click). User interfaces supporting gesture input provide mappings of a gesture to MASP input events that can be directly processed by the defined concrete interaction model. This approach also helps limiting the gestures that can be processed at a certain state of the interaction and can thus help to improve detection performance.

The **Natural Language Output Channel (NOC)** allows the connection of Text-To-Speech (TTS) engines as well as of text capable displays. The channel internally provides the means to present any natural language text to the user and thus accepts such texts for presentation. It has e.g. been used to connect the Loquendo and Voice Genie TTS, which allows to trigger voice output at any point in time. Additionally, the possibility to output the same information as text in the Wizard of Oz tool is provided, supporting a chat like interaction style. Chat like interaction can also be utilized in noisy environments or to keep information private and can be combined with textual or pen-based input.

The **Graphical Output Channel (GOC)** allows the connection of graphical displays as interaction resources to the MASP. It connects any graphical rendering device, which includes HTML-based web browsers as well as GUI-based applications like Java Swing. The channel provides the means to graphically present the concrete presentation objects to the user and serves as reference system for pointing based interaction, which is required as the coordinates supported by the pointing interaction strongly depend on the related graphical presentation. To address this issue, a global coordinate system is maintained, to which local graphical and pointing coordinates can be mapped to calculate pointing areas.

Last but not least, the **Signal Output Channel (SOC)** provides the means to convey very simple interaction, e.g. to gain the users attention. The SOC allows the connection of very simple interaction devices that produce sounds, light effects or haptic feedback like

vibrations. It thus allows the transportation of one dimensional, simple interaction that conveys (maybe) important, but not very detailed information. Signals mainly follow a fire and forget approach. That means a signal is simply been fired and not maintained in any way by the channel.

6.3.2. Integration of Channels and Models

To utilize the different types of channels, each channel provides a channel API, comprising similar methods for each channel. As each channel handles a set of concrete interaction objects, that define the current interaction (input or output) possible on this channel, each channel also provides the possibilities to manipulate this set of interaction objects. This includes adding and removing objects, as well as updating complete objects or only a certain field of an object. Additionally, the channel provides the means to alter the state of the interaction object in terms of focused, unfocused, selected, etc.

These functionalities are mapped to executable elements of the proxy element of each channel in the context model. These mappings are dynamically created at runtime and are of the *IsAccessedThrough* type. Thus, the activation of concrete output triggers a mapping, that add the element to the channel to trigger it rendering on the connected IR. In detail, this is done after the distribution has been calculated and the shape of the UI for each of the utilized IRs has been determined as described in section 6.4 and 6.5. A constant connection is kept between the state of the concrete element in the model and the channel, which ensures, that any state change is immediately conveyed through the channel to the connected IR. Additionally, each input channel allows the registration of a callback handler, processing the received user inputs, which are handled by the fusion engine described in section 6.6 and finally processed by the net of UI models to alter their state accordingly.

6.3.3. Summary

The described channels provide a multi-layered approach to handle user input and system output, with some similarities to the Pipe-Lines of Nigay and Coutaz (1997). They mediate between the interaction resource and the model, abstract from the very details of each interaction resource and allow the easy incorporation of new interaction capabilities into the MASP. The continuous access to the IR allows to immediately convey any changes of the state of the model to the connected resources. This means, if e.g. a new

voice command is added, the channel takes care of updating the grammar and retransmitting that grammar to the ASR. This way it is guaranteed, that the maintained user interfaces always represent the current state of the interaction, convey the most recent information and restrict the interaction to the currently processable input. Underlying the utilization of different interaction resources at runtime is the possibility to calculate a distribution of the UI among them. The underlying mechanism is strongly based on the interaction model and described in the next section.

6.4. User Interface Distribution

The goal of the distribution mechanism of the MASP is the assignment of concrete interaction elements to interaction resources for presentation purposes and input handling. The main challenge in this context is the provisioning of an optimal distribution with respect to context, interaction elements (content), available IRs, and user preferences. Figure 6.5 illustrates this approach.

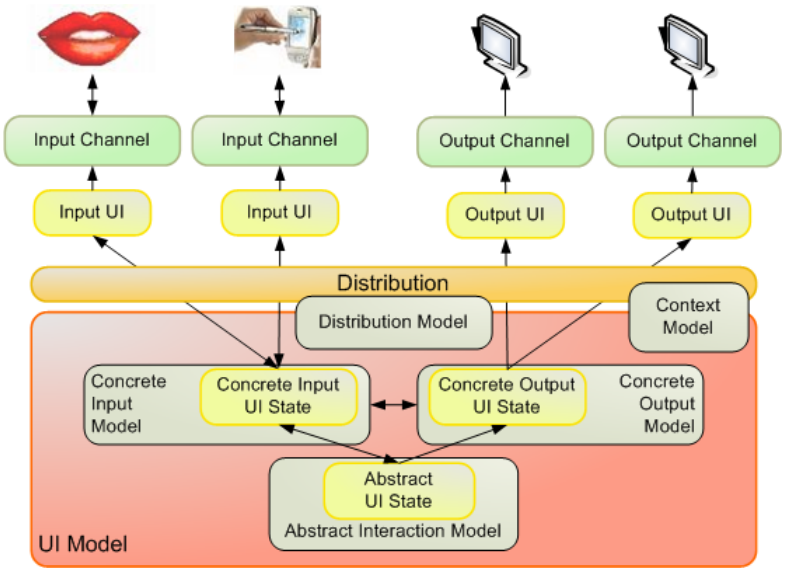

Figure 6.5.: Concept and goal of the distribution.

The UI model contains the abstract interaction model and concrete input and output models as well as the other core models. The interaction state identifies the UI elements that currently have to be presented to the user as well as the currently available input possibilities. Based on these elements and the available interaction resources a

distribution configuration is calculated, which assigns interaction elements to interaction resources. These input and output user interfaces are then delivered to the interaction resources via input and output channels.

Based on this general mechanism, the distribution component and the distribution model, which store the current and past distributions, are described in the following. Utilizing the described interaction model, the goal of the distribution component is the assignment of concrete interaction objects to interaction resources for presentation purposes and input handling. The distribution model stores the current state of the distribution and provides design information specified by the developer.

6.4.1. Distribution Component

Based on the set of active interactors, the distribution component has the goal to calculate which interactor shall be presented on which of the available resources. The discovery and management of interaction resources is thereby accomplished by the context model and provided to the distribution component. A distribution configuration is influenced, by different additional aspects:

- information describing the current situation (the active interactors and context information like the available IRs),
- previous distributions calculated by the system or configured by the user for the same set of interactors and the same context (incl. history and user configurations)
- requirements of the current interaction step, that are the constraints imposed by the developer (CARE properties in the interaction model and distribution configured by the developer).

Due to the anticipated independent handling of input and output the distribution has to consider the two types of user interfaces, leading to two distribution goals.

- Input user interfaces, restrict the interaction possibilities and support the integration of multiple input capabilities. Restricting the possible user interaction in a single modality can e.g. be done by influencing the recognition engines (see also the approach of Coen (2001)). In terms of input the distribution goal can be formulated as: support as many (equivalent) input resources as possible while considering the specified CARE relations between the input elements. This aims at leaving the control about which interaction resource to use to the user by supporting a wide selection. However, based on information from the context model, only currently available input resources within the vicinity of the user are considered.

- Output user interfaces, provide the presentation that conveys information from the system to the user. In terms of output the goal can be formulated as: find the most suitable combination of output resources while considering the specified CARE relations between the output elements. Distributed output thus aims at utilizing the most suitable combination of interaction resources to convey the user interface. The selection of resources again depends on their capabilities and context information like the resource location.

Based on these goals, the assignment of the different parts of the UI to the interaction resources is performed by the distribution component. Whenever a new set of concrete interactors is activated, the distribution component is notified and triggers the execution of the distribution algorithm. The algorithm first checks for suitable previous distributions within the history and the user specified distributions. As these distributions were already active in the past, they are considered as suitable. The algorithm first selects every distribution specified for the same set of concrete interactors. Furthermore, the situation describing the circumstances under which the distribution was established is compared to the current situation. Therefore, the relevant context information is retrieved from the context model and compared to the saved context information from the selected distributions. All distributions where the context comprehends less or equal amount of information than the current context are now marked as relevant for further consideration. These distributions were active when having at least the same amount of knowledge as currently available and are thus relevant for further consideration. One, more than one or no distribution can be found during the comparison process. If only one distribution is found, the distribution algorithm finishes and the distribution is set as current distribution. This results in the creation of a mapping between the concrete interactor and the interaction resources. If multiple distributions are found, the algorithm evaluates the creator of the distribution to determine the most suitable one. Distributions from the user are considered better than distributions of the application as user configurations are done by users to adapt the existing distribution to their current needs, which neither the system nor the application designer can calculate or foresee better. Similarly, application configurations are considered better than distributions calculated by the system as the application developer specifies distributions with a specific intention, which the system should not overrule. If the comparison of the saved distributions does not reveal any results, the current context comprehends more information than the saved contexts. Thus, the distribution might be done with better certainty. The system then calculates a distribution based on information from previous distributions, the current context and properties of the user and constraints defined by the application developer.

The saved distributions with the least proximity in terms of information context to the current context are prioritized and evaluated in relation to the additional information. The result is the storage of the calculated distribution in the distribution model, which is described next.

6.4.2. Distribution Model

The distribution model depicted in 6.6 is used to store distribution constraints as design-time configuration specified by the developer (left part) as well as the runtime results, user settings and history of the distribution calculations (right part).

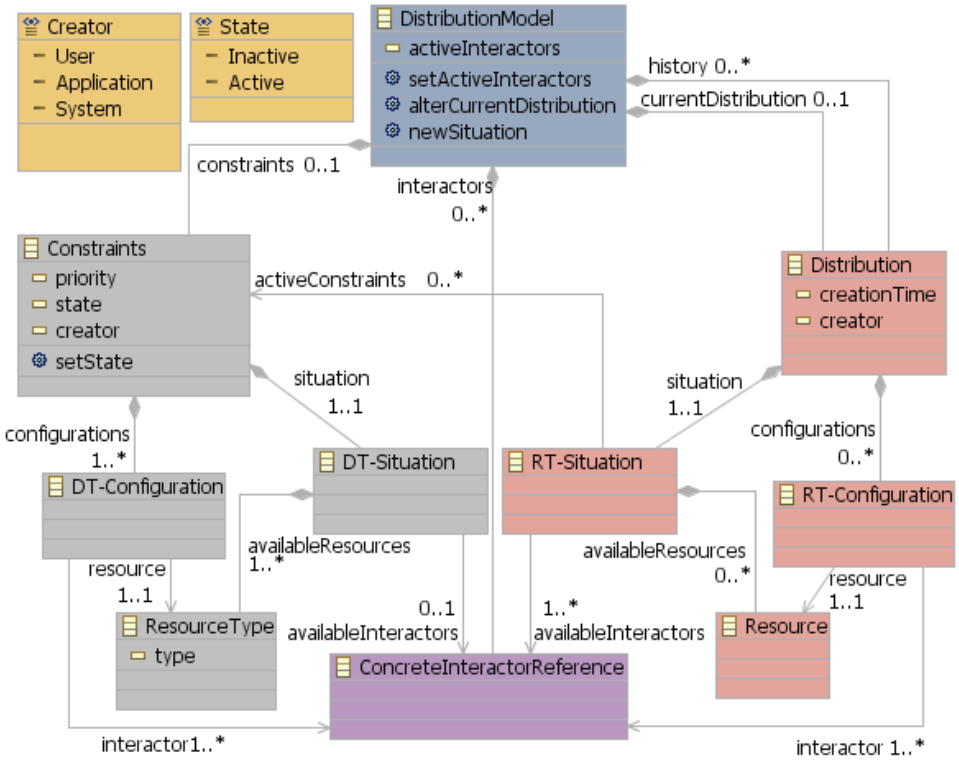

Figure 6.6.: The distribution model, storing/triggering the distribution of the added elements.

The design-time configuration of distribution constraints allows the developer to specify application specific distribution hints. Constraints therefore contain configurations that map concrete interactors to interaction resource types. This allows e.g. the assignment

of a PDA display as preferred IR for a set of interactors and the definition of a fixed touch display for others. Situations acting as precondition for the constraint can be defined based on information modeled in the context model. Constraints are dynamically (de-)activated at runtime and only active constraints are considered throughout the distribution calculation. At runtime, the distribution model mainly acts as storage for calculated and user distributions. The current distribution is expressed as a set of configurations that map concrete interactors to specific resources, based on the current context situation. This situation also contains the currently active constraints as these can change and lead to a new distribution. Each applied distribution is also defined by a timestamp of its applications and the creator of the distribution (user, application, and system can also directly assign interactors to resources). A history stores all applied distributions for later consideration within the algorithm. Storing the current distribution within the model also makes this information available to the interaction model. A mapping transports the information of the currently used interaction resources to the variables in the interaction model. At runtime, the interconnection of the interaction model and the distribution model allows the flexible handling of the interaction elements. The close interconnection of the distribution model with the distribution component bridges the gap between the modeled information and the results of the distribution algorithm that considers various information from multiple models, including context-, interaction- and distribution model.

A major reason for storing the current distribution configurations within the distribution model is to make it accessible to other executable models using the mapping mechanism. This is required to address the issue of referencing resources and modalities from within the user interface. The definition of variables in the interaction model, allows the definition of mappings, that map the current distribution configuration for a certain element to the variable. In detail, the user friendly name of the interaction resource and the used modality would be mapped to this variable. This allows output interfaces to refer to distribution information as well as to interpret user utterances referencing resources and modalities within the fusion process.

6.4.3. Distribution Sequence

To distribute a set of concrete interactors to the available interaction resources via the channels, the distribution component implements the mechanism, exemplary illustrated in figure 6.7.

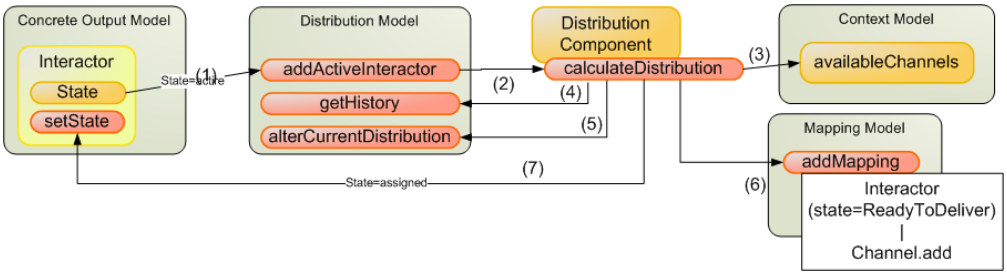

Figure 6.7.: Illustration of the sequence of the basic steps of the distribution calculation and the relation of the involved models.

Whenever a new set of concrete interactors becomes active, the distribution component is notified, that a new distribution might be needed. This is realized via a mapping between the concrete interactors state and the *addActiveInteractor* method of the distribution model (1). Triggered by the method call, the distribution component calculates a distribution (2). If a configuration for the context is available, it is applied. Otherwise a new configuration is calculated according to the algorithm described above, by querying the context model (3) and incorporating the capabilities of the resources, their location and information about the user like location and situation as well as earlier distributions (4), etc. In any case, the result is the assignment of the active interaction elements to specific interaction channels, stored in the distribution model (5). Based on the calculated distribution configuration, the distribution component creates a mapping between the concrete interactor and the executable element *add* of the selected channel (see the context model), which ensures that the presentation information of the interactor are conveyed by the channel (6). This mapping is triggered when the state of the interactor is set to *ReadyToDeliver*, which is done by the shaping component setting arrangement and style for all components on a given channel (explained in section 6.5). Finally, the state of the user interface is set to *assigned* to trigger the shaping as next step of the UI creation process (7). Once the UIs have been delivered to the IRs, user input is processed from the input channels. The goal of this input processing is an alteration of the state of the runtime model. Any state change of the model then results in an update of the presentation. The support of different interaction resources and the loose coupling of multiple interactors also requires means to control the arrangement and style of the user interface elements. This is discussed in the next section.

6.5. User Interface Shaping

Creating user interfaces, that can be dynamically arranged across multiple interaction resources poses high challenges in terms of user interface layouting to the user interface developer. If the combination of interactors on any interaction resource can not be completely defined at design-time the temporal and spatial arrangement of these interactors becomes very complex. The shaping component uses a constraint-based layouting for this purpose, which is briefly introduced in the following and described in detail in (Feuerstack et al., 2008).

The shaping of the user interface takes place after the different interactors have been assigned to the available channels. It arranges the user interface elements of each channel based on constraints, that specify their order, orientation, containment, and size. Thus once an interactor reaches the state *assigned*, the shaping model is notified and the interactor is scheduled for layouting. The underlying process is similar to the described distribution process. The shaping component determines the global coordinates for the interactor based on the defined constraints, the other elements assigned to the channel and additional information from other models like task model (for grouping and temporal relations), abstract interaction (for grouping) or the context model (for user preferences and interaction resource capabilities). These coordinates are then added to the interactor by the *setArrangement* method. Additional style information can be provided by the *setStyle* method. Once the layout of the component has been completely determined, its state is set to *ReadyToDeliver*, which triggers its delivery to the interaction resource via the channel. Important steps during this delivery are the translation of the set of concrete interactors into a device and toolkit specific representation and the translation of their global coordinates into absolute coordinates for the specific channel. The same mechanism can also be applied for the temporal arrangement of voice user interfaces.

Although this approach is very powerful, it might not always produce the desired results. While a main goal of the approach is to keep the interface usable even if modalities and devices change, there is also the need to support the provisioning of optimal presentations for predefined usage situations. While the general arrangement and layouting of the user interface is determined by layouting information on the concrete interaction level, providing e.g. coordinates and sizes of elements, the final style is applied by the channel, which is the instance aware of the exact capabilities of the interaction resource. To support this device specific adaptation the interaction channels include additional means to define specific styles that can be applied to the final presentation of a user interface. These stylesheets allow the very specific determination of user interface properties using veloc-

ity templates, XSLT transformations and Cascading Style Sheets (CSS). The different types of stylesheets are provided for different use cases. On the one hand each channel provides rendering templates for each concrete interaction object. These templates are available as velocity templates, which provide powerful templating mechanisms and can be changed by the developer if needed. Additionally, for any XML-based (intermediate) result of the rendering process in the channel, XSLT transformations can be applied to incorporate additional information. For HTML-based presentations there is also the possibility to directly incorporate Cascading Style Sheet (CSS) information into the web page.

Finally, the most powerful possibility to influence the presentation of a user interface is the extension of the MASP with customized interaction channels. Such channels have the complete freedom of applying any rendering techniques and technology the developer needs to transform the concrete user interface elements into the final presentation or input configuration. Using the channel API, new channels can be easily integrated into the MASP.

The utilization of constraint-based shaping algorithms and styling capabilities allows to optimize the presentation results of the created user interfaces. Based on the channel capability to continuously synchronize the state of the interactor in the model with its external representation on the interaction resource, any alterations to arrangement and style are immediately reflected within the user interface. This also allows the dynamic adaptation of the layout, style and shape of the user interface, according to any properties of the user interface state. One approach to convey dynamic alterations, currently applied within the MASP is the smooth translation of the alterations into animations via the Scriptaculous Animation Framework, which help the user to keep track of the modifications. This provides a stronger impression of a consistent interaction experience than e.g. having user interface elements jump around when the configuration changes.

6.6. Multimodal Input Processing

Besides the described handling of output, a major aspect of Ubiquitous User Interfaces is the handling, integration and interpretation of multimodal user input. While the concrete input model provides means for the developer to define expected user input, at runtime the need to match the actually received input with the expected input arises. This is supported by the concept of input user interfaces, that allow the preprocessing of any user input. Based on these input interfaces, input events are generated and transmitted to the

fusion component of the MASP architecture. The component analyzes the received input according to the concrete input model and aims at the resolution of references and the realization of CARE relations between received input. Processed (or combined) events then trigger executable elements of the concrete input model, which in turns triggers the event processing of the core models. Details about this model-based input processing and fusion process are given in this section.

6.6.1. Monomodal Input Processing

A major aspect of the described approach is the constraining of the interaction possibilities. Instead of aiming at the processing of any user input, the goal is to filter out unexpected input as early as possible. Restraining the interaction is thereby similar to what humans do during conversations, try e.g. going to burger king and order five stamps. They probably reply "Excuse me?", because you are outside of their domain of discourse and even if they understand you, they will most likely not be able to fulfill your request. This behavior is mapped to the system by input user interfaces.

Input user interfaces are derived from the concrete input model by distributing concrete input elements to interaction channels. Once a set of elements is delivered to a channel, it is the channel's responsibility to filter any user input it receives according to these elements. This can be done in various ways. In case of pointing input, relevant areas can be specified, for voice input, a grammar for Automatic Speech Recognition (ASR) can be defined, for character input, the known characters can be provided and for gesture input, the set of gestures can be determined. This allows the matching of the received input with the expected input and the removal of misleading or unexpected input. The goal of this process is the creation of input events that are delivered to the fusion component as has been illustrated in figure 6.4 in section 6.3.

Input events are generated by the input channel, based on the received user input and the underlying information from the concrete input model. They incorporate information about the user, the used IR, channel and modality, the confidence level, a timestamp as well as the actually received input. The concrete input interactor, this input is assigned to, is referenced if possible. Additionally, an input interface can also produce n-best lists containing multiple possible inputs and interpretations (e.g. pointing in between two objects or an ASR result) and a confidence level. Unknown events are forwarded if needed, as they might still be relevant for the fusion process, e.g. to improve the robustness of the input handling. Once the input event has been created, it is forwarded to the fusion component for further processing.

6.6.2. Fusion Component

Based on the different events that can be produced through the input user interface, the fusion component matches the received input events to what is expected according to the active input elements within the current state of the models and combines multimodal input according to the complex interactors of the concrete input model. The goal of the fusion is the mapping of the received input event to an executable element of the concrete input model to trigger the event processing of the core models.

To realize this, the fusion initially checks if the received event can be directly mapped to a concrete input element, in which case the executable element could be directly triggered. If a matching is possible, but the event belongs to a complex input element, expecting redundant or complementary input, additional input events have to be awaited, to trigger the related execution. If the event can not be directly matched, information from additional models is queried, to resolve the input event. The current distribution is checked, to evaluate the concrete elements assigned to the origin channel, while context and previous (partial) fusion results are also considered.

In addition the fusion can also deal with the handling of ambiguous input and reference resolving (deictic and temporal), querying missing information, resolution of unclear input (e.g. n-best results), and the handling of recognition errors to make the interaction more robust. While unclear input and recognition errors can also be filtered out by the mono-modal pre-processing of the channel, providing such input to the fusion engine can be helpful to support a lazy input interpretation. Querying missing input can happen via mappings of the fusion model to the concrete interaction, triggering state changes in the core models.

Three types of results of the fusion process can be distinguished:

- **Finalized self-contained input,** that can be directly mapped to an executable element of a concrete input element and is directly forwarded by calling this execution element.
- **Open input,** that requires additional parameters to be processed, is related to a complex input interaction object. Based on this object, the fusion engine determines the types of input that are missing to complete the complex object. Thus it stores the received open object in the fusion model and waits for additional input completing the element. After a given timeout or the reception of another distinct open event the processing is interrupted and the object is removed from the fusion model. Events to complete an open interaction can either be another open

interaction in which case the complex input object would contain another complex object, a self-contained event, that can be mapped to a sub-element of the complex element or an ambiguous event, that matches the open slot of the complex object.

- **Ambiguous input** is produced whenever an input user interface allows to find two internal representations for a single input event. In this case both interpretations are provided to the fusion engine, which has to resolve the ambiguity. This can be done based on context information, recent interactions from the interaction history or additional events (e.g. waiting open input).

Storing the (partial) results in the fusion model allows the user interface model to react to missing information. Mappings from the fusion model to other models allow the direct influencing of the user interfaces by partial fusion results. A missing deictic reference could e.g. result in highlighting all objects the user could point at.

6.6.3. Fusion Model

The fusion model allows the configuration of the fusion component and acts as storage for fusion results. It bridges fusion and the core models, which makes any partial results of the fusion transparent and observable for other models and components. This bridges the gap between the internal state of the fusion component and the state of the other models.

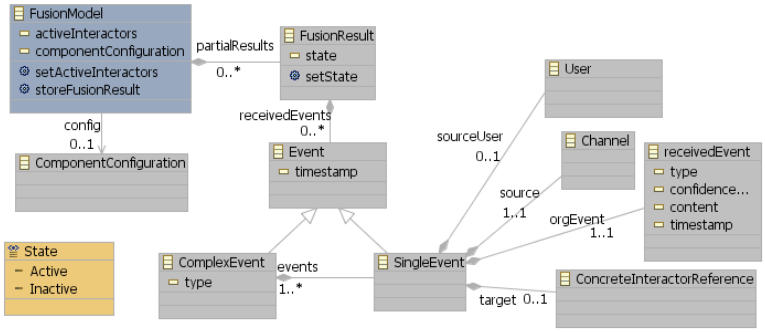

Figure 6.8.: The fusion model, storing (partial) fusion results and configuring the fusion component.

The fusion model, shown in figure 6.8. Besides the configuration provided by the developer, it holds fusion results in form of input events. An event can either be a complex event consisting of multiple single events, e.g. for complementary or redundant input, or

a single event. Each single event consists of the originating user and channel as well as a representation of the actual received input. Additionally, the concrete interactor, the event has been assigned to, is referenced in the model. This allows the fusion model to reflect partial fusion results which can be mapped to the concrete interaction model for semantic interpretation. Additionally, it allows other models and components to consider (partial) fusion results e.g. to adapt the rendering of the UI.

6.6.4. Input Interpretation Sequence

The resulting input interpretation process based on channels, fusion component and model is illustrated in figure 6.9.

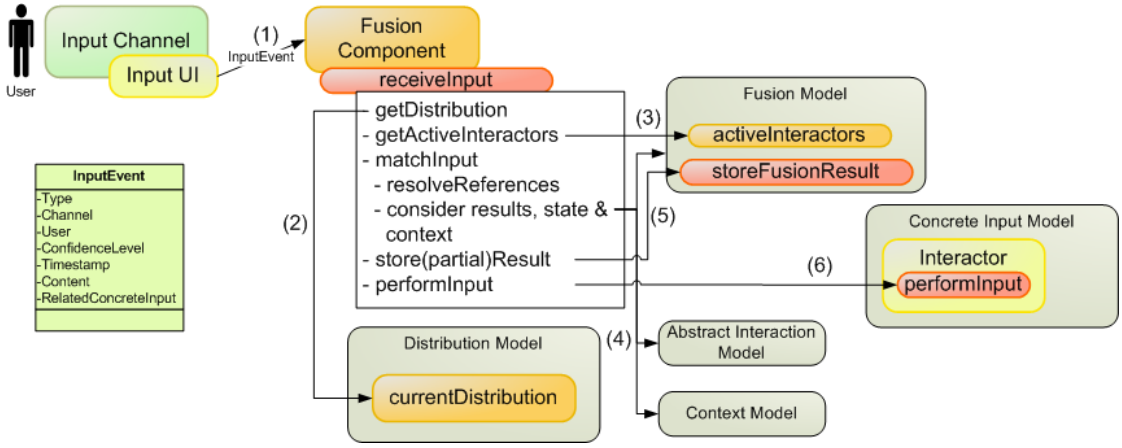

Figure 6.9.: Illustration of the sequence of the basic steps of the fusion process and the relation of the involved models.

After performing any input via an interaction resource, the input is received via the channel and preprocessed according to the input user interface. This results in an input event provided to the fusion component (1), containing the received input and additional information. Based on the received event, the fusion component is now responsible to find the instance of the concrete input interactor this event addresses. It therefore checks the current distribution (2) and the active input interactors (3). During the matching process, references e.g. within voice input are resolved and information from additional models like previous (partial) fusion results, the current interaction state and context information (e.g. the interaction history) are considered (4). Once the element has been matched it is stored as partial result in the fusion model, depicted in figure 6.8 (5). This fusion result combines the received input event data with details about the addressed

concrete input interactor. If the fusion result can be matched to a concrete interactor, the *performInput* method of that interactor is triggered and the received input is passed (6). This triggers the processing of the input in terms of the modeled interaction means and finally results in a state change of the user interface and possibly a service call. If the identified concrete input interactor is part of a complex input interactor, awaiting complementary or redundant input, the fusion engine processes further input to fill all defined slots (children) of the interactor before it activates the *performInput* method of the interactor.

6.6.5. Summary

In summary the described fusion process allows the processing of multimodal events, pre-determined by a designer. It can be compared with a frame-based slot-filling process and strongly utilizes the CARE relations defined in the concrete input model. Once the received input events have been matched to expected input modeled in the concrete input model, the corresponding executable elements of the interactors are called, triggering the event processing within the model. This causes a state change within the model, resulting in the triggering of additional events, which again influence the states of other models. In this way information is transported from the concrete to the abstract and then to the conceptual task and domain level and step by step translated into machine processable input until it results in the execution of backend logic of the application and the entering of the next interaction state of the user interface model. The derivation of meaning from user input in the MASP is thus based on the combination of the model-based approach with a fusion process, matching multimodal input events. While the underlying user interface model allows to restrain the interaction and to interpret received input based on the current state of the interaction, the fusion process takes the modeled information into account while matching input events received from multiple interaction resources.

Before this underlying multi-level event propagation process is described in section 6.8, the utilization of the adaptation model to adapt the user interface to different contexts of use is discussed in the next section.

6.7. User Interface Adaptation

The adaptation of user interfaces and interaction to context of use information aims at improving the interaction experience and usability of the application by incorporating

external information into the interaction. However, this requires capabilities to alter the adaptation according to foreseen as well as unforeseen situations.

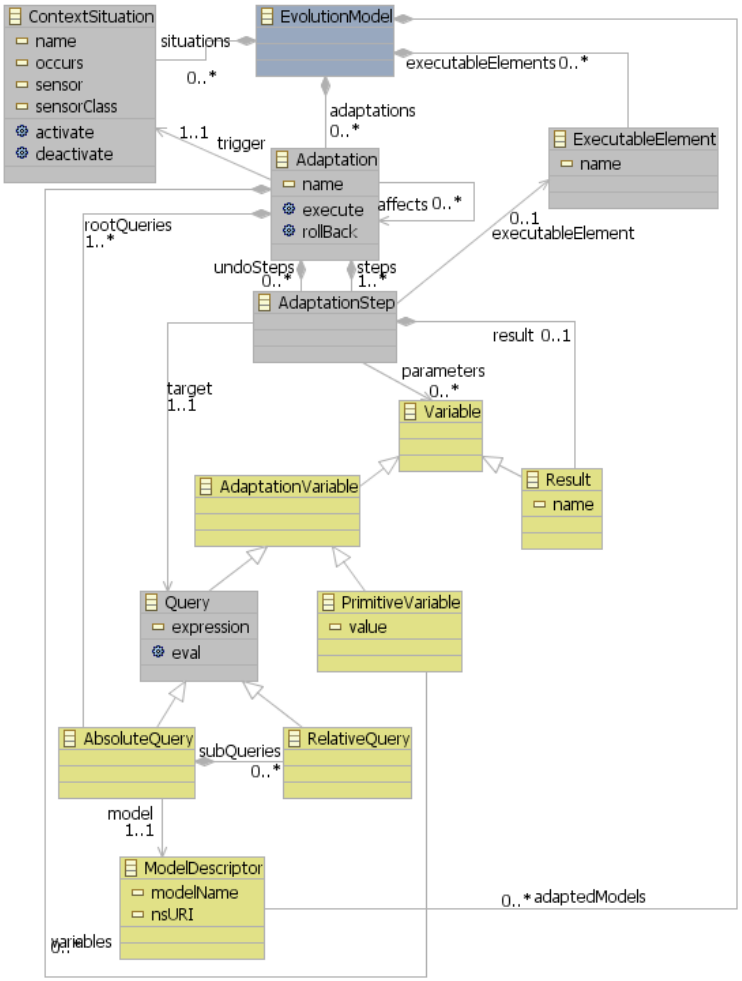

Figure 6.10.: The executable adaptation model.

In the following the adaptation model is described, allowing the definition of model adaptations that alter existing models based on available context information. The approach is based on the adaptation metamodel and the executable nature of the meta-metamodel and its definition modification elements. Due to the capability of the metamodel to alter elements of the realized models, the adaptation model is able to alter the structure of any model of the user interface according to the defined adaptations. While this approach is very powerful and allows the creation of an adaptation layer that can be defined without

having to touch the underlying models, it has also to be deployed very carefully to not break the application through careless adaptations.

The adaptation model defines manipulations to the provided user interface models and is able to alter the design of the user interface. As definition modification elements of the meta-metamodel provide the means to adapt any kind of model (obeying to an metamodel), the adaptation model references these elements to perform adaptations. Context situations act as triggers to perform adaptations with multiple steps. The basis for the triggers is provided by the context model, sensing context information and making it accessible for the application. The utilization of the adaptation model, the executable elements and the context information is described in the following. The adaptation metamodel is depicted in Figure 6.10.

The model is composed of *ContextSituations* and *Adaptations*. Similar to Sottet et al. (2007a) it has been defined according to the Event-Condition-Action principle, and thus the *ContextSituations* act as triggers for the adaptations. An *Adaptation* consists of multiple steps (represented by *AdaptationStep* elements) applied to model elements referenced in form of XPath-like queries (modeled as the *rootQueries*). Adaptation steps reference executable elements (situation modification or definition modification), which can be parametrized with variables (queries to model information, primitive values or execution results). The elements of an adaptation step are described in detail in the following.

- The **target query** represents the left hand side of the adaptation step by identifying the nodes the adaptation has to be applied to. The query allows referencing any element or set of elements in a model. In case of EMF-/XML-based models XPath expressions are utilized to define queries.
- The **executable element** provides the right hand side of the adaptation step by referencing executable elements of the target model. When the adaptation step is executed the specified executable element is invoked upon the model element referenced by the target query. The utilization of executable elements for the modification of models ensures a metamodel conform adaptation.
- **Variables** are used to parametrize the executable elements referenced by adaptation steps. A Variable can be a query to some model elements (ideally relative to the root query as this ensures reusability of the adaptation), a primitive variable holding fixed values (for example a predefined integer) or a result of the execution of an adaptation step. During execution the variables are evaluated and their values are passed to the executable elements.

- Any **result**, returned by the executable element of an adaptation step can be stored in the *result* variable and reused in the following adaptation steps.

In summary, an adaptation of a model (or multiple models) is defined by the node(s) to apply the transformations to and a description of the alteration of these nodes. The alteration of the nodes is defined in form of definition modification elements, that are provided by the metamodels itself and are specific for each given metamodel.

Utilizing executable elements, defined by the metamodel of the model to adapt, ensures compliance to the metamodel, as any applied modification is well defined and by definition conforms to the metamodel. It is thus the responsibility of the developer of the metamodel to ensure that no operation performed by an executable elements leads to an invalid model. This becomes even more important from the runtime perspective, where models are adapted while being executed. The utilization of definition modification elements guarantees, that the model remains executable while being adapted. In combination with executable models the execution of the adaptations at runtime also allows building adaptations, that take the situation elements of the model into account. This allows to flexibly react to the current state of the application to improve the quality of the adaptation. The defined adaptation model additionally allows the creation of reversible adaptations by the means of undo steps, which allows applying adaptations based on a recognized situation and applying counter adaptations if a situation is not active any more. This supports the definition of adaptations that ensure that the original state of the application can always be reached again.

One remaining issue is that adaptations can lead to a dead end, i.e. a state that no other adaptation can be applied any more. This also holds true for the application of adaptations to an already adapted model which might be difficult to foresee for the designer. However, checking all applied adaptations (adaptations with an active context situation) when applying a new adaptation allows reapplying them to the newly adapted model if necessary. For performance reasons the *affects* relation has been introduced. It denotes other adaptations that have to be reapplied after the application of the current adaptation – of course only if their triggering context situation is still valid.

Performing adaptations is strongly connected to various other models, acting as target for the adaptation, when being adapted or as trigger, initiating an adaptation. To trigger the execution of an adaptation, the system continuously monitors the context model for situations specified as triggers. Once such a situation is detected, the execute method of any related adaptation is called. This leads to the performance of the defined adaptation steps, calling the referenced definition modification elements, which results in an

alteration of the underlying user interface model. However, any model only provides well defined alteration means (in form of DMEs), that can be utilized to manipulate the model. Any changes to the models are reflected in the user interface, as there is a continuous synchronization between the state of the model and the presented interface. Based on this mechanism, the user interface developer can define adaptations to user interface models, altering the user interface according to context information.

The adaptation model completes the overview of the architecture specific aspects of the utilization of executable user interface models to create multimodal user interfaces and handle them at runtime. To summarize the utilization of the approach, the next section gives an overview of the interplay of all described elements.

6.8. MASP Event Propagation

In this section a complete interaction cycle within the MASP is described, based on an exemplary configuration. Starting with the registry of interaction devices, the request of an initial user interface, the creation and delivery of that user interface to multiple interaction resources, the perception of multimodal user input and its implications to the state of the interaction and the final update of the presentation according to the user input are illustrated.

After the MASP has been started and loaded the models of the available applications defined in the core model, the context model is ready to receive context information via the registered context providers. Any interaction device can now be found via UPnP or connect to the client discoverer to register its interaction resources with the MASP. For each registered interaction resource the MASP creates a channel allowing the direction of output to that resource and the receiving of input from it in case of an input resource. This approach has to be supported by the interaction device, which is ensured by a small, device specific application installed during or before registration. For the following example, a voice server registering itself by the MASP is assumed, which results in a Natural Language Input Channel and a Natural Language Output Channel being established. Additionally, a browser registers to the MASP, providing mouse, keyboard and a graphical display. The browser opens an URL pointing to the device observer and receives an initial web page, providing a initial MASP (Meta-)UI as well as some Javascript functions allowing the direct processing of mouse events and keystrokes as well as the adding and removing of HTML elements from the browsers underlying Document Object Model (DOM) Tree. The initial UI allows the user to start any of the available applications as

well as to configure the active IRs, the distribution, migration and personal preferences.

The user starts an application from the list and is provided with the initial view of the application. Behind the scenes, the MASP registers a new session with the MASP core model and sets the model in the initial state for this session. This activates the first state of the task tree and the related domain objects. Initial service calls are executed if any service task is present in the initial set of enabled tasks. Additionally, the abstract interaction elements related to the interaction tasks are activated (state is set to *active*) and filled with the dynamic information from the domain model. The mappings between the abstract interaction model and the domain model ensure that any element referencing information from the domain model is immediately updated as soon as the domain model object is altered. Based on the active set of abstract elements, the mapped concrete input elements and the concrete output elements are activated (state is set to *active*). As soon as these elements are activated, the *add* method of the distribution model is triggered, scheduling the new elements for the distribution process. This in turn triggers the distribution component to assign the different interaction elements to the available and active interaction resources, including the mouse, keyboard, screen as well as voice input and voice output.

Once a possible distribution has been calculated, the state of the elements is set to *assigned*. This triggers the arrangement of the concrete output elements by the shaping component for each channel. Afterwards, these enriched elements are set to state *readyToDeliver* and sent to the assigned channel, using its *add* method. This results in the channel rendering each received element on the provided position. Similarly, the input user interfaces are created from the concrete input elements and assigned to the related input channels. For the browser based device, this results in a configuration of the Javascript to send only events for some selected keys, the system can handle in the current state as well as the configuration of events for certain relevant areas for the mouse. A gesture input channel established to handle point and click gestures of the mouse maintains all relevant coordinates of the clickable elements, including buttons and the elements of a list presented on the screen. Additionally, the natural language input channel is provided with a grammar identifying known utterances and the events each utterance creates. The natural language output channel is used to provide the user with a short greeting message.

Once all concrete interaction elements are rendered, the user can interact via speech, mouse and keyboard. As first input the user utters "remove that element" and clicks with the mouse on a list item. The system processes the voice message, which is translated

to the selection of the remove option from the list of possible actions to carry out. Additionally, the remove action requires the selection of an element from the list. Thus the mouse click to the object is interpreted as a select event to that list item. Both events are processed in combination by the fusion engine, arranging them in the right order to actually perform the task. The fusion of the events thus results in the update of the state of the item to *Selected* and afterwards in the update of the state of the "remove" item from the list of possible commands to *Selected*. This update triggers a task done event, setting the "remove item" task to done, which in turn triggers the execution of an application task removing the item from the list stored in the domain model. After the execution of the application task, the task tree iteratively moves into the same state as before.

The update of the domain object now results in an update of the list presented on the screen with the new content from the domain model. To perform this, the related *choice* abstract interaction element is notified about the change of the associated domain object, relating in an update of the related concrete UI objects, related to the updated abstract element. This way, the updated information is transformed into a format perceivable for the user and finally results in an update message transported to the user via the channel that maps the update information to a DOM manipulation of the HTML page in the browser.

6.9. Summary

In this section, the architecture and selected features of the Multi-Access Service Platform have been described. Underlying the described approach is the goal to create Ubiquitous User Interfaces for smart environments and to combine the architectural elements with the concept of executable models and the utilization of the developed reference models.

It has been shown, that it is possible to combine the models with lightweight components at runtime, to address issues like shaping, distribution, fusion and adaptation. Based on the utilization of the set of models and the exploitation of mappings between them, the state of the user interface can be continuously derived and conveyed to the user. Presentation of the user interface and perceiving of interaction are realized by connecting independently addressable interaction resources via channels and continuously synchronizing their state with the state of the model. Based on the description of abstract interaction and concrete input and output facilities, the distribution of the user interface as well as the processing of multimodal fusion are supported. Additionally,

well defined hooks to alter the models via definition modification elements, allow the definition of adaptations of the user interface models to runtime needs, which makes the design decisions of the user interface directly manipulable.

The described Multi-Access Service Platform realizes a comprehensive model-based runtime environment for the interpretation of user interface models with a focus on Ubiquitous User Interfaces and smart home environments. To evaluate the described concepts and infrastructure, various demos and prototypes have been developed. In the next chapter, two selected prototypes will be described, each evaluating different aspects of the concepts and architecture.

7. Evaluation

The Multi-Access Service Platform has been under development since 2003 and has been revised multiple times. Consequently it served as basis for various demonstrations and evaluations and supported the continuous testing and revising of ideas and concepts. This chapter introduces two case studies and a validation against the original requirements to evaluate the approach to build Ubiquitous User Interfaces.

Both case studies aim at the realization of the concepts developed throughout this thesis. The first case study focuses on the realization of the identified features from a user perspective, allowing the creation of multimodal user interfaces that can be distributed and adapted dynamically within a smart environment. While it realizes the necessary infrastructure to do so, it has some open issues within the utilization of the underlying user interface models and thus the flexibility of the approach. The prototype of the second case study addresses these issues and sets a much stronger focus on the development of Ubiquitous User Interfaces and the developer perspective. It comprises the described set of metamodels, realized as executable models with the Eclipse Modeling Framework. This allows the formal definition of user interfaces in terms of models and mappings between the models. The major work for the different case studies has been done within the Service Centric Home project, which has been conducted with several industry partners and sponsored by the German government. The projects main goal was the creation of infrastructure for the development and deployment of assistive services for smart home environments.

After a brief introduction of the Service Centric Home in the next section, the two case studies and the developed applications are described. In section 7.4 the developed approach is evaluated against the originally defined requirements and an overview about how these requirements have been met is presented.

7.1. The Service Centric Home

The Service Centric Home (SerCHo) project has been running for three years, from 2005 - 2008, with a strong focus on the creation of an integrated platform for the development and provisioning of services for future smart home environments. An important part of the project was the development of integrated user interfaces in form of assistants, that support the user with different tasks at home and tools for their creation. Another major aspect of the project has been the creation of the Ambient Assisted Living Testbed, a smart home environment for evaluation-, testing- and demonstration purposes. Consisting of four rooms (figure 7.1), a kitchen, a living room, an office and a studio, the environment comprises numerous devices and appliances which are all interconnected via (wireless) LAN connections. The rooms are equipped with multiple interaction devices including touch screens, remote controls, gesture recognition devices, voice recognition and camera systems, sketched in figure 7.2. Cameras for gesture recognition and positioning of the user are located on the ceiling of each room as well as besides most of the indicated screens. All screens are connected to a PC to be controlled and run a web browser in the current setup. The available speakers are also connected to PCs running SIP clients or RTP renderers for voice output. Voice input is currently realized via a portable Sennheiser headset the user is carrying. In addition to the available interaction resources the environment has been equipped with various sensors and controllable appliances. An Ubisense localization system is capable of detecting the users location with an accuracy of about 30cm.

Figure 7.1.: The four rooms of the SerCHo Laboratory.

This environment served as development and evaluation testbed for all developed applications. Installed on the central home server, the Multi-Access Service Platform plays a central role for the delivery and distribution of Ubiquitous User Interfaces for this smart home environment.

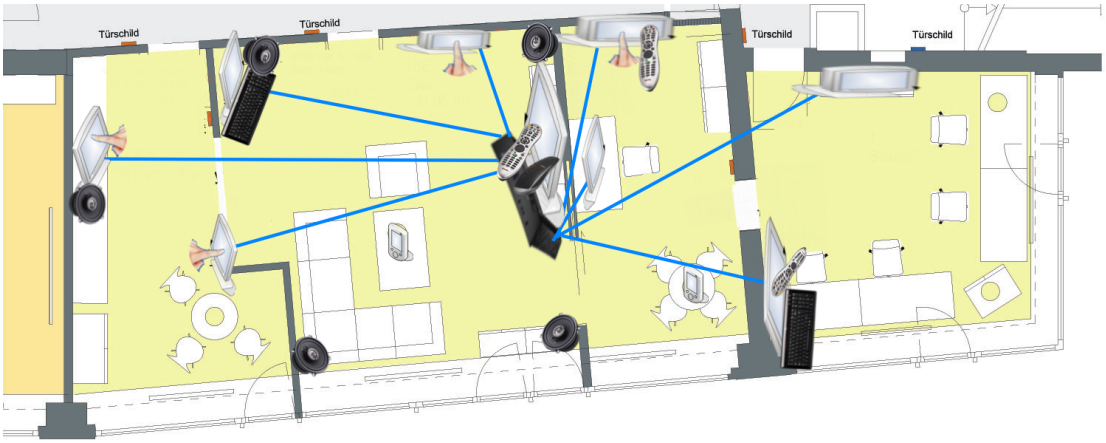

Figure 7.2.: Ground plan of the lab with sketched interaction devices (screens (touch is indicated by the finger), keyboards, remote control, speakers (a microphone is worn by the user) and PDAs), all connected to the home server running the MASP.

Two typical scenarios within a smart home environment are the control of the home appliances and the usage of additional services via the various interaction resources. In the first case, the MASP provides a home control user interface that can be utilized to switch appliances on and off, check and set the room temperature or schedule events like turning on the washing machine early in the morning. From the user perspective, realizing such an application with a Ubiquitous User Interface has the advantage that it would be available everywhere in the house. It could be controlled via various interaction resources and multiple modalities. Multi-user facilities are required as every inhabitant could use the home control application. In the second scenario, an additional application provides enhanced services while exploiting the facilities of the smart home. A cooking assistant supports the selection of a recipe, the search for available ingredients, handling of a shopping list and the step by step instruction of the cooking process. While sitting on the couch, the inhabitants can comfortably select a recipe using the TV and the remote control. Checking the available ingredients afterwards, the ingredients list can be send to the kitchen and read out via voice, while the family members search for each ingredient. Missing ingredients are added to a shopping list, which is directly synchronized with the mobile device of any family member, stopping at the supermarket on his way home. He/She can then check every bought element off the list, which is synchronized with the home displays where the others can follow the progress. The people at home can also add additional elements which directly appear on the list. When the groceries arrive,

cooking can begin and the assistant guides step by step through the process e.g. via voice and graphical output. Interaction within the kitchen can be realized via voice, touch and gestural input for the maximum convenience. Multi-application support allows the integration of the home control within the cooking assistant e.g. to comfortably program the oven. Additionally, a notification when the meal is ready can be received anywhere in the home and the oven can be turned off remotely.

7.2. Case Study: Infrastructure for UUIs

In this case study, the first prototype of the MASP has been developed with a strong focus on the quick realization of Ubiquitous User Interfaces within the home environment. It was the goal to develop a central server, able to host multiple applications and make these applications accessible via the available interaction resources. While this sounds similar to the task of a web server, there are indeed some parallels that have been utilized for the approach. Consequently, the MASP was developed as a web application inside an Apache Tomcat Web-Server, running on the central home server. This allows the easy connection of multiple clients via HTTP and supports the utilization of existing browser and web server technology.

The utilization of user interface models for this approach was a requirement from the very beginning. The implementation is thus based on a task model, a service model and the definition of the interaction means. The presented user interfaces have additionally been styled using a layout model, templates and CSS to beautify the demonstration and embed them into the smart home. Several main concepts have been integrated and evaluated within the demonstrations. This includes:

- a workflow description based on the interpretation of a task model at runtime,
- exchanging information between (hierarchical) models for synchronization purposes,
- accessing backend functionality via the service model,
- the independent creation and synchronization of voice and graphical interface,
- the utilization of channels to address various interaction resources,
- the creation of input events mapped to model alterations,
- context integration and consideration at runtime,
- the realization of different layouting features e.g. based on the task information,

- migration and distribution features and the integration of follow-me functionality.

While this list is by far not complete, it gives an overview of the implemented features. The first prototype had a strong focus on the realization of the basic concepts of Ubiquitous User Interfaces to gain experiences with the infrastructural needs, required for the realization of the features. It also served as a test and demo system allowing user studies and evaluations.

Based on this realization of the basic infrastructure, assistive applications with Ubiquitous User Interfaces have been implemented. This comprises a cooking assistant, an energy assistant and a Meta User Interface. The developed 4-Star Cooking Assistant (SCA) has been deployed in the kitchen of the testbed, supporting the search for recipes, the seeking of and shopping for the required ingredients as well as the cooking process. It provides voice interaction and can be used via the touch screen in the door of the kitchen cabinet. A shopping list can be distributed to and synchronized with a mobile device. The Smart Home Energy Assistant (SHEA) is based on the idea to visualize the energy consumption of the devices in the smart home and remotely control them. It also supports voice and touchscreen interaction, as well as utilization via a remote control. As third application, the Meta User Interface (Meta-UI) can be controlled via different modalities and provides configurational access to MASP features like modality control, migration, adaptation and distribution. All three applications can be seen in figure 7.3. A detailed description, illustrating the implementation details of the MASP and the underlying concepts can be found in appendix A.

In summary, the created MASP infrastructure utilizes a task-, domain-, service-, and layouting model, as well as rendering templates and interaction mappings to provide flexible interaction and various configuration possibilities for the user interface, by providing full control for the application developer. Using an infrastructure of connected tuple spaces in combination with interaction channels allows to convey the internal state of the applications to the outside world and the continuous synchronization between state and presentation. The context model and a distribution algorithm, encapsulated in the distribution component, allow the flexible distribution of user interfaces and the integration of context information. Additionally, a layouting implementation ensures that the shape of the user interface is always properly adjusted to the used interaction resource. Multimodality is supported by additional voice channels and rendering templates, providing voice user interfaces.

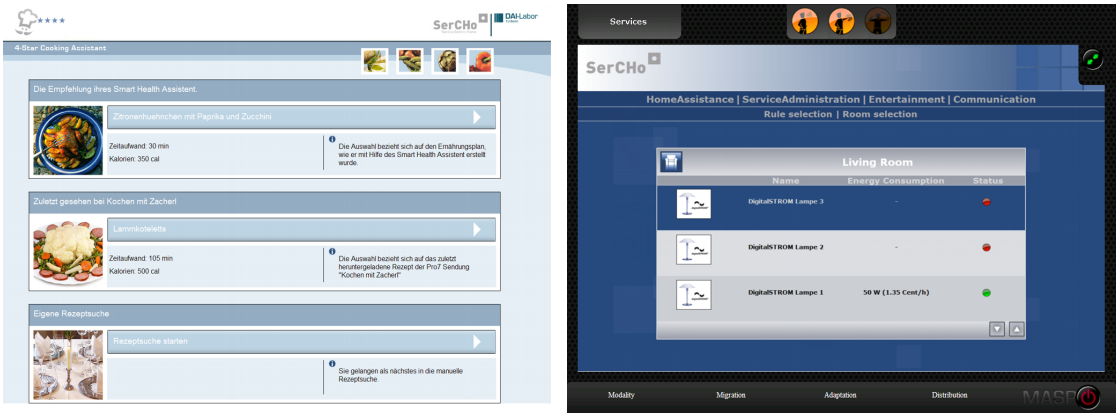

Figure 7.3.: The 4-Star Cooking Assistant startscreen (left) the energy assistant called from within the meta UI (right).

The major limitation of this case study was the limited expressiveness of the utilized models. While a comprehensive infrastructure has been developed and proven to be suitable for the development of Ubiquitous User Interfaces, the provided features are limited to the task level of the user interface and suffer from the loss of semantics in the used templates. While it is possible to define very flexible templates, as HTML and Javascript provide some flexibility per se, some limitation apply. The case study introduced in the next section, thus addresses these issues and evaluates the utilization of executable models to provide a more expressive definition of the user interface description, resulting in a better adaptation, more detailed distribution possibilities and stronger multimodal interaction.

7.3. Case Study: Executable UI Models

The second case study has been conducted with a much stronger focus on the utilization of executable models and the strict realization of user interfaces via an EMF-based implementation. The main concern of this part of the evaluation was to ensure that executable models can be implemented in EMF and that the models play well together. Concepts in the focus of this implementation were:

- the utilization of the complete set of models,
- the realization of the models as executable models,
- the interconnection of the models using mappings within the mapping model,

- the integration of models and external components,
- bootstrapping and model management within a model-based runtime system.

The developed system realizes these concepts with a set of different models (task-, domain-, service-, abstract interaction-, concrete input-, concrete output-, shaping-, distribution-, fusion-, context-, adaptation-, MASP core-, and mapping model) that are all connected via mappings. Based on the features realized in the first case study, the same applications have been modeled using EMF models. While the workflow is again defined by a task model, this model, as well as domain- and service model, is now completely realized as an executable model in EMF. An interaction model describes the basic interaction means and allows the derivation of user interfaces. The integration of context information and the delivery of user interfaces to the interaction resources are again realized via the channel-based distribution mechanism and the integration of the existing context model. EMF has been chosen as modeling notation, providing a modeling and code generation framework integrated into the Eclipse platform. It is widely used and provides comprehensive tool support. Figure 7.4 shows a screenshot of the Swing-based user interface of the cooking assistant (on the left) and a screenshot of the Eclipse editor, showing a live view of the model at runtime. A detailed description of the implementation can be found in appendix B.

From the developer's perspective, the utilization of executable models and the EMF technology provides access to the models at runtime and makes the connection between the model and the derived UI more transparent. Besides a life view of the runtime models, changes to the models can be directly made at runtime and are immediately reflected in the UI. Utilizing EMF, provides generic editors for this purpose, but also allows the easy creation of more powerful and customized editors for specific purposes. While executable models and runtime development are still not able to hide the existing complexity from the developer, they provide a big step towards making model-based approaches more transparent and usable.

In summary, this second case study showed how the features of the EMF ECore model support the implementation of models and their mapping to executable Java code. Different executable models have been implemented and connected via mappings to create a UI model, observable at runtime. Additionally, layouting- and context-model as well as a basic implementation of the adaptation model have been realized. Reusing the distribution component and the concept of channels allowed the creation of user interfaces and the evaluation of the approach. The developer's perspective has been studied and transparency and runtime development are supported. However, while the approach is

promising it does not yet ensure the same quality of the derived user interface as the application of rendering templates with more specific user interface code used in the first case study. This is mainly caused by the not yet fully developed interaction model, which is the major element for further extensions of the implementation. Additionally, more powerful tools and a mature development methodology are needed to better support the developer.

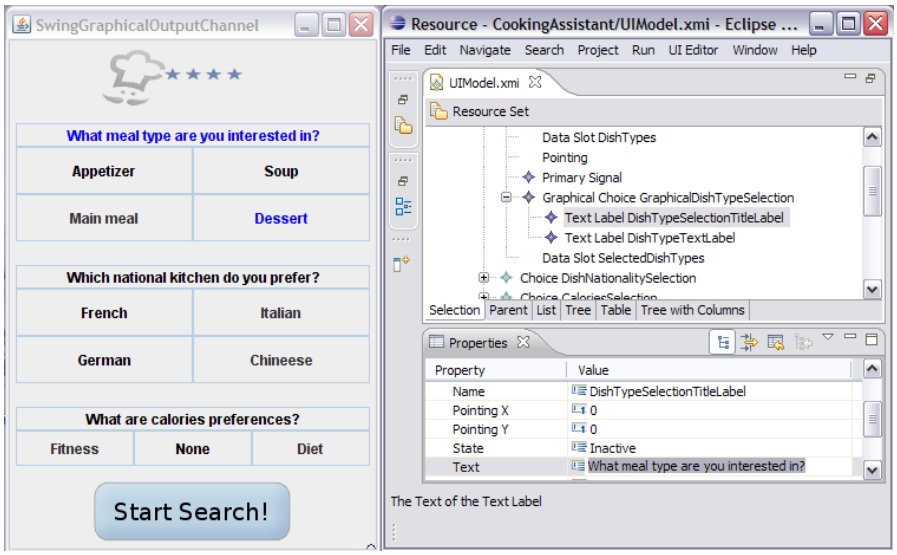

Figure 7.4.: Screenshot of the graphical user interface of the recipe finder step of the cooking assistant, that is derived from the executable model. To the right, a view of the Eclipse environment, monitoring the model at runtime, is depicted.

7.4. Requirements Validation

After the illustration of the two case studies and their results, this section validates the developed concepts with respect to the features of UUIs and against the original requirements identified in section 3.5.2. Besides, shapeability, distribution, multimodality, and adaptation, architecture- and UIDL concepts are discussed.

7.4.1. Shapeability

The shaping of the user interface is required to reflect IR capabilities and create an appropriate user interface even for a configuration unknown at design-time. Besides automatic shaping, special optimization for predefined settings should be supported.

Throughout the work, the adaptation of the shape of a user interface has mainly been addressed for graphical output user interfaces, which it is most relevant for. As described in section 6.5, the developed architecture comprises a shaping component, that adapts the layout of the user interface to the needs of the used interaction resources. A shaping model defines statements, relating model elements to properties of the presentation in terms of containment, order, size and orientation of the elements. At runtime these statements are translated to constraints that are solved to determine the arrangement of the user interface elements within the graphical presentation. The boundaries of the possible shaping are set by the user interface models in terms of relations between the elements, defining temporal and spatial constraints as well as multimodal CARE properties within the complex interactors.

Besides graphical output, additional temporal constraints can be considered for voice input and output. On top of the means to shape the user interface, provided by the layouting mechanism and the complex interactors, an adaptation model has been created to address more complex adaptations. It is described in section 6.7 and allows altering the underlying user interface model according to well defined adaptation rules. These alterations are expressed using the definition modification elements of the metamodels and are assembled into a dedicated adaptation model, which defines specific context information as trigger for these adaptations. Different aspects of the context of use, including e.g. the screen size of the used interaction resource, location or user preferences can be incorporated into this process.

7.4.2. Dynamic Distribution

The dynamic distribution of the user interface across interaction resources was one of the fundamental requirements of the presented approach. It provides the basis to address multiple modalities via multiple interaction resources and to flexibly incorporate any interaction resource within a smart environment. While input and output IRs should be handled independently, on a semantical level, the input and output UI elements are strongly interconnected. The synchronization of (distributed) presentation with the internal state of the user interface models as well as the synchronization of multiple presentations with each other were required to be supported.

Dynamic distribution has been realized by the combination of the distribution model with the distribution component (section 6.4) and supported by the underlying interaction model (section 5.4). Additionally, the independent handling of interaction resources via

context model (section 6.2) and interaction channels (section 6.3) is crucial to handle the distribution at runtime.

The basis for the distribution is provided by the interaction models, separating abstract interaction as well as concrete input and concrete output. This provides the required separation of input and output, while still keeping the semantic connection via the abstract interaction. Additionally, the defined atomic and complex elements identify the main building blocks of the user interface that can be independently distributed. Besides these elements, the developed approach allows the consideration of task relations as well as CARE properties, temporal and spatial relations as constraints for the distribution.

Distribution means are defined by an additional distribution model, that holds distribution configurations. These configurations are created by user, application or the system and are incorporated into the distribution process in this order. Once the distribution has been calculated for all currently active elements of the user interface model, the elements for each interaction channel are passed to the shaping component for arrangement and styling. Afterwards, the channel takes care of their suitable presentation on the connected interaction resource. While the elements are presented, any state changes are continuously communicated to the interaction resource and reflected by the presentation.

Providing the capability to store the current distribution within the distribution model also allows the consideration of the current distribution configuration within other models at runtime. This allows to reference the current distribution configuration within the user interface to e.g. direct the attention of the user to a specific interaction resource or modality. As the developer can not foresee any distribution of elements at design-time, variables, that can be filled e.g. with human understandable names of interaction resources at runtime, address this issues. The additional issue of resolving such references used by the user, is addressed by incorporating such variables not only into the output user interface, but in the input user interface as well.

Major aspects of the distribution of user interfaces have been evaluated in the two case studies and during experiments with the developed system in the Ambient Assisted Living Testbed. The realized Meta-UI e.g. allows to freely configure the distribution on a task basis. However, using this features is rather complex and not yet suitable for the end-user. A more suitable approach seems to be the embedding of distribution features within the applications as for the shopping list of the cooking assistant.

7.4.3. Multimodality

A main requirement of the approach was the utilization of multiple modalities to support robust and natural interaction. From the perspective of the user interface description, this required the definition of modality independent aspects as well as modality and device specific aspects. Additionally, the flexible combination of interaction resources and input and output modalities required the separation of UI parts (fission) and of input and output within the UI description. Fusion is a major aspect to process user input from different modalities, which can be supported by the definition of modality relations and semantics within the user interface description.

Multimodal interaction is deeply integrated as a key feature and addressed by various aspects in the developed approach. Initially, the identified architecture provides strong means to address input and output independently and to integrate interaction resources independent of the connected devices. This results in the ability to flexibly utilize multiple modalities, based on the idea of distributing interaction elements across various interaction resources at runtime (section 6.4). The underlying fission of the UI building blocks is supported by the separation of modality independent task- and abstract interaction model and modality specific concrete input and output interaction model on the UI description level (section 5.4). The handling of the UI building blocks and their assignment to interaction resources is then handled by the distribution component via the channels (section 6.3).

A fusion component (section 6.6), receives the user input events from the channel and can apply different fusion mechanisms. The utilization of this component is strongly connected to the user interface description, which defines interaction semantics as well as the intended relations between the modalities in form of the CARE (complementary, assignment, redundancy and equivalency) properties. This strongly supports the creation of input user interfaces, that pre-process the user input with respect to the current interaction goals of the system. The component is supported by a fusion model, that stores partial fusion results to make them available to other models and e.g. allow their reflection within the user interface.

The utilization of multiple modalities has also been evaluated within both case studies, setting a focus on the combination of graphical output with mouse, keyboard, touchscreen and voice interaction. It has however to be noted here, that the application of the CARE properties at runtime has not been evaluated in detail yet within the presented approach (but e.g. in (Bouchet et al., 2004)). The realized case studies mainly supported equivalent input, reducing the need for fusion and semantic input processing.

7.4.4. Adaptation

Supporting interaction within smart environments with changing contexts of use, also lead to the need to adapt the interaction to dynamically reflect user preferences and capabilities, interaction resource capabilities and the environment. A major aspect to support adaptation was the utilization of comprehensive UI models, which explicitly reflect the state of the interaction at any point in time. This also supports the persistent handling of this state as well as of user input and output. The handling of context information, including the available interaction resources, is crucial to trigger the required adaptation steps.

Adaptation has been defined as the possibility to alter the configuration of the user interface features at runtime (section 2.3.7). It is addressed by the various components in terms of the alteration of the shape of the user interface according to changing contexts or configurations, the flexible distribution of the user interface, that can be altered at runtime to support migration of partial or complete user interfaces, and the dynamic utilization of interaction resources and modalities. Additionally, an adaptation model (section 6.7) allows the utilization of executable elements of the models to alter their structure and adapt them to the context of use. The utilization of the definition modification elements (section 4.1) ensures that the performed adaptations conform to the metamodel of the altered model and do not render the model unusable. The utilization of a central adaptation model supports the coordinated alteration of multiple models to ensure consistency across the models.

The utilization of executable models (section 4.1) also ensures the explicit availability of both, state as well as transitions between states within the user interface models. Design-time definitions in form of definition elements are coupled with runtime information in form of situation elements and transitions between runtime states in form of executable elements. This explicit state information is also fundamental for the utilization of the user interface models to convey the current state of the interaction and define the behavior of the user interface. Additionally, it allows to ensure persistence on each of the different abstraction levels of the user interface. While the task level (section 5.1), stores the completion of any task, the continuous synchronization of the interaction model (section 5.4) with the actual user interface and the direct processing of any input, also ensure that interaction with any interactor is persistently stored. Dependent on the capabilities of the channel and the realization of the input user interface, the approach is also open for the storage of any atomic interaction, e.g. a typed character in a form field, as this can be directly processed by the interaction model as well.

Context and device management are addressed by the context model (section 6.2) and the utilization of channels to directly address any interaction resources. The executable context model comprises the definition of the available context information, which can be used for the specification of the user interface models at design-time, as well as proxy elements, that integrate the actual context sensing systems into the execution process at runtime. An EMF-based context model has been developed for this purpose. It allows to handle information about users, location, interaction resources and the established channels and integrates location provider and device discovery services to illustrate the usage at runtime.

Some adaptation capabilities have been realized in both of the conducted case studies. While case study one supported distribution, migration and adaptive user interface layouts, case study two additionally focused on the utilization of an adaptation model to alter user interface models at runtime. An explicit interaction state and the persistent storage of this state and any user input has been supported in both case studies.

7.4.5. Architecture Concepts

The developed runtime architecture (chapter 6) is based on the concept of executable models (chapter 4) and the defined metamodels (chapter 5). Using user interface models at runtime allows incorporating design information into the interaction. Making these models executable adds information about the state and the evolution over time and makes the model dynamic. From the perspective of the architecture (and of the developer), this allows to continuously observe the state of the model and to trigger state transitions by stimulating the model with external input. The main task of the architecture is to translate the internal state of the model into a perceivable user interface and to map the input received from the user to stimulations of the model that trigger state changes.

The developed architecture provides components to bridge between the model state and the outside world (section 4.2). Components integrate means to translate between model and outside world, e.g. distribute the UI elements to interaction resources or fuse user input to stimulate the model and create the related response. The channels, shaping, distribution and fusion work in this manner, based on the handling of the available interaction resources by the architecture. Additionally, the architecture exposes APIs (section 6.1) to access the models and realizes the bootstrapping and management of the executable user interface descriptions and provides the basic means to integrate the proxy elements into the user interface descriptions.

7.4.6. UI Concepts

The developed executable model concept has been applied to various metamodels throughout this work, to describe the different user interface aspects via specialized models on different levels of abstraction. Separating task (section 5.1) and domain (section 5.2), abstract interaction (section 5.4.1) and concrete input (section 5.4.2) and output (section 5.4.3) provides the basic means to support the definition of modality and device independent as well as modality and device specific interaction capabilities. Having these models available at runtime makes the design process and its underlying decisions completely accessible and understandable and supports the transparency of design decisions. The connection of the different models to a complex net via mappings (section 5.5), supports the information exchange at runtime as well as powerful means to interpret user input and derive user interface presentations. The model thus defines the structure of the anticipated user interface as well as its behavior over time, allowing the observation and stimulation of the model of the system under study (the user interface).

The concept of proxy elements (section 4.2) has been integrated into the models to allow the integration of external processes into the modeled domain. Based on these elements, the runtime integration of the functional core is provided by the utilization of the service model (section 5.3). This allows the definition of service calls to backend functionality via different technologies like Java methods or web services. Proxy objects reference the external services and realize service calls that are triggered by application tasks. Parameters can be passed from the domain model, where results are also stored. The service model has been extensively used in both case studies to equip the developed applications with actual functionality. Within the Meta-UI, the service model is used to connect the MASP API to the Meta-UI frontend. Similarly, the context model (section 6.2) supports the utilization of proxy elements to integrate sensor information and context observers.

7.5. Summary

In this chapter, two case studies, illustrating the usage of user interface models at runtime to create Ubiquitous User Interfaces and a comparison of the development results with the initial requirements have been presented. While the first case study focused on the foundations of the basic infrastructure and the creation of quick examples to show the potential of Ubiquitous User Interfaces, the second case study aimed at the realization

of executable user interface models to address shortcomings of the expressiveness of the models used in the first study. In combination, the two case studies complement each other and evaluate the infrastructure as well as the application of executable models. Different concepts like the levels of abstraction of the models, the exchange of information between the models, the multi-level interpretation of user input and the consideration of the internal state of the models to derive the user interface have explicitly been targeted in both approaches. Runtime development and tool support have been addressed by the second case study, but were not the main focus of this work. Additionally, the validation against the requirements showed that all major requirements have been addressed in the concept. However, some open issues as the expressiveness of the interaction models, the seamless integration of a fusion component as well as the interconnection of distribution, fusion, and adaptation within the implementation still have to be further elaborated and evaluated in detail. The next chapter concludes this work and points to future work to address the remaining open issues.

8. Conclusion

In this work, the concept of Ubiquitous User Interfaces to realize interaction within smart environments has been investigated. UUIs combine shapeability, distribution, multi-modality, shareability and mergeability as five main features to support human-computer interaction. Underlying these features is the observation, that five main dimensions drive the interaction within smart environments: multi-device, multi-modal, multi-user, multi-application, and multi-situation. This leads to a vast multiplicity of variants that a UUI has to cover. Aiming at the realization of UUIs for smart home environments, a lack of support for UUIs has been observed within the current state of the art. Three major shortcomings have been identified within the scope of this thesis:

1. the lack of a comprehensive approach to reflect the close interconnection of the features of UUIs,
2. a low level of integration between UIDLs and the results they produce at runtime,
3. some missing concepts of UUIs that were not supported by current UIDLs.

To fully address these shortcomings two interconnected elements have to be considered. The user interface description language, which allows the definition of user interface models to express the interaction means, and the architecture, which is needed to support the interaction handling at runtime and the mapping of the model to the actual user interface. This leads to the combination of an architecture for the handling of user interaction at runtime, with the concept of executable models to express the dynamic nature of UUIs. Based on a reference set of metamodels, the Multi-Access Service Platform integrates the metamodels and the underlying concept of executable models within the runtime architecture. The approach has been evaluated in two case studies, addressing the realization of the runtime system to develop and deploy UUIs in a smart home environment and the utilization of executable models to provide the means to express the dynamic nature of such user interfaces.

In summary, the main contributions of this work are:

the utilization of executable models to express the dynamic aspects of UUIs: Executable models have been identified as possibility to express the dynamic nature of UUIs, especially important for their runtime management. Consisting of static definition and dynamic situation elements, combined with construction means and execution logic in terms of definition modification and situation modification elements, executable models combine syntax and semantic interpretation within a single model.

the application of the concept of executable models to a set of reference metamodels: The concept of executable models has then been applied to define a set of reference metamodels for UUIs. These metamodels have been designed to express the interaction means with a focus on the utilization of the models at runtime to continuously describe and reflect the state of the user interface and thus the interaction between system and user. In this sense, they define the internal representation of the interaction means of the system. Mappings between the different models (and their concepts) link them together and facilitate the exchange of (state) information at runtime. The developed reference metamodels comprise the application of the executable models to state of the art concepts like task and domain model as well as an extended version of an interaction model, reflecting the requirements of adaptable distributed multimodal user interfaces.

the combination of metamodels with the architecture to create UUIs: The Multi-Access Service Platform has been presented as an approach to combine the concept of executable models, and its realization in form of the reference metamodels, with a runtime architecture. The MASP makes extensive use of the execution capabilities of the developed user interface models and projects the internal state of the model to the outside world to create Ubiquitous User Interfaces. It derives flexible presentations from the models and interprets user input according to the defined interaction means. The concepts of the Multi-Access Service Platform and thus the architecture, the executable models and the reference metamodel concepts have been extensively evaluated in two conducted case studies.

The described contributions of this work focus the strong connection of UIDL and the developed user interface models with the realization of user interfaces at runtime. Underlying this approach is the observation, that each system providing UUIs would be in need to maintain an internal representation of the user interface (ideally on multiple levels of abstraction and with various details) to present interaction means to the user and interpret the received user input. Utilizing the design model of the user interface for

this purpose has the advantage, that all taken design decisions are still available (and even revisable) at runtime. Thus, formalizing the design model and extending it with interpretation semantics has been evaluated as well suitable approach, to handle the increasing complexity of such user interfaces.

Reconsidering the user and developer perspective to the problem, the combination of user interface models with the runtime system, allows to focus on the user interaction and enhance it by providing a better knowledge about the concepts of the anticipated interaction instead of just a final user interface. From the developer's perspective, the utilization of executable models combines the expressiveness of the models with their execution semantics, making the model interpretation and behavior visible to the developer. Additionally, the dynamic nature of the models and the interconnection via mappings facilitates the definition of the behavior of the user interface and thus its dynamic aspects instead of providing a static snapshot of the system. Allowing to inspect and manipulate the underlying models at runtime provides the basis for new development approaches and makes the underlying processes more transparent.

The realization of the developed concepts within the two case studies, different demonstrations and implemented applications, provided deep insights into the details and open issues of the presented approach. The results have been build and evaluated as part of the Service Centric Home project and deployed in the Ambient Assisted Living Testbed at the Technische Universität Berlin. Additionally, various artefacts and the prototypes of the case studies can be found at http://masp.dai-labor.de. However, the work on this thesis and the application of the results to the development of UUIs within smart home environments also revealed several open issues and possibilities for future work.

8.1. Future Work

With respect to the state of the art and the identified shortcomings and requirements, some issues beyond the scope of this work remain to be explored within future work. They range from multi-user and multi-application scenarios to the integration of natural language processing and tool development.

From the perspective of this work, the next logical step would be the application and extension of the developed approach to support multi-user and multi-application scenarios. While the developed approach is able to host multiple applications and support multiple users, the information exchange between these instances is not yet supported. Merging applications to share interaction resources or support simultaneous usage by multiple

users requires conflict resolution strategies and concepts for user interface sharing and integration of application spanning UI elements, which have also to be reflected within the underlying models. To further extend the approach to real world scenarios, these aspects are essential.

Utilizing the architecture in smart environments also raises the issue of system performance and of distributing not only the user interface across interaction resources, but also the architecture across computing resources. As a major aspect of ubiquitous computing, the distribution of resources is crucial. Computing resources entering and leaving environments, mobile computing and heterogeneous environments lead to the need to expand the currently centralized approach to support multiple servers, raising coordination issues and requiring the application of distributed computing best practices. The service model has to be extended to cope with a broader range of dynamically appearing services that might be unknown at design-time.

While the developed Multi-Access Service Platform has been evaluated within a smart home environment and utilized to build several applications, it has put a main focus on the runtime issues raised by the realization of UUIs. The built applications have been tested and evaluated. However, yet they lack an underlying concept and best practices for the design of Ubiquitous User Interfaces. As the technology to create UUIs has just been developed there is now the need to explore how the features can be optimally deployed to address the users' needs. Similarly, the realization of UUIs is a big challenge for application developers and raises an urgent need for a mature set of tools to build and evaluate models for UUIs. While the implemented possibility to observe the state of the models at runtime via the Eclipse EMF is a first step, visualization techniques and optimized editors yet have to be developed. Stronger support for the creative aspects of user interface development has to be added and limitations of the developed metamodels have to be overcome.

Additional aspects like the integration of natural language processing facilities or extended distribution and fusion algorithms go hand in hand with the improvement of the produced UIs and the available development tools. Integrating more aspects from existing UIDLs into the presented reference models, uniting existing UIDLs, enriching the net of models with extended world knowledge, best practices or design guidelines and enhancing the capabilities to project the internal state to the presented UI can make the developed interactive systems more robust and natural.

8.2. Concluding Remarks

In this work the utilization of user interface models at runtime to create Ubiquitous User Interfaces has been investigated to address interaction within smart environments. Aiming at the integration of model-based technologies with runtime aspects, a meta-metamodel of executable models, integrating syntax and semantic in a dynamic model that evolves over time, has been presented. An architecture integrates a net of executable user interface metamodels to express the internal state of a Ubiquitous User Interface and provides means to convey this state to the user. User input is processed and mapped onto the model, stimulating internal state changes. The developed concepts have been implemented and evaluated in two successful case studies.

The work showed that executable user interface models can be deployed to cope with the increasing complexity and raising needs for flexible and adaptive user interfaces in smart environments. It also paves the road for the development of new methods for user interface design and development by making the internal state of interactive systems observable and accessible. Connecting multiple models into networks and bridging these networks and the world outside of the modeled domain has the potential to strongly influence the utilization of models within the design and implementation of any kind of software, interactive or not.

Runtime models make internal state and processes more transparent and blur the boundaries between design- and runtime. This opens the doors for understandable software and end-user development. It has the potential to take user interface design and development from an artistic process to a well-structured engineering approach and contributes to the idea to make software accessible everywhere, at everytime and for everyone. However, a crucial issue for the acceptance of model-based user interface development is to overcome the constraints the models pose to the developer. While first steps have been taken, it is still a long way until the creative output of user interface designers can be directly mapped to formal models processable by computers.

A. Case Study: Infrastructure for UUIs

The main focus of this case study was the realization of UUIs within a smart home environment. The challenge was thus to develop an architecture, implementing the developed concepts. However, initially a focus was set on the technical realization of the infrastructure relevant aspects of the approach, namely shaping, multimodality, migration and distribution, and context integration. While the utilization of the core models was an important aspect, it has been limited to a task-, domain-, service-, and layouting model as well as the integration of a context model. In the following the basic infrastructure underlying the concept is illustrated. Afterwards, the implementation details of the 4-Star Cooking Assistant and the Smart Home Energy Assistant are explained. Both assistants aim at multimodal interaction in smart home environments and support various aspects of UUIs. Additionally, the implementation of the Meta User Interface to provide full control over the infrastructure is explained. The initial cooking assistant example is utilized as main example and the two additional examples are described to illustrate selected aspects.

A.1. The General Concept

The development of MASP applications is strongly based on a task model, described in the CTT notation, that defines the workflow of the application, and a domain model, that defines the objects manipulated by the defined tasks and thus the internal state of the application. Each interaction task references objects defined in the domain model as well as a user interface description allowing the creation of partial user interfaces for each task. User interface descriptions are realized as velocity templates (http://velocity.apache.org/) allowing the creation of user interfaces in various output formats, based on the objects referenced in the related task. Application tasks ensure the execution of service calls to the backend logic.

Pursuing the utilization of user interface models at runtime, the developed prototype is based on the utilization of tuple spaces to store the model information. Each space stores

a single model and events between the spaces allow the synchronization of the stored objects. Based on this technology, a system of distributed components, exchanging information about the state of each component can be realized quickly. The content of the tuple spaces is initially configured via spring configuration (http://www.springsource.org/), allowing the definition of java objects via an XML description. The overall architecture is illustrated in figure A.1 and explained in the following.

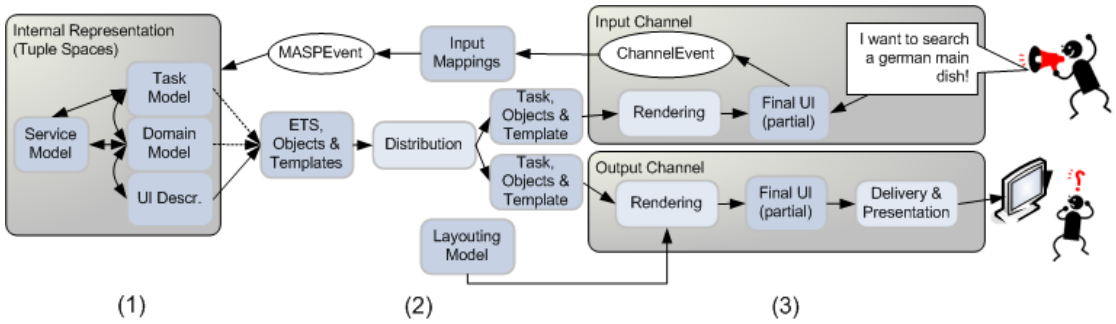

Figure A.1.: Illustration of the architecture of the first prototype of the MASP, allowing the initial realization of Ubiquitous User Interfaces for smart home environments.

Figure A.1 (1) shows the parts of the user interface model. During the user interface creation process in a first step the Enabled Task Set (ETS) is derived from the task model and the related objects and user interface descriptions are loaded. According to the set of available interaction channels the set of tasks is distributed to the channels (figure A.1 (2)). A distribution configuration that can be provided externally or by the application itself, defines which tasks are distributed to which channels. For rendering purposes, multiple templates are provided for each task, supporting the multiple types of interaction resources. The appropriate template is selected accordingly to the type of the used channel. The manipulation of the distribution by the user or the application allows to send single tasks to different channels. Finally, task, objects and the selected rendering template are pushed to the selected delivery channel (figure A.1 (3)). As soon as the channel receives the request to deliver a (partial) user interface to the connected resource, the velocity template is rendered, producing the final user interface based on the passed objects defined in the related task. The result of the rendering process is then delivered to the interaction resource, presenting the UI to the user. Interaction between system and user takes place in two ways:

1. Changes to the models are communicated as UI updates via the channels and alter the presented UIs. The annotation of the related objects in the task tree allows

the relation of objects to templates as well as the implementation of an observation mechanism for the domain model, ensuring that each UI element, that references a domain object, is updated as soon as the domain object changes.

2. User input is pre-processed via the channel, interpreting the input to create a channel event from it. This channel event is then translated to a processable MASP event via interaction mappings. Each of the channel events, received from the channel is mapped to a MASP event, with the goal to reflect the user input within the user interface model.

In the implementation, user interaction is communicated via the three different channel events (focus, selection and input) shown in table A.1.

Event	Explanation
FocusEvent	Triggered, when the user focuses an element, e.g. moves the mouse over a button.
SelectionEvent	Triggered, when the user selects an element, e.g. say a specific word.
InputEvent	Triggered, when the user enters information, e.g. writes text into a text field.

Table A.1.: Supported channel events for input received from the channels.

To provide meaning to these events from the system perspective and enable them to alter the underlying application model, channel events are translated to internal MASP events. These internal events comprise alterations of the task model (*taskDone*) or alterations of domain data (*DSWrite*) and are shown in table A.2.

Event	Explanation
DSWrite	An object in the domain storage is changed. A DSWrite event contains a query to the object, which has the form ObjectID(.FieldID)*, which specifies where to put the new value. It is possible to replace not only whole objects, but also only parts of them.
TaskDone	The user has finished performing a task, which results in marking the task as done.

Table A.2.: Supported MASP events to alter the internal representation of the interaction state in terms of task and domain model.

The Mapping of channel events to MASP events and thus alterations of the underlying

models is defined according to the interaction mappings, provided by the user interface developer. A selection of such mappings is shown in table A.3.

The transformation of user input from distinct events occurring in the UI to manipulations of the underlying mappings nicely reflects the concept of user input interpretation according to model state and given interpretation means. Any changes to the underlying models can then be reflected in the presented UI, even simultaneously across distributed interaction resources and multiple modalities. The update of the related UIs is realized via the utilization of code snippets, that are sent to the remote device and then integrated into the DOM Tree. In both cases (input and output UIs) the execution of the template creates a code snippet (a tag, in case of HTML and a <form> tag in case of VoiceXML). For HTML, using Javascript, initially deployed to the browser when the channel is established, allows adding tags to the displayed web page initially or replacing existing whenever information in the model changes. The layout is in this case defined by Cascading Style Sheets (CSS) allowing the definition of the properties of the tags, e.g. the position. The position can either be determined by the interface developer or be calculated by a layouting algorithm, which allows the dynamic positioning and resizing of the elements when distributing the UI. The layouting model, serves as basis for the calculation of the arrangement of the elements in the final presentation. It defines layouting constraints based on the defined tasks and thus allows the flexible definition of user interface layouts, that adapt to context changes and distribution. The input channel allows the processing of input by delivering Javascript code executed by the browser that creates the related events.

The described implementation aims at the creation of multimodal user interfaces, distributed across multiple interaction resources. Resources currently supported are (touch-) screen, mouse and keyboard - in this case a web browser is used as integration platform providing access to the resources via Javascript functions - as well as voice input and output - in this case Dragon Natural Speaking as well as a Loquendo and a VoiceGenie server are used - and gesture based input. Gesture based input is realized via a small device, that allows the interpretation of simple gestures detected by an accelerometer. Any interaction resource can thereby be manually registered, by calling a dedicated URL or be discovered using UPnP.

In the following, the utilization of the described platform for the development of UUIs for smart home environments is illustrated based on three examples, the 4-Star Cooking Assistant, the Smart Home Energy Assistant and the MASP Meta User Interface.

Mapping	Input	Output	Explanation
SelectionToTaskDone	SelectionEvent	TaskDone	A selection of a UI element causes a task to be marked as done.
InputToDSWrite	InputEvent	DSWrite	The user enters some input, which is then written into the domain storage.
FocusToDSWrite	FocusEvent	DSWrite	The user focuses an element (e.g. by moving the mouse pointer over it), which is then written into the domain storage (e.g. to cause a highlighting of the related element).
SelectionToDSWrite	SelectionEvent	DSWrite	The user selects a UI element and the identifier of the element is written into the domain storage. This mapping can be used to store the information which element has been selected and use it for output synchronization or highlighting.
DSModify	SelectionEvent	DSWrite	The user selects a UI element, which leads to the modification of an object from the domain storage. The initial SelectionEvent only triggers the mapping, but has no further influence on it. The mapping contains a transformer which is responsible of modifying the value of the domain object. When the mapping becomes activated, the value of the object is read from the domain storage and passed to the transformer. The transformer modifies the object and updates it.

Table A.3.: Mappings to transform channel events into MASP events.

A.2. 4-Star Cooking Assistant

The 4-Star Cooking Assistant (SCA) provides multimodal cooking instructions based on four main interaction steps. In the first step the assistant welcomes the user and introduces three options to select a recipe: a recommendation of the system, the recipe of the cooking show last seen on TV and a customized search. Selecting the first two options takes the user directly to the configuration of the number of persons, while the last option guides the user to the recipe search, allowing the provisioning of search criteria. The selection of the option as well as the input of the search criteria can happen via mouse and keyboard as well as via the touch screen or voice input. Output is provided via the screen and supported by voice hints.

After selecting the number of persons in the next step the user is taken to the list of needed ingredients according to the selected recipe. The goal of this interaction step is the creation of a shopping list based on the identification of the available ingredients. The system supports different means to identify the ingredients in this step. Available ingredients can be provided by the backend system, allowing the incorporation of different mechanisms like RFID or camera-based object recognition. However, from the perspective of this work the active provisioning of information by the user is of greater interest. Based on the list of ingredients the system now queries each ingredient, allowing the user to check the availability and then communicate either via voice or a touch on the screen if that ingredient is available or not. Additionally, the ingredients list can be moved (migrated) to a mobile device to carry it around, e.g. into the cellar.

Based on the availability of the ingredients, a shopping list is created, which can also be completed with additional items via free text input (voice or optionally keyboard). Dependent on the availability of a mobile device, belonging to the current user, the shopping list can be cloned (migrated) to this device, while still keeping interaction capabilities and being synchronized with the system. When the shopping list is migrated, the input of additional items remains in the kitchen, allowing family members to add items to the list, even if someone is shopping already. Added items are immediately synchronized with the remote presentation of the list; checked items are immediately synchronized between the two lists.

Once all ingredients are available the user can start the cooking process, based on the guidance of the system. The SCA reads out each step description and allows the user to navigate between steps using “back” and “forward” voice commands or buttons on the touch screen. Additionally, an explanatory video can be requested and controlled via

voice and devices involved in the cooking step can be controlled via the UI. Device and video control are also possible via voice, touch or mouse and keyboard. Ingredients and step details are presented visually and via voice output.

Realizing the SCA allowed the evaluation of different features of the underlying approach. An executable task model describes the basic workflow of the application. A service model interconnects services from the smart environment, specifically the kitchen appliances in this case. The combination of voice input and output with graphical output and a touchscreen allows the evaluation of multimodal interaction capabilities. Additionally, scenarios like the system driven querying of available ingredients and the migration of the shopping list to a mobile device allowed the evaluation of distribution and migration features of the system. The continuous development and enhancement of the MASP and the cooking assistant as a running demonstration, allowed to gain deep insights in the issues and downsides of the utilization of such a system at runtime. Pursuing an evolutionary approach, the implementation has been evolved and evaluated over a period of several years. The realization of the SCA scenario, based on the MASP infrastructure, is described in the following.

Underlying Models

Utilizing the Multi-Access Service Platform, the main models the cooking assistant is based on, are a task model, a domain model and a service model as well as a layouting model for the arrangement of the output user interface and the mappings from the received user inputs to these models. Additionally, a context model is utilized to integrate context information into the interaction process. Based on this set of models, the MASP is able to derive the intended user interface and control the interaction with the system. To illustrate the underlying models, the interaction with the ingredients list is analyzed, as this step comprises different interaction concepts and also involves information from all underlying models. The graphical user interface underlying the interaction is depicted in figure A.2.

The task model defines the basic workflow of the application by a set of tasks, describing the interaction in multiple steps. A screenshot of an CCT-based Eclipse plug-in showing the ingredients list sub-tree is depicted in figure A.3. In a first step the needed ingredients are calculated and the list is presented on the screen in combination with two buttons allowing to toggle if the selected ingredient is available or not. This task (*checkOff*) iteratively allows checking the available ingredients and in combination with a system task continuously updates the list. Additionally, a parallel task allows the distribution

of the shopping list to the mobile device by triggering a system task. Another parallel task provides the hook for the application backend to provide additional information about available ingredients on the fly. The whole interaction can finally be aborted by a disabling task to continue with the next step. In contrast to the CTTE Tool, the used Eclipse-based editor allows the easy annotation of interaction object to the tasks and is based on EMF/GMF, which makes it also suitable to work with executable models. The editor supports CTT XML as well as EMF/XMI as format for the task models.

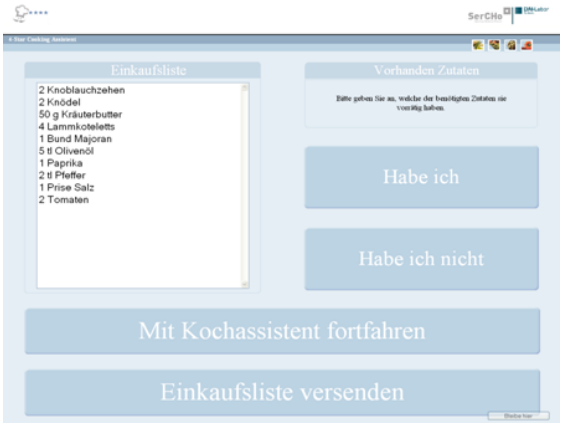

Figure A.2.: Graphical user interface of the shopping/ingredients list (in German).

At runtime, the underlying task tree interpreter works on the XML specification of the task model. The definition of the *ShowShoppingList* (shopping- and ingredients list are used synonymously in the task model) task in CTT-XML can be found in figure A.4. The XML has been slightly tidied, by removing empty and unused elements. Figure A.5 shows the same task in the XMI syntax of the EMF model. In both representations, one can see that the description tag of the definition is used to reference the related representations for the task from the configuration file of the application (line 3-7 in the CTT-XML and line 2-5 in the XMI). Additionally, the object elements are utilized to reference the task related elements from the domain model. Lines 12, 16, 20 in the CTT-XML and 8, 11, 14 in the XMI, identify the *shoppingList*, *selectedIngredients* and *newNumberofPersons* as input element, read by the task. While this definition of the task model is not optimal for runtime interpretation, it supports the comfortable editing, using the CTTE or the shown MTTE tool.

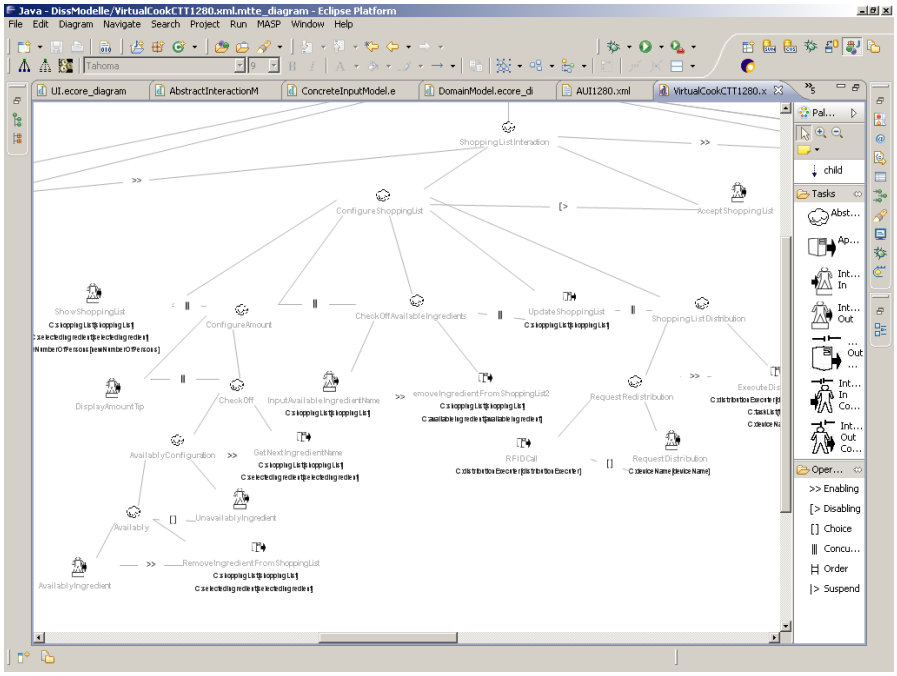

Figure A.3.: The task model of the ingredients list of the cooking assistant opened in the MTTE Editor developed at the DAI-Labor.

```
<Task Identifier="ShowShoppingList" Category="Interaction Task" Iterative="false"
  Optional="false" PartOfCooperation="false">
  <Description>
    org/sercho/masp/demos/vcook/config.xml:ShowShoppingList,
    org/sercho/masp/demos/vcook/config.xml:ShowShoppingListVoice,
    org/sercho/masp/demos/vcook/config.xml:ShowShoppingListInVoice
  </Description>
  <TemporalOperator name="Concurrent"/>
  <Parent name="ConfigureShoppingList"/>
  <SiblingRight name="ConfigureAmount"/>
  <Object name="shoppingList" class="" type="" access_mode="" cardinality="">
    <InputAction Description="shoppingList" From="null"/>
    <OutputAction Description="null" To="null"/>
  </Object>
  <Object name="selectedIngredient" class="" type="" access_mode="" cardinality="">
    <InputAction Description="selectedIngredient" From="null"/>
    <OutputAction Description="null" To="null"/>
  </Object>
  <Object name="newNumberOfPersons" class="" type="" access_mode="" cardinality="">
    <InputAction Description="newNumberOfPersons" From="null"/>
    <OutputAction Description="null" To="null"/>
  </Object>
</Task>
```

Figure A.4.: The XML representation of the ShowShoppingList interaction task in CTT XML.

```
<child xsi:type="org.dailab.sercho.masp:ChildTask" identifier="ShowShoppingList"
  description="
    org/sercho/masp/demos/vcook/config.xml:ShowShoppingList,
    org/sercho/masp/demos/vcook/config.xml:ShowShoppingListVoice,
    org/sercho/masp/demos/vcook/config.xml:ShowShoppingListInVoice"
  type="INTERACTION_OUT" operator="CONCURRENT" target="ConfigureAmount">
  <object identifier="shoppingList">
    <value type="shoppingList" accessMode="READ"/>
  </object>
  <object identifier="selectedIngredient">
    <value type="selectedIngredient" accessMode="READ"/>
  </object>
  <object identifier="newNumberOfPersons">
    <value type="newNumberOfPersons" accessMode="READ"/>
  </object>
</child>
```

Figure A.5.: The XML representation of the ShowShoppingList interaction task in EMF/XMI.

```
<div style="width:100%;height:100%" align="center">
  <div style="height:94%;width:85%;">
    <div style="height:7%;background-color:#C8D9E6;
      -moz-border-radius-topleft:10px;-moz-border-radius-topright:10px;
      color:white;font-size:1.8em;text-align:center;">
        Zutatenliste
    </div>
    <div style="background-color:#E5EEF5;border:solid 1px #C8D9E6;
      font-size:1.2em;height:93%;-moz-border-radius-bottomleft:10px;
      -moz-border-radius-bottomright:10px;">
        <div style="width:100%;height:2%;"></div>
          <select name="shoppingList" id="#auswahlingredient" size="8"
             style="height:96%;width:90%;margin-left:5%;margin-right:5%;font-size:1.0em;">
                #foreach($ingredient in $shoppingList.iterator())
                  #if($selectedIngredient.lastIndexOf($ingredient.name) > -1)
                    #if($ingredient.getAmountToBuy() > 0.0)
                        <option selected="selected" style="" value="$ingredient.name">
                          #formatAmount($ingredient.getAmountToBuy().toString())
                            $ingredient.getUnit().getName()  $ingredient.name
                        </option>
                    #else
                        <option selected="selected" style="text-decoration:line-through;"
                          value="$ingredient.name">
                            #formatAmount($ingredient.getAmountToBuy().toString())
                              $ingredient.getUnit().getName()  $ingredient.name</option>
                    #end
                  #else
                    #if($ingredient.getAmountToBuy() > 0.0)
                      <option value="$ingredient.name">
                        #formatAmount($ingredient.getAmountToBuy().toString())
                          $ingredient.getUnit().getName()   $ingredient.name
                      </option>
                    #else
                      <option value="$ingredient.name" style="text-decoration:line-through;">
                        #formatAmount($ingredient.getAmountToBuy().toString())
                          $ingredient.getUnit().getName()   $ingredient.name
                      </option>
                    #end
                  #end
                #end
          </select>
    </div>
  </div>
</div>
```

Figure A.6.: Code of the velocity template, rendering the ingredients list for an HTML channel.

Via a configuration file, each task is directly connected to velocity templates, rendering different representation of the task. The HTML-based rendering template for the ingredients list part of the screen (left of figure A.2) is shown in figure A.6. As one can see e.g. in line 14, elements from the domain storage are directly accessible for rendering purposes. However, this is only the case for those objects that have been assigned to the related task in the task model. The template of the same element for a voice channel is shown in figure A.7.

```
1 #macro(formatUnit $unit)
2   #if($unit.equalsIgnoreCase("tl"))Teelöffel#else
3     #if($unit.equalsIgnoreCase("EL"))Esslöffel#else
4     #if($unit.equalsIgnoreCase("ml"))Milliliter#else$unit#end
5     #end
6   #end
7 #end
8
9 #foreach($ingredient in $shoppingList.iterator())
10   #if($selectedIngredient.lastIndexOf($ingredient.name) > -1)
11     <taskDescription>Hast du #formatAmount($ingredient.getAmountToBuy().toString())
12       #formatUnit($ingredient.getUnit().getName()) $ingredient.name ?</taskDescription>
13   #end
14 #end
```

Figure A.7.: Code of the velocity template, rendering the ingredients list for a voice channel.

The domain model of the cooking application is defined by a set of java objects, that are instantiated at application startup with the initial values the developer provided. The objects are related to the object identifiers specified in the task list and can easily be referenced from the velocity templates. For the ingredients list example, three objects are required: *shoppingList*, holding the elements on the list, *selectedIngredient* holding the selected elements, *newNumberOfPersons*, holding the number of persons for the calculation of the amount of ingredients. At runtime these objects are stored in a tuple space, acting as domain storage, that allows the easy access to any of the stored objects.

The service model of the ingredients list part of the application comprises a service call to the ingredients manager, retrieving the list of ingredients from the application backend and the parallel call, handling the alteration of the availability of the ingredients. The calls are executed, when the application task, referencing the call (currently in the description attribute) becomes active. As the service model also specifies the related objects, parameters can be passed and return values are stored in the domain storage. Service calls are realized as java function calls to beans configured via a spring configuration. This configuration also identifies the constructor arguments and parameters to retrieve from the domain model and pass to the function. An example configuration is

shown in figure A.8. Additionally, the alteration of the distribution is handled as a service call using the model manipulation capabilities of the MASP to alter the distribution of the shopping list for the given user according to the available devices.

```
<bean id="GenerateInitialShoppingList" class="org.sercho.masp.meta.service.ServiceCallDescriptic
  <constructor-arg><ref bean="session"/></constructor-arg>
  <constructor-arg><value>GenerateInitialShoppingList</value></constructor-arg>
  <constructor-arg><ref local="JavaServiceType"/></constructor-arg>
  <constructor-arg><value>false</value></constructor-arg>
  <property name="instruction">
    <value>shoppingListFactory.create(recipes,selectedRecipeName)</value>
  </property>
  <property name="parameterIDs">
    <set>
      <value>shoppingListFactory</value>
      <value>recipes</value>
      <value>selectedRecipeName</value>
    </set>
  </property>
</bean>
```

Figure A.8.: Spring bean configuration of a service call.

During startup the context model of the MASP is connected to the available context providers of the Ambient Assisted Living Testbed, delivering context information received from the different sensors. Of most interest for the realized use cases is the location of the user within the environment. A focus has thus been set on the connection of the Ubisense system and the embedding of the information into the context model. An EMF-based context model (as presented in section 6.2) is used to provide access to context information within the MASP implementation. It defines the known users and their locations, as well as the available interaction resources and the channels, set up to each resource.

Finally, the layouting model defines constraints for the arrangement of the user interfaces. It allows the definition of restrictions for containment, size, orientation and order of the user interface elements in terms of statements. In the current implementation these statements are defined on the granularity of the task model, as the next level of detail is directly embedded into the code of the Velocity templates. Figure A.9 shows two statements about the *ShowShoppingList* task. The first statement defines that it is contained in the *ShoppingListContainer* node, the second statement defines that its size is calculated relatively to this container. Based on the defined statements, constraints will be derived and solved by a cassowary constraint solver at runtime. See Feuerstack (2008) for more details about the layouting.

```
<Layouting:LayoutingModel>
  <statements xsi:type="Layouting:ContainmentNode" active="true" priority="user">
    <characteristic xsi:type="Layouting:NodeCharacteristic">
      <node xsi:type="Layouting:LeafNode" name="ShowShoppingList"/>
    </characteristic>
    <sequence value="80"/>
    <scope model="//@predefined/@predefinedModels.1"/>
    <node xsi:type="Layouting:ContainerNode" name="ShoppingListContainer"/>
  </statements>
  <statements xsi:type="Layouting:SizeRelativNode" active="true" priority="user">
    <characteristic xsi:type="Layouting:NodeCharacteristic">
      <node xsi:type="Layouting:LeafNode" name="ShowShoppingList"/>
    </characteristic>
    <sequence/>
    <scope model="//@predefined/@predefinedModels.1"/>
    <factor enumerator="1" denominator="2"/>
    <node xsi:type="Layouting:ContainerNode" name="ShoppingListContainer"/>
  </statements>
</Layouting:LayoutingModel>
```

Figure A.9.: Statements in the layouting model, that define information about the shopping list task and its container.

While the channel events are directly generated by the channel, based on the element id of e.g. the HTML elements in the Velocity templates, the mappings of these channel events to MASP events have to be manually defined. Figure A.10 shows two mappings of the ingredients list, defined as Spring (http://www.springsource.org/) configuration. The first mapping stores the selected ingredient in the respective object in the domain model, the second mapping sets the *AcceptShoppingList* task to done if the button is selected. While the mappings are configured via Spring, they are realized as Java implementations, parametrized with the provided arguments.

```
<bean id="ShowShoppingList" class="org.sercho.masp.meta.interaction.InputToDSWriteMapping">
  <constructor-arg><ref bean="session"/></constructor-arg>
  <constructor-arg>
    <set>
      <value>ShowShoppingList</value>
    </set>
  </constructor-arg>
  <constructor-arg><value>#auswahlingredient</value></constructor-arg>
  <constructor-arg><value>selectedIngredient</value></constructor-arg>
</bean>
<bean id="SelectionToTaskDone"
  class="org.sercho.masp.meta.interaction.SelectionToTaskDoneMapping">
  <constructor-arg><ref bean="session"/></constructor-arg>
  <constructor-arg>
    <set>
      <value>AcceptShoppingList</value>
    </set>
  </constructor-arg>
</bean>
```

Figure A.10.: Mappings of the shopping list to the domain storage.

Runtime Interpretation

The runtime interpretation of the described building blocks follows the general concept that has been described in A.1. However, there are some interesting additional aspects within this application.

The general workflow of the application has been defined by the task model. Semantics have been added to each task by referencing a service call for each application task and Velocity rendering templates for each interaction task. Domain information is provided by Java objects within the domain storage. Based on this basic structure, the application is split into building blocks on a per task-basis. These building blocks can be independently distributed to different or even multiple interaction resources where they are rendered and arranged by the layouting constraints. This allows the independent addressing of multiple modalities and the synchronization of user interfaces across devices and modalities. Thus the basic mechanism for the creation of distributed user interfaces is provided by the channels and heir utilization at runtime. While each task can be distributed independently and its input and output can be separated, a flexible combination of interaction resources is possible, which can even be calculated at runtime. The distribution of the shopping list can be seen in figure 2.4 in section 3.1.3.

A major feature that this implementation provides is the configuration of the own distribution by an application at runtime. Within the SCA this has been utilized to move the ingredients list to a mobile device, e.g. to collect ingredients from the cellar, as well as to realize the cloning of the shopping list to a mobile device, while keeping a completely synchronized copy of the application in the kitchen. Underlying these features is the capability of the distribution, depicted in figure A.1 (2) to assign tasks to channels for presentation. While the distribution component is capable to calculate different distributions, is can also be configured from the outside. For this purpose, it provides an API that can be used for services to configure the MASP distribution. In case of the cooking assistant, this feature has been used to alter the distribution by a service call, when the user requests to do so. The same MASP API also makes all storages and the context model accessible to external services and thus allows the influencing of the UI presentation and handling in various ways.

Another feature that has been realized as part of the SCA is the possibility to adapt the layout of the user interface according to context information. Based on the layouting model, a relation between the distance of the user and the current screen is defined by conditions as depicted in figure A.11, influencing the presentation of the user interface. In particular, the relation prioritizes the statements according to their relation to each

type of interaction elements. Thus, the space taken by input elements is lower prioritized than the space taken by presentation elements, if the user moves further away from the display. Figure 2.3 in section 2.3.2 illustrates the effect of the adaptation of the shape of the ingredients list.

```
<predefinedSituations name="distance">
  <properties xsi:type="Layouting:EnviromentCondition">
    <function>
      <variables name="Distance">
        <assigned value="100.0"/>
      </variables>
      <preconditions
        variable="//@predefined/@predefinedSituations.0/@properties.0/@function/@variables.0">
        <value value="100.0"/>
      </preconditions>
    </function>
  </properties>
</predefinedSituations>
```

Figure A.11.: Definition of a situation as part of the layouting statement definition.

Using the System

The developed cooking assistant has been installed as continuous demonstration in the Ambient Assisted Living Testbed. In this environment it is continuously presented to visitors from economy and academics. The system has additionally been presented on various fairs and conferences within a mobile setup. However, the feedback gained from visitors was manifold. While many people liked the ideas and concepts underlying the approach, having a talking cooking assistant actually at home in the kitchen polarizes people. This has also been shown by a small study within the SerCHo project, where the different applications have been presented and people have been asked for their feedback.

The applications have been presented to about 20 users between 18 and 60 with different educational backgrounds. While the study had a focus on the acceptance of the overall system, additional questions about the multimodal aspects of the cooking assistant and the features of Ubiquitous User Interfaces have been asked. The users were asked to on the one hand interact with the CA as well as rate it and develop ideas for improvements. After briefly introducing the cooking assistant, the users were asked to find the lamb chops and check the first three steps of the cooking process. The users were guided and supported by the interviewer; voice interaction has been realized through a Wizard-of-Oz setup, where the system was manually controlled. While it was very unclear, if users would like the idea of multimodal applications within their home environment and especially within their kitchen, it turned out, that this topic really polarizes users. While more than 2/3 of the users were positive about the idea to use multiple modalities to control the cooking

assistant and the possibility to seamlessly switch between the used modalities (touch and speech) about half of the users would not want such an application in their kitchen. The availability of the touch screen as additional input modality provided an idea of safety against failures of the voice recognition and the ability to control the application redundantly with multiple modalities was mentioned positive.

A.3. Smart Home Energy Assistant

The Smart Home Energy Assistant has been developed, based on the same implementation of the underlying MASP infrastructure as the 4-Star Cooking Assistant. It provides similar features in terms of shaping, multimodal usage, distribution and migration. However, the actual goal of the assistant is different from that of the cooking assistant. While the SCA has a continuous step by step interaction, the SHEA requires jumping around between dialogs. It initially provides an overview of the available rooms and selecting a room leads to a view of the devices within the room. Selecting a device then leads to an overview of the energy consumption of the device in form of a graph and the possibility to control the device. From the energy graph, the user can jump back to the device list or the room overview. The room overview and the energy consumption graph are shown in figure A.12.

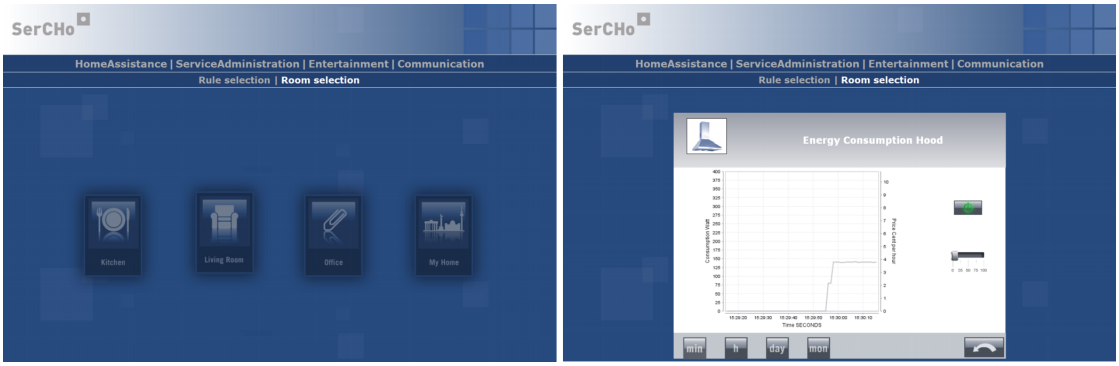

Figure A.12.: Screenshots of the Smart Home Energy Assistant.

In contrast to the cooking assistant the user is always able to jump back and forth between dialog steps, which requires more navigation capabilities and poses a greater challenge on the task tree than the step by step process of the cooking assistant. Additionally, the assistant emphasizes the ability of the user interface to follow the user throughout the rooms and illustrates the possibility to address elements via voice, that are not

currently shown on the screen. The utilization of the application via a remote control also posed additional requirements to the user interface that had to be considered during the design and development. The follow-me functionality is based on the distribution and migration functionality, that has been used for the shopping- and ingredients list. It is realized as a combination of the location information provided by the context model and the capability of the distribution component to automatically calculate the combination of selected channels. In dependence of the position of the user and its distance to the used interaction resources, the MASP calculates a new set of interaction resources, that might be more suitable if the follow-me mode is activated. This leads to the continuous availability of the user interface on the interaction resources closest to the user.

This basic functionality required for the follow-me of a user interface has been realized as part of the distribution component. It consists of the possibility to register observers to the context model, that notify the distribution component about any changes to the user location or the available interaction resources (appearing and disappearing). This is realized using tuple space observers in case of any information stored in tuple spaces. However, as the context model is stored as EMF model, the EMF capabilities are used. Whenever interaction resources appear and disappear or the user changes his location, the distribution component is notified. This in turn leads to a recalculation of the used interaction channels, based on the new information from the context model. As a result, the elements of the presented user interface are removed from the channels of the resources, that are no longer suitable. This is done via the update mechanism, allowing e.g. the removal of any of the -tags of the presented HTML-based graphical user interface. Additionally, user interface elements are added to the channels of the newly assigned interaction resources via the same mechanism. A basic need for this interaction is thus the continuous connection of the interaction resources and the unrestricted manipulability of the user interface at any time.

However, important factors that rose in combination with the developed applications are user control and configurability of the provided features. While features like dynamic distribution at runtime and follow-me, multimodality or the adaptation of the user interface can be powerful features there is a strong need to provide user control. Although, the features can be either triggered by the application to support application functionality, they are also provided as application spanning features of the underlying infrastructure, which requires to provide an application spanning mechanism to control them. These observations are the basis for the third application, presented in the next section: the Meta-UI.

A.4. Meta-UI

A third application that was realized via the Multi-Access Service Platform is the Meta-UI, providing direct control to the MASP features for the user. It allows the management of multiple assistants and supports switching between them. Additionally, it allows the user to alter the distribution of the user interface by configuring interaction devices, to change the utilization of different modalities, to move assistants back and forth between specific devices and to control adaptation capabilities of the MASP like distance-based layouting or follow-me. The Meta-UI is also the initial MASP application, loaded at startup and provides user management capabilities.

Initialization

During the initialization phase, the MASP in the first step connects all available interaction resources through the interaction channels. Each interaction channel delivers an initial user interface to the interaction resource signaling the connection to the MASP on output IRs and monitoring any initial input on input IRs. One initial problem is the separation of input and output in this context, as initially input and output resources are separated and not related until associated to a user. This leads to the a setup where the event of the user hitting a button on a keyboard or a remote control triggers an interaction request, but initially lacks the relation to a screen, which makes it difficult to start some interaction. Thus, on the initial input event a distribution configuration, combining input and output resources, has to be calculated based on the information about the IRs, e.g. which IRs are combined in the same interaction device or which IRs are close to the user. This can be calculated based on the context model. While this concept is understandable for a remote control, that can be used to control multiple screens, it becomes a bit awkward for a touch screen. However, once the initial distribution configuration has been calculated the user is either provided with a login screen if he could not be identified by the Meta-UI application or with his personal view of the Meta-UI allowing him to control the MASP features and use additional MASP applications.

Meta-UI Features

The Meta-UI allows to control the features of UUIs. The main goal of the Meta-UI can be described as the provisioning of control about the configuration of the personal interactive space of a user. An ambient interactive space has been defined as *a dynamic assembly*

of physical entities coupled with computational and communicational entities to support human activities (Coutaz, 2006). According to this definition the personal interactive space can be defined as *the set of currently used services and interaction resources as well as the connections between them* as illustrated in figure A.13.

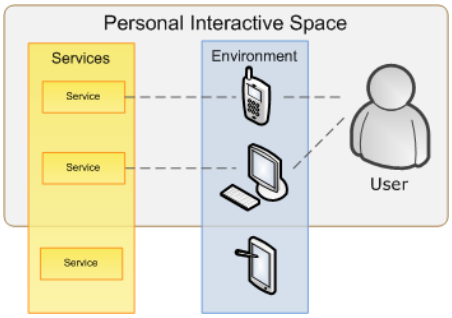

Figure A.13.: The personal interactive space of a user, comprising the used devices, services and the connections between them.

As illustrated in figure A.14, the user uses the interaction resources available in the environment to interact with the provided UUIs and accesses backend services through them. In the same way, the Meta-UI is provided as a UUI, allowing to access configuration services provided by the MASP API. These configuration services in turn alter the connections between UUIs and interaction resources and thus the distribution of the user interface, via the API provided by the distribution component.

Utilizing this configuration of the distribution, the Meta-UI can alter the input and output devices currently used by an application (user interface) and thus provide full control about this configuration to the user. Features that have been integrated into the Meta-UI in a similar manner are:

1. the configuration of modalities, which can be mapped to the alteration of the distribution with respect to the type of devices,
2. the migration of a UI between interaction resources as described above,
3. as well as the additional configuration of features like follow-me or distance-based layouting by additional services and functionalities provided by the MASP API.
4. the (re-)distribution of parts of a UI to different interaction resources, which has been realized on a task-basis in the described implementation, meaning that each task can be independently assigned to an interaction resource,

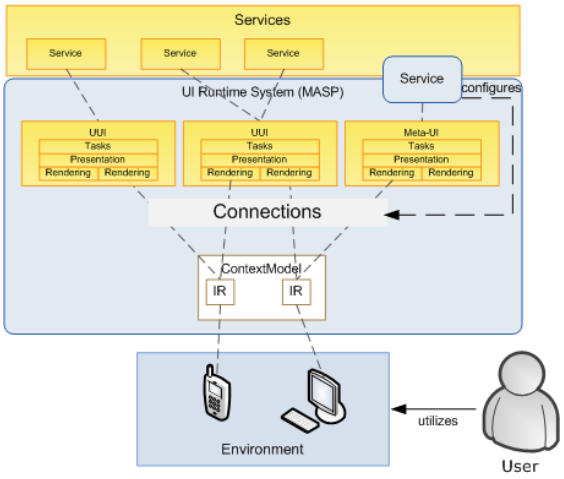

Figure A.14.: Utilization of the Meta-UI. The user uses device in the environment to access the Meta-UI (and other services). The Meta-UI is then used to configure the utilization of the interaction resources by altering the distribution, connecting a UI to an interaction resource.

A screenshot of the Meta-UI illustrating the four different configuration possibilities (from left to right) is shown in figure A.15. The screenshot shows the capability to start and visualize a service in the top left corner. The symbols at the top-middle visualize the currently active modalities. At the bottom, the four configuration possibilities are visualized. Additionally, the Meta-UI is controllable via voice. The command “Open Meta-UI” shrinks the current application and opens the Meta-UI around it. Afterwards, additional commands like e.g. “Migration to kitchen display” can be used to control the provided features.

Technically, the different configuration scenarios described above can be brought down to the atomic operations of creating a connection between a task representation and an IR or removing such connections. For example changing the interaction modality of a task from graphical to vocal includes the removal of connections between IRs and graphical UI elements of that task and the creation of new connection between the voice elements and the appropriate IR (or IRs). When a "UI to IR" connection is established the MASP runtime system sends the element to the IR, which then creates the final user interface and delivers it to the IR. If a UI element should no longer be accessible through an IR, the appropriate connection between both is destroyed, which results in the removal of the corresponding UI element (e.g. the -tag) from the IR. Again, the association between the presented UI elements and the elements at higher levels of abstraction (task and domain) are always preserved, which is necessary for the state

synchronization of all elements. For example, if a task becomes no longer available to the user, the associations assure that all connections between the UI elements belonging to the task and the interaction resources are removed. As the result the user cannot access the user interface of the task and has no possibility to perform it.

Figure A.15.: Screenshot of the Meta-UI with the different configuration features. From left to right: Modality allows to configure the used modalities (voice out, voice in, graphical out), Migration allows to migrate complete applications to the current device or redirect their output, Adaptation allows to enable and disable distance-based layouting and follow-me adaptation capabilities, Distribution allows to configure which task should be presented on which interaction resource.

This utilization of the Meta User Interface to control the MASP features shows the possibility to on the one hand completely integrate and utilize the provided features from within an application via service calls to the MASP API. On the other hand, it also illustrates the possibility to provide application spanning configuration features to personalize the personal interactive space and the presentation and interaction means of any Ubiquitous User Interface.

A.5. Summary

In this section the implementation details of a case study to evaluate the idea of Ubiquitous User Interfaces and the required infrastructure features have been presented. An important aspect of the approach was to veil the boundaries of each device and let the system appear as one complex system. This is technically supported by the network connections between the different systems, and has been exploited by the created distributed

user interface. The independent addressing of input and output and the free configuration of used modalities and resources provides very flexible and dynamic interaction means for smart environments.

Based on this infrastructure, different example smart home applications have been created, deployed, and evaluated in a smart home laboratory. The presented cooking assistant puts a focus on the utilization of multiple modalities for interaction and the evaluation of application dependent migration and distribution features. The energy assistant incorporates a remote control as IR and puts a focus on the utilization of the follow-me feature to provide seamless access to the application if requested. Finally, the Meta-UI provides strong customization possibilities and full control over the UUI features for the user. Additionally, it showed the recursion of the approach, by allowing to implement the Meta-UI as meta user interface, using the same features that are provided for any MASP application.

Based on the results of these implementations, a second case study has been conducted, with a stronger focus on the modeling aspects of the approach. Details about this study are described in the next section.

B. Case Study: Executable UI Models

Based on the results of the implementation of the infrastructure concepts, a second case study has been conducted with a stronger focus on the full utilization of the developed set of models and components. Main aspects were the evaluation of the model execution capabilities, the interconnection of the models via mappings, as well as the bootstrapping and management of the models at runtime. Thus, the described metamodels have been implemented and selected features of the 4-Star Cooking Assistant have been reimplemented/remodeled. Additional aspects that have been investigated but are outside of the scope of this thesis are the development of the models at design-time using Eclipse and Eclipse Modeling Framework (EMF) / Graphical Modeling Framework (GMF) tools, and the utilization of these tools to monitor and control the runtime state of the system, which lead to the realization of interesting approaches and tools.

In the following, the utilization of the executable models and their realization in EMF is discussed on the example of the 4-Star Cooking Assistant. EMF has been chosen as modeling notation, as it is a modeling and code generation framework integrated into the Eclipse platform, which is widely used and provides comprehensive tool support. EMF integrates three main components:

- The core EMF framework, which provides the ECore metamodel for describing models. It also includes runtime support for the models (notification, persistence, and a reflective API).
- The EMF.Edit framework, which includes generic classes for building editors for EMF models.
- The EMF code generation facility, which is capable of generating code from a developed EMF (meta-) model.

In the following the utilization of the ECore metamodel, its runtime support and the code generation facilities are of most interest for the creation of executable models. ECore is probably the most used model and metamodel exchange format and has been defined as a subset of MOF, making it compliant to OMGs MDA approach. The elements of ECore are shown in figure B.1.

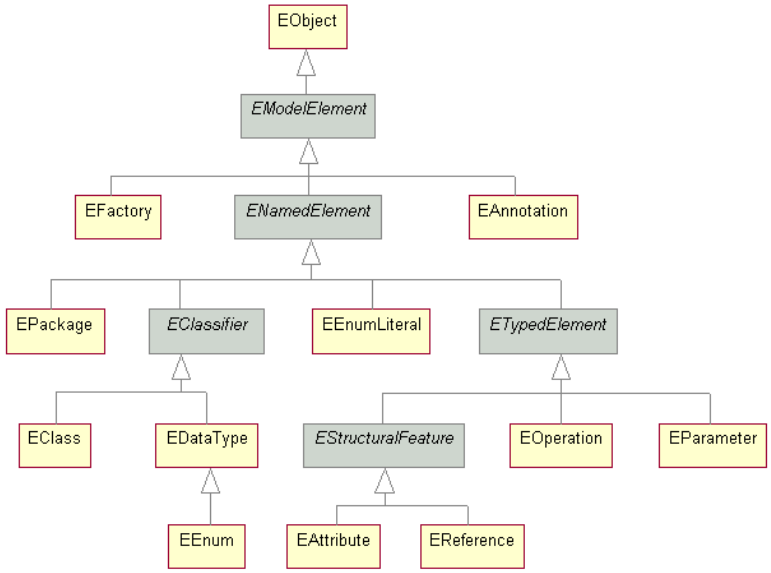

Figure B.1.: The main elements of the ECore metamodel.
From http://www.eclipse.org/modeling/emf/ (last visited February 12th, 2009).

The root entity of this metamodel is the *EObject*, of which all other elements are subtypes. Similarly to Java, ECore elements are organized in *EPackages*, storing *EClassifiers*, which are *EClasses* and *EDataTypes*. Similarly to object oriented languages, *EClasses* are defined by *EOperations* and *EAttributes*. Other classes can be referenced by the *EReference* element, where aggregations and compositions are supported. Both the *EAttribute* and the *EReference* are *EStructuralFeatures* contained in the *EClass* elements and resemble the notion of class fields from the Java programing language. Defining metamodels can be done with the graphical EMF editor shown in figure B.2. Models can be edited with the build-in tree-based editor shown in figure B.4. Alternatively, own editors can be developed for comfortable and metamodel specific editing. Metamodels as well as models are stored in the XML-based XMI format and can thus be easily exchanged. An example XMI definition of the task metamodel is shown in figure B.3.

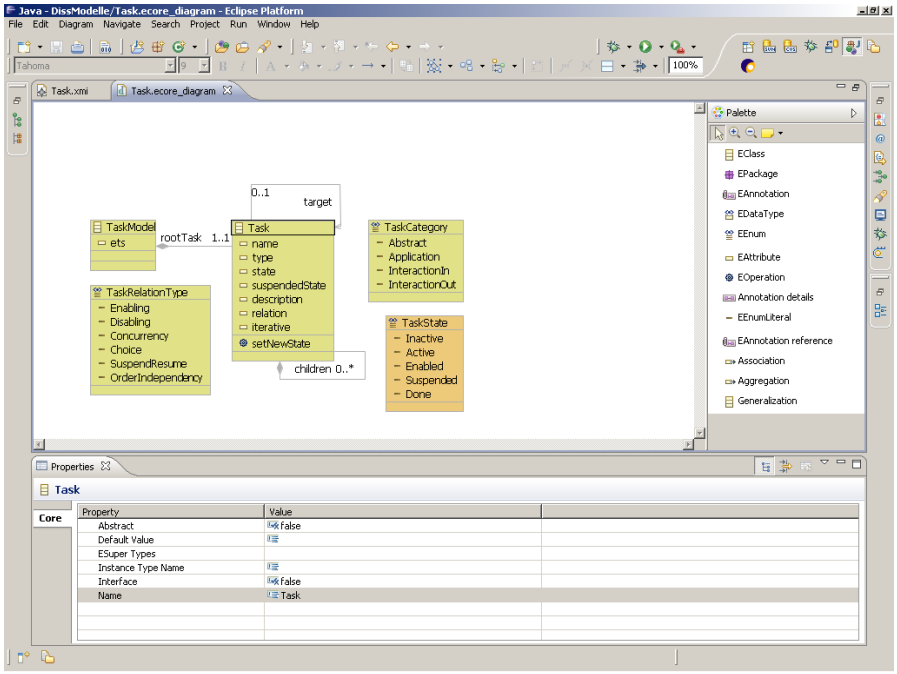

Figure B.2.: Screenshot of the graphical editor for EMF metamodel.

Another important aspect for the utilization of ECore for executable models is its direct mapping to Java code. This allows the generation of Java classes from metamodels which can also be enhanced with custom code. Any creation of a model then relies on these derived classes. In case of Eclipse this means, that the internal editor not only shows the tree-view of the model, but also holds the related instances of the generated classes in its memory. This allows the direct execution of models from within the Eclipse platform and even their alteration at runtime. The EMF technology thus provides the foundations for the following description of the implementation of executable models.

The realization of the metamodels with EMF, strongly follows the descriptions of the metamodels provided in chapter 5 and 6. As the models have been extensively discussed there, only some additional implementation related aspects are illustrated in the following, before the SCA example is described.

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore" name="TaskModel"
    nsURI="http://www.dai-labor.de/~lehmann/Task.ecore" nsPrefix="TaskModel">
  <eClassifiers xsi:type="ecore:EClass" name="Task">
    <eOperations name="setNewState" lowerBound="1">
      <eParameters name="newState" lowerBound="1" eType="#//TaskState"/>
    </eOperations>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"
        iD="true"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="type" lowerBound="1"
        eType="#//TaskCategory"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="state" lowerBound="1"
        eType="#//TaskState">
      <eAnnotations source="SituationElement"/>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="suspendedState" eType="#//TaskState">
      <eAnnotations source="SituationElement"/>
    </eStructuralFeatures>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="description"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EString"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="children" upperBound="-1"
        eType="#//Task" containment="true" eOpposite="#//Task/parent"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="relation" lowerBound="1"
        eType="#//TaskRelationType"/>
    <eStructuralFeatures xsi:type="ecore:EAttribute" name="iterative" lowerBound="1"
        eType="ecore:EDataType http://www.eclipse.org/emf/2002/Ecore#//EBoolean"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="parent" eType="#//Task"
        eOpposite="#//Task/children"/>
    <eStructuralFeatures xsi:type="ecore:EReference" name="target" eType="#//Task"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EEnum" name="TaskState">
    <eLiterals name="Inactive"/>
    <eLiterals name="Active" value="1"/>
    <eLiterals name="Enabled" value="2"/>
```

Figure B.3.: Example XMI-code of the task metamodel.

B.1. The Executable Task Model

The basic structure of the executable task metamodel, as described in section 5.1, provides the foundations for the realization of the model within EMF. Based on the defined metamodel, Eclipse allows the creation of various task models via the internal editor as shown in figure B.4 or the specialized task editor that has been shown in figure A.3. To be able to utilize the CTT-based task model the static part of the CTT meta-model has been extended with the required dynamic state information. During execution - starting with the root task - the *setNewState* operation is used to change the state of the task as well as all related child-tasks (according to their temporal relations). The execution state of the model is then stored as part of the model to derive the Enabled Task Set (ETS) containing all enabled tasks. Once a task is completed it is set to state *done* and removed from the ETS. To realize the described execution functionality of the task

model, Java classes have been derived from the metamodel and adapted to support the runtime execution. They initially implement getter and setter methods for each element of the model. Figure B.5 shows the interface of the implementation of the task element.

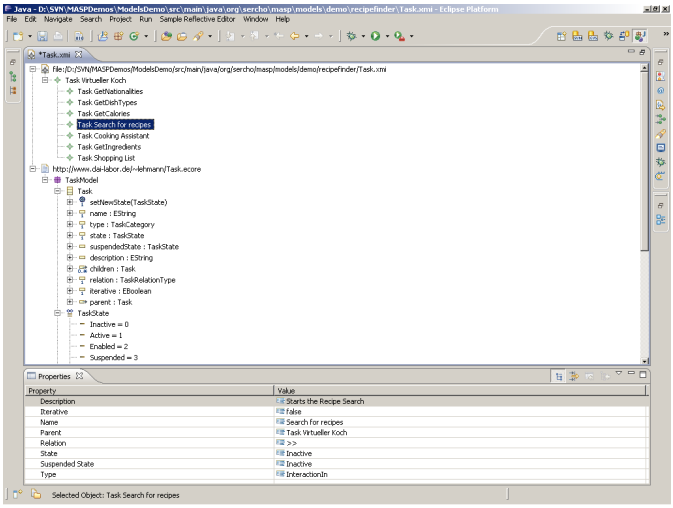

Figure B.4.: Screenshot of the Eclipse ECore editor, showing an excerpt of the executable task model of the cooking assistant and the properties of the "SearchForRecipes" task..

```
1  package org.sercho.masp.models.TaskModel;
2  import org.eclipse.emf.common.util.EList;
3  import org.eclipse.emf.ecore.EObject;
4
5  public interface Task extends EObject {
6      TaskCategory getType();
7      void setType(TaskCategory value);
8      TaskState getState();
9      void setState(TaskState value);
10     TaskState getSuspendedState();
11     void setSuspendedState(TaskState value);
12     String getName();
13     void setName(String value);
14     String getDescription();
15     void setDescription(String value);
16     EList<Task> getChildren();
17     TaskRelationType getRelation();
18     void setRelation(TaskRelationType value);
19     boolean isIterative();
20     void setIterative(boolean value);
21     Task getParent();
22     void setParent(Task value);
23     Task getTarget();
24     void setTarget(Task value);
25     void setNewState(TaskState newState);
26 }
```

Figure B.5.: Java interface of the implementation derived from the task element of the task metamodel.

During the code generation EMF implements interfaces and standard implementations for different ECore element. Some mappings between ECore elements and Java classes are shown in table B.1. The defined interfaces realize an API, which allows to access models conforming to the metamodel.

EMF	Java
EClasses	Java interface with getter and setter methods for each EStructuralFeature of the EClass
EStructuralFeature	getter and setter methods
EOperations	Java methods stubs, that have to be filled by the developer
EEnums	Java enumerations
EDataTypes	no code is generated as they encapsulate already existing Java types
EPackage	Java package plus a special interface, which contains information needed for metamodel reflection

Table B.1.: Selection of the most relevant mappings from EMF elements to Java code.

Based on these interfaces, the implementation of the elements can be adapted to reflect the behavior expected from the metamodel. This is especially required for the execution elements identified in the metamodel, which realize the execution logic of the final model. The mapping between the elements of the meta-metamodel to EMF elements is listed in table B.2.

Additionally, the generated code allows the registration of adapters to each of the model elements, which support the exchange of notifications about any changes to any of the model elements the adapter has been registered for. Every received notification contains the information about the *EStructuralFeature*, which has been changed, as well as its new and previous values. Additionally, a notification informs whether an element has been added, removed or updated. This feature is especially used by the mapping model and to interconnect multiple models.

Meta-Metamodel Element	ECore Element	Description
Definition Elements	EClasses	The Definition Elements are represented by EClasses in ECore.
Situation Elements	EStructuralFeatures	The situation elements find their representation in the ECore's EStructuralFeatures although not all EStructuralFeatures are situation elements as some attributes of an element (EAttribute) may describe runtime state data. The differentiation is therefore done by the adoption of an extra EAnnotation.
Executable Elements	EOperations	The executable elements are in ECore expressed as EOperations, which allows adding execution logic into a metamodel in form of Java code fragments. In Java the execution logic is defined within methods and these are represented by EOperations within ECore.

Table B.2.: Mapping of meta-metamodel elements to ECore elements.

B.2. Other Models

In a similar way as the described task model, the additional metamodels have been implemented. This comprises the domain-, service-, interaction-, layouting- and context-model. Additionally, a basic implementation of the adaptation model has been realized. Some insights and selected details about the implementations will be explained in the following. It has to be noted here, that the distribution- and fusion model have not yet been implemented as EMF models, as they have been implemented as Java components using proprietary configuration and storage technologies (e.g. tuple spaces in case of the distribution).

The realized domain model allows a straightforward realization within EMF, a main aspect here was the modification of the *getClass* method of the Class element to actually refer to the classes defined in the domain model. More interestingly, the implementation of the service model required the implementation of facilities to perform calls to Java

methods as well as to web services. Thus the facilities to generically perform such calls have been implemented as part of the classes underlying the metamodel. Main portions here have been addressed via Java reflection methods.

The realization of the interaction model, however, required an immense additional programming effort. Here, the interaction logic of each of the interactors had to be implemented on the server side. This reflects the fact, that the interaction is completely handled and synchronized via the server-side elements. Thus, each of these elements is required to handle incoming user interaction or state updates from the system and to adapt its internal state accordingly. Additionally, each adaptation of the internal state also has to be conveyed to the user by an update message through the related channel. An example would e.g. be the user, selecting a list element. This would result in a *selected* event, passed to the concrete input interactor, which is again related to e.g. the *selectElement* method of the concrete graphical list interactor. Here the internal logic of the interactor has to identify the selected element and ensure, that it is marked as highlighted, which is then again reflected in the current presentation.

The most interesting aspect of the context model is probably the realization of the connection between the model and external components outside of the model. For this purpose an *EClass* is used, to define the proxy element for external process. The *EClass* provides fields for the class name of the Java class implementing the external process, configuration information for the external process, the start and stop method of the process, and a reference to the instantiated external process at runtime. Additionally, callback elements identify *EOperations*, that can be used by the process to push information back into the model. This proxy, allows managing the process at run time so it does not run out of control and can be started or stopped when necessary.

One of the proxies that has been implemented for the context model, connects the localization system to the position information of the *ElementWithPosition*. This proxy references a Java class, that connects to a server providing the location information of the Ubisense localization system. The reference object is initialized during the initialization phase of the context model and connects to the Ubisense server at startup. Once started, the currently relevant tags can be registered for the process. Afterwards, the proxy uses the *setPosition* executable element as callback method to push positioning updates into the model. Altering the position of an element (which changes the position situation element) can then trigger any mapping to adjust the corresponding models accordingly.

B.3. Mappings

To connect the different executable models that have been implemented, the additional mapping metamodel has also been implemented using EMF, to facilitate the exchange of information between the models at runtime. The EMF implementation of the mappings reflects the metamodel illustrated in figure 4.2 in section 4.3 and conforms to the described meta-metamodel of the executable models. The main principle behind the realization of the mapping model with EMF is the ability of EMF to include and reference a model within another model. This feature allows to create standard mappings as types, that refer to the metamodels of the system. Once a UI developer creates models according to these metamodels, the pre-defined mappings can directly be used to relate dedicated model elements and thus easily provide the necessary information exchange.

The implementation of the mapping metamodel is derived from the mapping of the meta-metamodel with the ECore meta-metamodel. This way it is possible to define mapping types on top of any executable ECore metamodel (M2) used within the approach. The mappings use the mapping types to connect M1 entities and thus reference EObjects. The mapping type of a mapping defines what links it contains, whereas each link may be triggered by a different situation element. The implementation utilizes the build in eventing mechanisms provided by EMF in the generated Java objects derived from the models. Using these, the mapping model is able to register adapters to every EObject acting as source for a mapping. These adapters become notified about any occurrence within the model element. Every received notification contains the information about the EStructuralFeature (situation element), which has undergone a change and its new and previous values. A transformation language allows the definition of additional transformation logic, which can be used to mediate between the concepts of different models. After a link has been triggered and the transformation produced new data for the target model the Java method denoted by the EOperation of the execution element is invoked. As described in section 4.3, the mapping metamodel distinguishes mapping types, defined between metamodels, and the actual mappings, defined between the actual models. Figure B.6 shows a mapping type, that has been defined between the task model (line 2-8) and the abstract user interface (line 9-11). The defined mapping reacts on the alteration of the task state to Enabled or Active (line 13-18), which results in the activation of the abstract interactor (line 19-22). Additionally, setting a task to Inactive, Disabled or Done (line 25-31), results in the deactivation of the interactor (32-35).

```
<mappingType name="InteractionIn2Choice">
  <source>
    <constraints xsi:type="Mapping:AttributeValueConstraint">
      <eAttribute href="http://masp.dai-labor.de/~mappings/Task.ecore#//Task/type"/>
      <expectedValues>InteractionIn</expectedValues>
    </constraints>
    <eClass href="http://masp.dai-labor.de/~mappings/Task.ecore#//Task"/>
  </source>
  <target>
    <eClass href="http://masp.dai-labor.de/~mappings/AbstrIteraction.ecore#//Choice"/>
  </target>
  <sourceToTarget>
    <trigger>
      <expectedValues>Enabled</expectedValues>
      <expectedValues>Active</expectedValues>
      <eAttribute xsi:type="ecore:EAttribute"
      href="http://masp.dai-labor.de/~mappings/Task.ecore#//Task/state"/>
    </trigger>
    <target>
      <eOperation
      href="http://masp.dai-labor.de/~mappings/AbstrIteraction.ecore#//Interactor/activate"/>
    </target>
  </sourceToTarget>
  <sourceToTarget>
    <trigger>
      <expectedValues>Inactive</expectedValues>
      <expectedValues>Disabled</expectedValues>
      <expectedValues>Done</expectedValues>
      <eAttribute xsi:type="ecore:EAttribute"
      href="http://masp.dai-labor.de/~mappings/Task.ecore#//Task/state"/>
    </trigger>
    <target>
      <eOperation
      href="http://masp.dai-labor.de/~mappings/AbstrIteraction.ecore#//Interactor/deactivate"/>
    </target>
  </sourceToTarget>
</mappingType>
```

Figure B.6.: Mapping type, mapping interaction tasks to abstract choice objects.

Defining the mapping to relate a specific task to a specific abstract interactor, requires the definition of a mapping between the two distinct elements as shown in figure B.7. Type, as well as source and target elements of the mapping are identified via XPath expressions, identifying the absolute path to the model elements.

```
<mapping>
  <type href="MappingTypes.xmi#InteractionIn2Choice"/>
  <source href="Task.xmi#Search for recipes"/>
  <target href="AUIModel.xmi#//@abstractInteractors.2"/>
</mapping>
```

Figure B.7.: A mapping defined between a distinct interaction tasks and an abstract choice element.

Mapping the definition of mapping types and mappings to Java implementations allows the relation of the defined model elements and facilitates the exchange of information between the elements at runtime. An overview of the relation of the models for the

described recipe finder example is sketched in figure B.8.

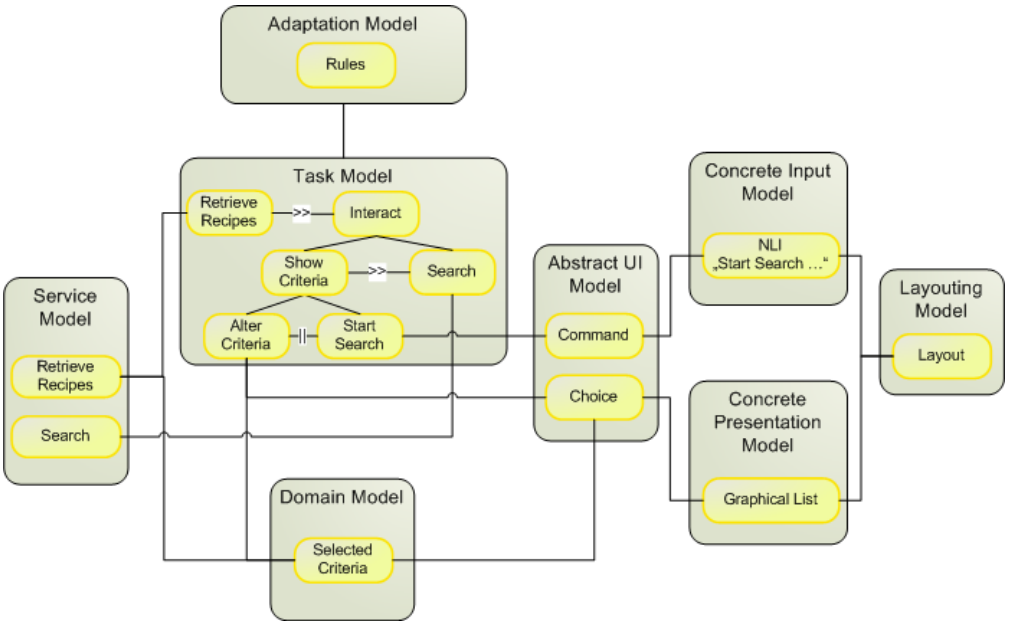

Figure B.8.: Illustration of the relation of the different models via the mappings.

B.4. Bootstrapping

One interesting issue when expressing a user interface as a set of executable models is the bootstrapping mechanism, starting up the system. To address this, a core model has been defined for each application. This model lists all models and components required to execute the models. The metamodel of the core model is shown in figure B.9. As one can see, the MASP itself is defined as a set of metamodels, combined with models, that belong to an application as well as components and session. While an application is expressed in terms of multiple models and the required components, a session identifies the user, using the application. During bootstrapping, the known metamodels, applications and components are initially loaded. This also leads to the possibility to register interaction devices manually or automatically. At this stage now, a user can log into the system, by “taking over” one of the interaction resources and requesting an application. This in turn leads to the creation of a session for the given user and application and initializes the application state for the given user. At this point the application is rendered on the used interaction resources and ready to be used.

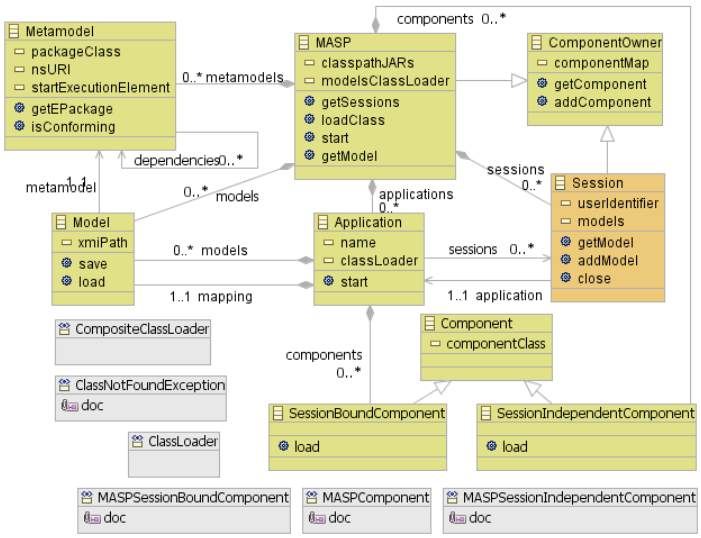

Figure B.9.: The metamodel of the MASP core model, ensuring the bootstrapping by referencing the required models and components.

B.5. Resulting User Interface

Executing the defined user interface models within the MASP runtime environment results e.g. in the graphical representation of the recipe finder of the cooking assistant that has been shown in figure 7.4 in section 7.3. Additionally, voice and HTML support have been developed, to produce multimodal user interfaces, similar to what has been presented in section A. However, the possibilities to actually style the user interface to achieve similar results as in the first case study are currently still limited. The main reasons for this are the limitations of the rather rough definition of the interaction models. To achieve similar results as e.g. with current HTML/Javascript-based user interfaces, requires an extended definition of user interface design characteristics within the models. The developed approach allows the additional definition of style information for specific channels, that can override the automatically created style in selected cases.

Based on the recipe finder, the model adaptation has been evaluated as approach to adapt the user interface dynamically. To realize this, the adaptation model defines different variants of the task model. The search capability of the recipe finder gives the user the possibility to input several criteria and start a database search for recipes. Criteria include three different options: dish type, calories, and nationality of the recipe; each with several possibilities, designed as multiple-choice selection. Figure B.10 (a) shows the

graphical user interface of the recipe finder when sufficient display size is available and the corresponding task tree. Active tasks are marked green and the task currently performed by the user (recognized because the user focuses the associated GUI elements with the mouse) is marked blue. Assuming that the user switches to a different interaction device with a reduced screen size it might become impossible to render all recipe finder elements at once, which requires an adaptation of the user interface. A possible adaptation includes rendering only one recipe criteria container at once and letting the user select the criteria he wishes to input. Although, this mainly influences the presentation, it can be done with a modification of the task tree by putting concurrent tasks in a sequence. Figure B.10 (b) shows the recipe finder after such an adaptation. Now the user may only set one type of criteria at once and navigate between the criteria types using the Next>> button. Ideally the adaptation algorithm takes the current state of the application into account and does not hide the currently focused task (*DishTypeSelection*) from the user so his interaction is not interrupted.

The code of the adaptation model for this example is shown in figure B.11. As one can see, multiple adaptation steps are defined. The first step (line 2-6), e.g. queries the focused children from the task to adapt by applying the root query and the identified subquery (line 18-23) and stores the result in a variable. In the additional steps, the operators are rebuild and a new interaction task, the related mapping and the abstract interaction elements are added by calling the identified executable elements.

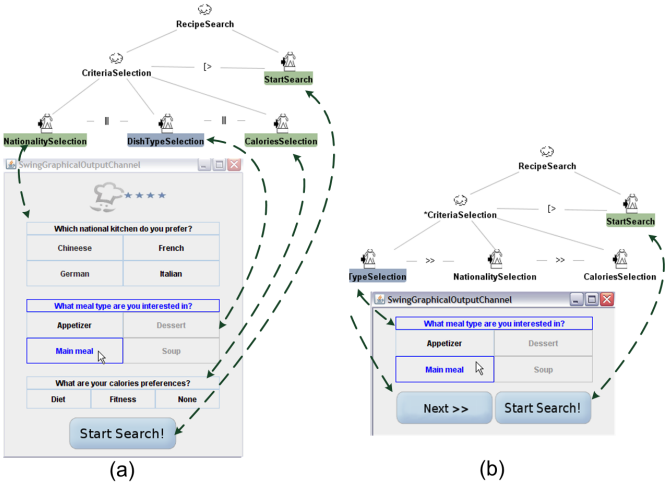

Figure B.10.: Example for the application of the adaptation model, altering the task tree to adapt the user interface.

```
<adaptations trigger="DisplayTooSmall" name="CreateTaskSequence">
    <steps target="//@adaptations.0/@rootQueries.0"
        parameters="//@adaptations.0/@rootQueries.0/@subQueries.0"
        executableElement="remove">
        <result name="removedFocusedTask" />
    </steps>
    <steps target="//@adaptations.0/@rootQueries.0"
        parameters="removedFocusedTask" executableElement="addAsFirst" />
    <steps target="//@adaptations.0/@rootQueries.0"
        parameters="true" executableElement="setIterative" />
    <steps target="//@adaptations.0/@rootQueries.0/@subQueries.1"
        parameters="Enabling" executableElement="setOperator" />
    <steps target="//@adaptations.0/@rootQueries.1"
        parameters="Task2ButtonMapping
        //@adaptations.0/@rootQueries.0/@subQueries.1
        //@adaptations.0/@rootQueries.2"
        executableElement="addMapping" />
    <rootQueries
        expression=".//TaskModel:children[TaskModel:name='CriteriaSelection']"
        model="TaskModel">
        <subQueries expression="/TaskModel:children[TaskModel:state='Focused']"/>
        <subQueries expression="/TaskModel:children" />
    </rootQueries>
    <rootQueries expression="./" model="MappingModel" />
    <rootQueries
        expression=".//AUI:interactors[AUI:name='NextCriteriaCommand']"
        model="AUI" />
    <variables value="Enabling" />
    <variables value="Task2ButtonMapping" />
    <variables value="true" />
</adaptations>
<executableElements name="addMapping" />
<executableElements name="addAsFirst" />
<executableElements name="remove" />
<executableElements name="setOperator" />
<executableElements name="setIterative" />
<situations name="DisplayTooSmall" />
```

Figure B.11.: Code of the adaptation model, performing the adaptation visualized in figure B.10.

B.6. Summary

In this section the implementation of a set of executable models using the Eclipse Modeling Framework has been described. The developed system utilizes eight executable models, connected via mappings. The integration of ECore and Java have been described and selected details of the developed models have been illustrated to provide a deeper understanding of the underlying realization and the utilization of the created Java objects at runtime. While each metamodel defines the Java classes of its building blocks and thus allows to realize the execution logic, the instantiation of the classes happens within the developed user interface models. As the objects are also hold in the memory by the Eclipse environment at design-time, a close coupling of design- and runtime is supported by this approach. Additionally, the described set of models and the mappings between them allow the creation of user interface models, suitable to fit the illustrated runtime infrastructure of the previous case study.

List of Figures

List of Tables

Bibliography

[Abascal et al. 2008] Abascal, J. ; Fernández De Castro, I. ; Lafuente, A. ; Cia, J. M.: Adaptive Interfaces for Supportive Ambient Intelligence Environments. In: *ICCHP '08: Proceedings of the 11th International Conference on Computers Helping People with Special Needs*. Berlin, Heidelberg : Springer-Verlag, 2008. – ISBN 978–3–540–70539–0, S. 30–37

[Abowd et al. 1999] Abowd, G. D. ; Dey, A. K. ; Brown, P. J. ; Davies, N. ; Smith, M. ; Steggles, P.: Towards a Better Understanding of Context and Context-Awareness. In: *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*. London, UK : Springer-Verlag, 1999. – ISBN 3–540–66550–1, S. 304–307

[Abrams et al. 1999] Abrams, M. ; Phanouriou, C. ; Batongbacal, A. L. ; Williams, S. M. ; Shuster, J. E.: UIML: An Appliance-Independent XML User Interface Language. In: *Computer Networks* 31 (1999), Nr. 11-16, 1695–1708

[Badros et al. 2001] Badros ; J., G. ; Borning, A. ; Stuckey ; J., P.: The Cassowary Linear Arithmetic Constraint Solving Algorithm. In: *ACM Transactions on Computer-Human Interaction* 8 (2001), Nr. 4, 267–306

[Baggia et al. 2008] Baggia, P. ; Burnett, D. C. ; Carter, J. ; Dahl, D. A. ; McCobb, G. ; Raggett, D.: EMMA: Extensible MultiModal Annotation markup language - W3C Proposed Recommendation 15 December 2008 / W3C. 2008

[Balme et al. 2004] Balme, L. ; Demeure, A. ; Barralon, N. ; Coutaz, J. ; Calvary, G.: CAMELEON-RT: A Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces. In: *EUSAI*, 2004, S. 291–302

[Bandelloni and Paternò 2004] Bandelloni, R. ; Paternò, F.: Flexible Interface Migration. In: *Proceedings of the 9th International Conference on Intelligent User Interfaces*. Funchal, Madeira, Portugal : ACM Press New York, NY, USA, 2004, 148 - 155

[Barralon et al. 2007] Barralon, N. ; Coutaz, J. ; Lachenal, C.: Coupling Interaction Resources and Technical Support. In: *HCI International 2007* Bd. 4555, 2007 (Lecture Notes in Computer Science), S. 13–22

[Bass et al. 1992] Bass, L. ; Faneuf, R. ; Little, R. ; Mayer, N. ; Pellegrino, B. ; Reed, S. ; Seacord, R. ; Sheppard, S. ; Szczur, M. R.: A Metamodel for the Runtime Architecture of an Interactive System: The UIMS Tool Developers Workshop. In: *SIGCHI Bull.* 24 (1992), Nr. 1, S. 32–37. – ISSN 0736–6906

[Becker 2008] Becker, M.: Software Architecture Trends and Promising Technology for Ambient Assisted Living Systems. In: Karshmer, A. I. (Ed.) ; Nehmer, J. (Ed.) ; Raffler, H. (Ed.) ; Tröster, G. (Ed.): *Assisted Living Systems - Models, Architectures and Engineering Approaches.* Dagstuhl, Germany : Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2008 (Dagstuhl Seminar Proceedings 07462). – ISSN 1862–4405

[Bernsen 1994] Bernsen, N. O.: Modality Theory in Support of Multimodal Interface Design. In: *Proceedings of the AAAI Spring Symposium on Intelligent Multi-Media Multi-Modal Systems*, 1994, 37-44

[Bernsen 1995] Bernsen, N. O.: A Toolbox of Output Modalities. Representing Output Information in Multimodal Interfaces. Esprit Basic Research Project AMODEUS-2 Working Paper RP5-TM-WP21 / Centre for Cognitive Science, Roskilde University. 1995

[Bernsen 1997a] Bernsen, N. O.: Towards a Tool for Predicting Speech Functionality. In: *Speech Commun.* 23 (1997), Nr. 3, S. 181–210. – ISSN 0167–6393

[Bernsen 2001] Bernsen, N. O.: Multimodality in Language and Speech Systems – From Theory to Design Support Tool. In: *From Theory to Design Support Tool*, Kluwer Academic Publishers, 2001

[Bernsen 1997b] Bernsen, N. O.: Defining a Taxonomy of Output Modalities from an HCI Perspective. In: *Comput. Stand. Interfaces* 18 (1997), Nr. 6-7, S. 537–553. – ISSN 0920–5489

[Berti et al. 2004] Berti, S. ; Correani, F. ; Paternò, F. ; Santoro, C.: The TERESA XML Language for the Description of Interactive Systems at Multiple Abstraction Levels. In: *Proceedings of Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages.* Gallipoli (LE), Italy, 2004, S. 103–110

[Berti et al. 2005] Berti, S. ; Paternò, F. ; Santoro, C.: A Taxonomy for Migratory User Interfaces. In: Harrison, M. (Ed.): *Proceedings of the 12th Int. Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'2005)*, Springer-Verlag, 2005

[Blattner and Glinert 1996] Blattner, M. ; Glinert, E.: Multimodal Integration. In: *IEEE Multimedia* 3 (1996), Winter, Nr. 4, S. 14–24. – ISSN 1070–986X

[Blattner et al. 1992] Blattner, M. ; Glinert, E. ; Jorge, J. ; Ormsby, G.: Metawidgets: Towards a Theory of Multimodal Interface Design. In: *Proceedings of the 16th International Computer Software and Applications Conference (COMPSAC '92)* (1992), Sep, S. 115–120

[Bleul et al. 2004] Bleul, S. ; Schaefer, R. ; Mueller, W.: Multimodal Dialog Description for Mobile Devices. In: Luyten, K. (Ed.) ; Abrams, M. (Ed.) ; Vanderdonckt, J. (Ed.) ; Limbourg, Q. (Ed.): *Developing User Interfaces with XML: Advances on User Interface Description Languages, Satellite Workshop of Advanced Visual Interfaces (AVI'04)*, 2004

[Bodart et al. 1995] Bodart, F. ; Hennebert, A.-M. ; Leheureux, J.-M. ; Provot, I. ; Sacré, B. ; Vanderdonckt, J.: Towards a Systematic Building of Software Architecture: The TRIDENT Methodological Guide. In: Palanque, P. (Ed.) ; Bastide, R. (Ed.): *Design, Specification and Verification of Interactive Systems '95.* Wien : Springer-Verlag, 1995, 262–278

[Bolt 1980] Bolt, R. A.: "Put that there": Voice and Gesture at the Graphics Interface. In: *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '80).* New York, NY, USA : ACM Press, 1980. – ISBN 0–89791–021–4, S. 262–270

[Bouchet et al. 2004] Bouchet, J. ; Nigay, L. ; Ganille, T.: ICARE Software Components for Rapidly Developing Multimodal Interfaces. In: *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces.* New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–995–0, S. 251–258

[Bourguet 2004] Bourguet, M.-L.: Software Design and Development of Multimodal Interaction. In: *IFIP Congress Topical Sessions*, 2004, S. 409–414

[Bregler and Konig 1994] Bregler, C. ; Konig, Y.: "Eigenlips" for Robust Speech Recognition. In: *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-94)* Bd. ii, 1994, S. II/669–II/672 vol.2

[Breton and Bézivin 2001] Breton, E. ; Bézivin, J.: Towards an Understanding of Model Executability. In: *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*. New York, NY, USA : ACM, 2001. – ISBN 1–58113–377–4, S. 70–80

[Bézivin 2005] Bézivin, J.: On the Unification Power of Models. In: *Software and System Modeling (SoSym)* 4 (2005), Nr. 2, 171–188

[Calvary et al. 2001a] Calvary, G. ; Coutaz, J. ; Thevenin, D.: Supporting Context Changes for Plastic User Interfaces: A Process and a Mechanism. In: A. Blandford, J. V. (Ed.) ; Gray, P. (Ed.): *Joint Proceedings of HCI'2001 and IHM'2001*. Lille : Springer-Verlag, September 2001, 349-363

[Calvary et al. 2001b] Calvary, G. ; Coutaz, J. ; Thevenin, D.: A Unifying Reference Framework for the Development of Plastic User Interfaces. In: *EHCI '01: Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*. London, UK : Springer-Verlag, 2001. – ISBN 3–540–43044–X, S. 173–192

[Calvary et al. 2003] Calvary, G. ; Coutaz, J. ; Thevenin, D. ; Limbourg, Q. ; Bouillon, L. ; Vanderdonckt, J.: A Unifying Reference Framework for Multi-Target User Interfaces. In: *Interacting with Computers* 15 (2003), Nr. 3, 289–308

[Calvary et al. 2002] Calvary, G. ; Coutaz, J. ; Thevenin, D. ; Limbourg, Q. ; Souchon, N. ; Bouillon, L. ; Florins, M. ; Vanderdonckt, J.: Plasticity of User Interfaces: A Revised Reference Framework. In: *TAMODIA '02: Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design*, INFOREC Publishing House Bucharest, 2002. – ISBN 973–8360–01–3, S. 127–134

[Card et al. 1991] Card, S. K. ; Mackinlay, J. D. ; Robertson, G. G.: A Morphological Analysis of the Design Space of Input Devices. In: *ACM Transactions on Information Systems* 9 (1991), Nr. 2, S. 99–122. – ISSN 1046–8188

[Carpenter 1992] Carpenter, B.: *The Logic of Typed Feature Structures*. New York, NY, USA : Cambridge University Press, 1992. – ISBN 0–521–41932–8

[Chai et al. 2005] Chai, J. Y. ; Pan, S. ; Zhou, M. X.: Mind: A Context-Based Multimodal Interpretation Framework in Conversational Systems. In: *Advances in Natural Multimodal Dialogue Systems* 30 (2005)

[Cheyer and Martin 2001] Cheyer, A. ; Martin, D.: The Open Agent Architecture. In: *Journal of Autonomous Agents and Multi-Agent Systems* 4 (2001), March, Nr. 1, S. 143–148

[Clerckx et al. 2004] Clerckx, T. ; Luyten, K. ; Coninx, K.: DynaMo-AID: A Design Process and a Runtime Architecture for Dynamic Model-Based User Interface Development. In: *EHCI/DS-VIS*, 2004, S. 77–95

[Clerckx et al. 2007] Clerckx, T. ; Vandervelpen, C. ; Coninx, K.: Task-Based Design and Runtime Support for Multimodal User Interface Distribution. In: *Proceedings of Engineering Interactive Systems 2007 (EHCI-HCSE-DSVIS'07)*, 2007

[Clerckx et al. 2006] Clerckx, T. ; Vandervelpen, C. ; Luyten, K. ; Coninx, K.: A Task-Driven User Interface Architecture for Ambient Intelligent Environments. In: *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*. New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–287–9, S. 309–311

[Coen 2001] Coen, M. H.: Multimodal Integration - A Biological View. In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 2001

[Cohen et al. 1989] Cohen, P. R. ; Dalrymple, M. ; Moran, D. B. ; Pereira, F. C. ; Sullivan, J. W.: Synergistic Use of Direct Manipulation and Natural Language. In: *SIGCHI Bull.* 20 (1989), Nr. SI, S. 227–233. – ISSN 0736–6906

[Cohen et al. 1994] Cohen, P. R. ; Cheyer, A. ; Wang, M. ; Baeg, S. C.: An Open Agent Architecture. In: *AAAI Spring Symposium on Software Agents*, AAAI Press, 1994, S. 1–8

[Cohen et al. 1997] Cohen, P. R. ; Johnston, M. ; McGee, D. ; Oviatt, S. ; Pittman, J. ; Smith, I. ; Chen, L. ; Glow, J.: Quickset: Multimodal Interaction for Distributed Applications. In: *ACM International Multimedia Conference*, 1997

[Coninx et al. 2003] Coninx, K. ; Luyten, K. ; Vandervelpen, C. ; Bergh, J. V. ; Creemers, B.: Dygimes: Dynamically Generating Interfaces for Mobile Computing Devices and Embedded Systems. In: Chittaro, L. (Ed.): *Mobile HCI* Bd. 2795, Springer, 2003 (Lecture Notes in Computer Science). – ISBN 3–540–40821–5, 256–270

[Coutaz 2006] Coutaz, J.: Meta-User Interfaces for Ambient Spaces. In: Coninx, K. (Ed.) ; Luyten, K. (Ed.) ; Schneider, K. A. (Ed.): *TAMODIA* Bd. 4385, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 978–3–540–70815–5, 1–15

[Coutaz et al. 2003] Coutaz, J. ; Balme, L. ; Lachenal, C. ; Barralon, N.: Software Infrastructure for Distributed Migratable User Interfaces. In: *Workshop At the Crossroads: The Interaction of HCI and Systems Issues in Ubicomp. 2003*, 2003

[Coutaz et al. 1993] Coutaz, J. ; Nigay, L. ; Salber, D.: The MSM Framework: A Design Space for Multi-Sensori-Motor Systems. In: Bass, L. J. (Ed.) ; Gornos-

TAEV, J. (Ed.) ; UNGER, C. (Ed.): *EWHCI* Bd. 753, Springer, 1993 (Lecture Notes in Computer Science). – ISBN 3–540–57433–6, S. 231–241

[Coutaz et al. 1995] COUTAZ, J. ; NIGAY, L. ; SALBER, D. ; BLANDFORD, A. ; MAY, J. ; YOUNG, R. M.: Four Easy Pieces for Assessing the Usability of Multimodal Interaction: The CARE Properties. In: *INTERACT*, 1995, S. 115–120

[Crease et al. 2000] CREASE, M. ; BREWSTER, S. ; GRAY, P.: Caring, Sharing Widgets: A Toolkit of Sensitive Widgets. In: *Proceedings of BCS HCI2000*, Springer, 2000, S. 257–270

[Delgado and Araki 2006] DELGADO, R. L.-C. ; ARAKI, M.: *Spoken, Multilingual and Multimodal Dialogue Systems.* John Wiley & Sons, Ltd, 2006

[Demeure et al. 2005] DEMEURE, A. ; CALVARY, G. ; SOTTET, J.-S. ; VANDERDONKT, J.: A reference model for distributed user interfaces. In: *TAMODIA '05: Proceedings of the 4th international workshop on Task models and diagrams.* New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–220–8, S. 79–86

[Demeure et al. 2008] DEMEURE, A. ; SOTTET, J.-S. ; CALVARY, G. ; COUTAZ, J. ; GANNEAU, V. ; VANDERDONKT, J.: The 4C Reference Model for Distributed User Interfaces. In: *The Fourth International Conference on Autonomic and Autonomous Systems (ICAS 2008)*, 2008

[Denecke and Yang 2000] DENECKE, M. ; YANG, J.: Partial Information in Multimodal Dialogue. In: *ICMI '00: Proceedings of the Third International Conference on Advances in Multimodal Interfaces.* London, UK : Springer-Verlag, 2000. – ISBN 3–540–41180–1, S. 624–633

[Dey 2000] DEY, A. K.: *Providing Architectural Support for Building Context-Aware Applications*, Georgia Institute of Technology, Diss., 2000

[Ding et al. 2006] DING, Y. ; ELTING, C. ; SCHOLZ, U.: Seamless Integration of Output Devices into Intelligent Environments: Infrastructure, Strategies and Implementation. In: *International Conference on Intelligent Environments IE06*, 2006

[Duarte 2008] DUARTE, C.: *Design and Evaluation of Adaptative Multimodal Systems*, Department of Informatics, University of Lisbon, Diss., March 2008

[Duarte and Carriço 2006] DUARTE, C. ; CARRIÇO, L.: A Conceptual Framework for Developing Adaptive Multimodal Applications. In: *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces.* New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–287–9, S. 132–139

[Dupuy-Chessa et al. 2005] Dupuy-Chessa, S. ; Bousquet, L. du ; Bouchet, J. ; Ledru, Y.: Test of the ICARE Platform Fusion Mechanism. In: *DSV-IS*, 2005, S. 102–113

[Elting and Hellenschmidt 2004] Elting, C. ; Hellenschmidt, M.: Strategies for Self-Organization and Multimodal Output Coordination in Distributed Device Environments. In: *Workshop on Artificial Intelligence in Mobile Systems 2004 In conjunction with UbiComp 2004, September 2004*, 2004

[Emiliani and Stephanidis 2005] Emiliani, P. L. ; Stephanidis, C.: Universal Access to Ambient Intelligence Environments: Opportunities and Challenges for People with Disabilities. In: *IBM Syst. J.* 44 (2005), Nr. 3, S. 605–619. – ISSN 0018–8670

[Encarnação and Kirste 2005] Encarnação, J. ; Kirste, T.: Ambient intelligence: Towards Smart Appliance Ensembles. (2005)

[Fang Chen 2005] Fang Chen, R. D. C. P. Julien Epps E. Julien Epps: NICTA-HCSNet Multimodal User Interaction Workshop - Outcomes Report / National ICT Australia & HSCNet. Australian Technology Park, Sydney, September 2005

[Feuerstack 2008] Feuerstack, S.: *A Method for the User-centered and Model-based Development of Interactive Applications*, Technische Universität Berlin, Diss., 2008

[Feuerstack et al. 2007] Feuerstack, S. ; Blumendorf, M. ; Albayrak, S.: Prototyping of Multimodal Interactions for Smart Environments based on Task Models. In: *Constructing Ambient Intelligence: AmI 2007 Workshops Darmstadt*, 2007

[Feuerstack et al. 2008] Feuerstack, S. ; Blumendorf, M. ; Schwartze, V. ; Albayrak, S.: Model-based Layout Generation. In: Bottoni, P. (Ed.) ; Levialdi, S. (Ed.): *Proceedings of the working conference on Advanced visual interfaces*, ACM, 2008

[Flippo et al. 2003] Flippo, F. ; Krebs, A. ; Marsic, I.: A Framework for Rapid Development of Multimodal Interfaces. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*. New York, NY, USA : ACM Press, 2003. – ISBN 1–58113–621–8, 109–116

[Floch et al. 2006] Floch, J. ; Hallsteinsen, S. ; Stav, E. ; Eliassen, F. ; Lund, K. ; Gjorven, E.: Using Architecture Models for Runtime Adaptability. In: *IEEE Software* 23 (2006), Nr. 2, S. 62–70. – ISSN 0740–7459

[Florins et al. 2006] Florins, M. ; Simarro, F. M. ; Vanderdonckt, J. ; Michotte, B.: Splitting Rules for Graceful Degradation of User Interfaces. In: *IUI '06: Proceed-*

ings of the 11th international conference on Intelligent user interfaces. New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–287–9, S. 264–266

[Foster 2002] Foster, M. E.: State of the Art Review: Multimodal Fission, COMIC Deliverable D6.1 / COMIC consortium. 2002

[Gajos and Weld 2004] Gajos, K. ; Weld, D. S.: SUPPLE: Automatically Generating User Interfaces. In: *IUI '04: Proceedings of the 9th international conference on Intelligent user interface.* New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–815–6, S. 93–100

[Garlan 2004] Garlan, S.-W. H. A.-C. S. B. S. P. D.; Cheng C. D.; Cheng: Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure. In: *Computer* 37 (2004), October, Nr. 10, S. 46–54. – ISSN 0018–9162

[Ghorbel et al. 2006] Ghorbel, M. ; Mokhtari, M. ; Renouard, S.: A Distributed Approach for Assistive Service Provision in Pervasive Environment. In: *WMASH '06: Proceedings of the 4th international workshop on Wireless mobile applications and services on WLAN hotspots.* New York, NY, USA : ACM, 2006. – ISBN 1–59593–470–7, S. 91–100

[Glinert and Wise 1996] Glinert, E. P. ; Wise, G. B.: Adaptive Multimedia Interfaces in PolyMestra. In: *First European Conference on Disability, Virtual Reality and Associated Technologies (ECDVRAT '96)*, 1996

[Grolaux 2007] Grolaux, D.: *Transparent Migration and Adaptation in a Graphical User Interface Toolkit*, Université Catholique de Louvain, Diss., 2007

[Grolaux et al. 2005] Grolaux, D. ; Vanderdonckt, J. ; Roy, P. V.: Attach Me, Detach Me, Assemble Me Like You Work. In: *Human-Computer Interaction - INTERACT 2005*, 2005

[Heider and Kirste 2002] Heider, T. ; Kirste, T.: Architecture Considerations for Interoperable Multi-modal Assistant Systems. In: *DSV-IS '02: Proceedings of the 9th International Workshop on Interactive Systems. Design, Specification, and Verification.* London, UK : Springer-Verlag, 2002. – ISBN 3–540–00266–9, S. 253–268

[Holzapfel and Fuegen 2002] Holzapfel, H. ; Fuegen, C.: Integrating Emotional Cues into a Framework for Dialogue Management. In: *ICMI '02: Proceedings of the 4th IEEE International Conference on Multimodal Interfaces.* Washington, DC, USA : IEEE Computer Society, 2002. – ISBN 0–7695–1834–6, S. 141

[Holzapfel et al. 2004] Holzapfel, H. ; Nickel, K. ; Stiefelhagen, R.: Implementation and Evaluation of a Constraint-based Multimodal Fusion System for Speech and 3D Pointing Gestures. In: *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–995–0, S. 175–182

[Hosobe 2001] Hosobe, H.: A Modular Geometric Constraint Solver for User Interface Applications. In: *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*. New York, NY, USA : ACM Press, 2001. – ISBN 1–58113–438–X, S. 91–100

[Hosobe 2005] Hosobe, H.: Solving Linear and One-Way Constraints for Web Document Layout. In: *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA : ACM Press, 2005. – ISBN 1–58113–964–0, S. 1252–1253

[Huebscher and McCann 2005] Huebscher, C. ; McCann, A.: An Adaptive Middleware Framework for Context-Aware Applications. In: *Personal Ubiquitous Comput.* 10 (2005), Nr. 1, S. 12–20. – ISSN 1617–4909

[Janssen et al. 1993] Janssen, C. ; Weisbecker, A. ; Ziegler, J.: Generating User Interfaces from Data Models and Dialogue Net Specifications. In: *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM Press, 1993. – ISBN 0–89791–575–5, S. 418–423

[Johnston 1998] Johnston, M.: Unification-based Multimodal Parsing. In: *ACL-36: Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*. Morristown, NJ, USA : Association for Computational Linguistics, 1998, S. 624–630

[Johnston and Bangalore 2000] Johnston, M. ; Bangalore, S.: Finite-State Multimodal Parsing and Understanding. In: *Proceedings of the 18th conference on Computational linguistics*. Morristown, NJ, USA : Association for Computational Linguistics, 2000. – ISBN 1–55860–717–X, S. 369–375

[Johnston and Bangalore 2005] Johnston, M. ; Bangalore, S.: Finite-State Multimodal Integration and Understanding. In: *Nat. Lang. Eng.* 11 (2005), Nr. 2, S. 159–187. – ISSN 1351–3249

[Johnston et al. 2002] Johnston, M. ; Bangalore, S. ; Vasireddy, G. ; Stent, A. ; Ehlen, P. ; Walker, M. ; Whittaker, S. ; Maloor, P.: MATCH: An Architecture for Multimodal Dialogue Systems. In: *Annual Meeting of the Association*

for Computational Linguistics. Morristown, NJ, USA : Association for Computational Linguistics, 2002, 376–383

[Johnston et al. 1997] Johnston, M. ; Cohen, P. R. ; McGee, D. ; Oviatt, S. L. ; Pittman, J. A. ; Smith, I.: Unification-based Multimodal Integration. In: *Proceedings of the eighth conference on European chapter of the Association for Computational Linguistics*. Morristown, NJ, USA : Association for Computational Linguistics, 1997, S. 281–288

[Katsurada et al. 2003] Katsurada, K. ; Nakamura, Y. ; Yamada, H. ; Nitta, T.: XISL: A Language for Describing Multimodal Interaction Scenarios. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*. New York, NY, USA : ACM Press, 2003. – ISBN 1–58113–621–8, S. 281–284

[Kay 1984] Kay, M.: Functional Unification Grammar: A Formalism for Machine Translation. In: *ACL-22: Proceedings of the 10th International Conference on Computational Linguistics and 22nd Annual Meeting on Association for Computational Linguistics*. Morristown, NJ, USA : Association for Computational Linguistics, 1984, S. 75–78

[Kirste 2004] Kirste, T.: DynAMITE - Dynamisch Adaptive Multimodale IT Ensembles. In: *Tagungsband: Forschungsoffensive "Software Engineering 2006"*, 2004

[Kirste and Rapp 2001] Kirste, T. ; Rapp, S.: Architecture for Multimodal Interactive Assistant Systems. In: *Statustagung der Leitprojekte "Mensch-Technik-Interaktion"*, 2001

[Klug and Kangasharju 2005] Klug, T. ; Kangasharju, J.: Executable Task Models. In: *Proceedings of TAMODIA 2005*. Gdansk, Poland : ACM Press, September 2005, S. 119–122

[Kobayashi et al. 2005] Kobayashi, N. ; Tokunaga, E. ; Kimura, H. ; Hirakawa, Y. ; Ayabe, M. ; Nakajima, T.: An Input Widget Framework for Multi-modal and Multi-device Environments. In: *Proceedings of the Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS '05)*, 2005

[Larson et al. 2003] Larson, J. A. ; Raman, T. ; Raggett, D. ; Bodell, M. ; Johnston, M. ; Kumar, S. ; Potter, S. ; Waters, K.: W3C Multimodal Interaction Framework - W3C NOTE 06 May 2003 / W3C. 2003

[Lehmann 2008] Lehmann, G.: *Model Driven Runtime Architecture for Plastic User Interfaces*, Technische Universität Berlin, Diplomarbeit, 2008

[Limbourg et al. 2004a] Limbourg, Q. ; Vanderdonckt, J. ; Michotte, B. ; Bouillon, L. ; Florins, M. ; Trevisan, D.: USIXML: A User Interface Description Language for Context-Sensitive User Interfaces. In: *Proceedings of the ACM AVI'2004 Workshop" Developing User Interfaces with XML: Advances on User Interface Description Languages*, 2004, S. 55–62

[Limbourg et al. 2004b] Limbourg, Q. ; Vanderdonckt, J. ; Michotte, B. ; Bouillon, L. ; López-Jaquero, V.: USIXML: A Language Supporting Multi-path Development of User Interfaces. In: Bastide, R. (Ed.) ; Palanque, P. A. (Ed.) ; Roth, J. (Ed.): *EHCI/DS-VIS* Bd. 3425, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–26097–8, 200–220

[Luyten et al. 2006a] Luyten, K. ; Thys, K. ; Vermeulen, J. ; Coninx, K.: A Generic Approach for Multi-device User Interface Rendering with UIML. In: *CADUI 2006: Computer-Aided Design of User Interfaces V*, 2006

[Luyten et al. 2005] Luyten, K. ; Vandervelpen, C. ; Bergh, J. V. ; Coninx, K.: Context-sensitive User Interfaces for Ambient Environments: Design, Development and Deployment. In: Davies, N. (Ed.) ; Kirste, T. (Ed.) ; Schumann, H. (Ed.): *Mobile Computing and Ambient Intelligence: The Challenge of Multimedia*, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005 (Dagstuhl Seminar Proceedings 05181). – ISSN 1862–4405

[Luyten et al. 2002] Luyten, K. ; Vandervelpen, C. ; Coninx, K.: Migratable User Interface Descriptions in Component-Based Development. In: *DSV-IS*, 2002, S. 44–58

[Luyten et al. 2006b] Luyten, K. ; Vermeulen, J. ; Coninx, K.: Constraint Adaptability of MultiDevice User Interfaces. In: Richter, K. (Ed.) ; Nichols, J. (Ed.) ; Gajos, K. (Ed.) ; Seffah, A. (Ed.): *Proceedings of the CHI*06 Workshop on The Many Faces of Consistency in Cross Platform Design*, 2006

[Maes and Saraswat 2003] Maes, S. H. ; Saraswat, V.: Multimodal Interaction Requirements - W3C NOTE 8 January 2003 / W3C. 2003

[Maoz 2008] Maoz, S.: Model-Based Traces. In: *3rd Int. Workshop on Models at Runtime at MoDELS'08*, 2008

[Martin 1998] Martin, J.-C.: TYCOON: Theoretical Framework and Software Tools for Multimodal Interfaces. In: *Intelligence and Multimodality in Multimedia interfaces, AAAI Press* (1998)

[Mellor 2004] Mellor, S. J.: Agile MDA / Project Technology, Inc. 2004

[Mellor et al. 2004] Mellor, S. J. ; Scott, K. ; Uhl, A. ; Weise, D.: *MDA Distilled: Principles of Model-Driven Architecture*. Boston : Addison-Wesley, 2004

[Mühlhäuser 2007] Mühlhäuser, M.: Multimodal Interaction for Ambient Assisted Living (AAL). In: Karshmer, A. I. (Ed.) ; Nehmer, J. (Ed.) ; Raffler, H. (Ed.) ; Tröster, G. (Ed.): *Assisted Living Systems - Models, Architectures and Engineering Approaches* Bd. 07462, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007 (Dagstuhl Seminar Proceedings)

[Mikalsen et al. 2006] Mikalsen, M. ; Floch, J. ; Paspallis, N. ; Papadopoulos, G. A. ; Ruiz, P. A.: Putting Context in Context: The Role and Design of Context Management in a Mobility and Adaptation Enabling Middleware. In: *MDM '06: Proceedings of the 7th International Conference on Mobile Data Management*. Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 0–7695–2526–1, S. 76

[Miller and Mukerji 2001] Miller, J. ; Mukerji, J.: *Model Driven Architecture (MDA)*. OMG Document ormsc/2001-07-01 edition. Object Management Group, Juli 2001

[Minsky 1975] Minsky, M.: A Framework for Representing Knowledge. In: *The Psychology of Computer Vision* (1975). ISBN 0–262–62101–0

[Müller et al. 2003] Müller, J. ; Poller, P. ; Tschernomas, V.: A Multimodal Fission Approach with a Presentation Agent in the Dialog System SmartKom. In: *KI 2003: Advances in Artificial Intelligence*, 2003

[Molina et al. 2006] Molina, J. ; Vanderdonckt, J. ; González, P. ; Fernández-Caballero, A. ; Lozano, M.: Rapid Prototying of Distributed User Interfaces. In: *Proceedings of 6th Int, Conf. on Computer-Aided Design of User Interfaces (CADUI'2006)*, Springer-Verlag, 2006, S. pp. 151–166

[Moran et al. 1997] Moran, D. B. ; Cheyer, A. J. ; Julia, L. E. ; Martin, D. L. ; Park, S.: Multimodal User Interfaces in the Open Agent Architecture. In: *IUI '97: Proceedings of the 2nd International Conference on Intelligent User Interfaces*. New York, NY, USA : ACM Press, 1997. – ISBN 0–89791–839–8, S. 61–68

[Mori et al. 2004] Mori, G. ; Paternò, F. ; Santoro, C.: Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions. In: *IEEE Trans. Softw. Eng.* 30 (2004), Nr. 8, S. 507–520. – ISSN 0098–5589

[Muller et al. 2005] Muller, P. A. ; Fleurey, F. ; Jézéquel, J. M.: Weaving Executability into Object-Oriented Meta-languages. In: Briand, L. C. (Ed.) ; Williams, C. (Ed.): *Proceedings of the 8th International on Model Driven Engineering Languages*

and Systems Bd. 3713. Montego Bay, Jamaica : Springer, October 2005 (Lecture Notes in Computer Science), S. 264–278

[Myers and Rosson 1992] Myers, B. A. ; Rosson, M. B.: Survey on User Interface Programming. In: *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA : ACM, 1992. – ISBN 0–89791–513–5, S. 195–202

[Neal and Shapiro 1988] Neal, J. G. ; Shapiro, S. C.: Intelligent Multi-Media Interface Technology. In: *SIGCHI Bull.* 20 (1988), Nr. 1, S. 75–76. – ISSN 0736–6906

[Nehmer et al. 2006] Nehmer, J. ; Becker, M. ; Karshmer, A. ; Lamm, R.: Living Assistance Systems: An Ambient Intelligence Approach. In: *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–375–1, S. 43–50

[Nichols et al. 2002] Nichols, J. ; Myers, B. ; Harris, T. K. ; Rosenfeld, R. ; Shriver, S. ; Higgins, M. ; HughesInc., J.: Requirements for Automatically Generating Multi-Modal Interfaces for Complex Appliances. In: *Fourth IEEE International Conference on Multimodal Interfaces (ICMI'02)*, 2002, S. p. 377

[Nichols et al. 2006] Nichols, J. ; Myers, B. A. ; Rothrock., B.: UNIFORM: Automatically Generating Consistent Remote Control User Interfaces. In: *Proceedings of CHI'2006*, 2006, S. pp. 611–620

[Nigay and Coutaz 1993] Nigay, L. ; Coutaz, J.: A Design Space for Multimodal Systems: Concurrent Processing and Data Fusion. In: *CHI '93: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM Press, 1993. – ISBN 0–89791–575–5, S. 172–178

[Nigay and Coutaz 1995] Nigay, L. ; Coutaz, J.: A Generic Platform for Addressing the Multimodal Challenge. In: *CHI '95: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1995. – ISBN 0–201–84705–1, S. 98–105

[Nigay and Coutaz 1997] Nigay, L. ; Coutaz, J.: Multifeature Systems: The CARE Properties and Their Impact on Software Design. In: *Intelligence and Multimodality in Multimedia Interfaces*. 1997

[Obj 2002] Object Management Group (Ed.): *Meta Object Facility (MOF) Specification — Version 1.4*. Object Management Group, April 2002

[Oviatt 1999] Oviatt, S.: Ten Myths of Multimodal Interaction. In: *Commun. ACM* 42 (1999), Nr. 11, S. 74–81. – ISSN 0001–0782

[Park and Kwon 2007] Park, T. H. ; Kwon, O.: Identifying a Generic Model of Context for Context-Aware Multi-services. In: *Proceedings of 4th International Conference* Bd. Volume 4611/2007, 2007 (LNCS)

[Paternò 1999] Paternò, F.: *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, 1999 (Applied Computing). – 208 S. – ISBN 1–85233–155–0

[Paternò 2005] Paternò, F.: Model-based Tools for Pervasive Usability. In: *Interacting with Computers* 17 (2005), Nr. 3, 291–315

[Paternò and Santoro 2002] Paternò, F. ; Santoro, C.: One Model, Many Interfaces. In: Kolski, C. (Ed.) ; Vanderdonckt, J. (Ed.): *CADUI*, Kluwer, 2002. – ISBN 1–4020–0643–8, S. 143–154

[Paterno et al. 2008] Paterno, F. ; Santoro, C. ; Mantyjarvi, J. ; Mori, G. ; Sansone, S.: Authoring Pervasive Multimodal User Interfaces. In: *International Journal of Web Engineering and Technology* 4 (2008), 235-261(27)

[Pavlovic 1998] Pavlovic, V.: Multimodal Tracking and Classification of Audio-Visual Features. In: *International Conference on Image Processing (ICIP 98)*, 1998, S. 343–347

[Pfleger 2004] Pfleger, N.: Context Based Multimodal Fusion. In: *ICMI '04: Proceedings of the 6th International Conference on Multimodal interfaces*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–995–0, S. 265–272

[Plomp and Mayora-Ibarra 2002] Plomp, J. ; Mayora-Ibarra, O.: A Generic Widget Vocabulary for the Generation of Graphical and Speech-Driven User Interfaces. In: *International Journal of Speech Technology V(5), Issue 1*, 2002

[Portillo et al. 2006] Portillo, P. M. ; García, G. P. ; Carredano, G. A.: Multimodal Fusion: A New Hybrid Strategy for Dialogue Systems. In: *ICMI '06: Proceedings of the 8th International Conference on Multimodal Interfaces*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–541–X, S. 357–363

[Pribeanu 2007] Pribeanu, C.: Tool Support for Handling Mapping Rules from Domain to Task Models. In: *Task Models and Diagrams for Users Interface Design* Bd. Volume 4385/2007, Springer Berlin / Heidelberg, 2007, S. 16–23

[Puerta and Eisenstein 2001] Puerta, A. ; Eisenstein, J.: XIML: A Universal Language for User Interfaces / RedWhale Software, 227 Town & Country, Palo Alto. 2001

[Puerta and Eisenstein 1999] Puerta, A. R. ; Eisenstein, J.: Towards a General Com-

putational Framework for Model-Based Interface Development Systems. In: *Intelligent User Interfaces*, 1999, 171-178

[Reithinger et al. 2003] Reithinger, N. ; Alexandersson, J. ; Becker, T. ; Blocher, A. ; Engel, R. ; Löckelt, M. ; Müller, J. ; Pfleger, N. ; Poller, P. ; Streit, M. ; Tschernomas, V.: SmartKom: Adaptive and Flexible Multimodal Access to Multiple Applications. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*. New York, NY, USA : ACM Press, 2003. – ISBN 1–58113–621–8, S. 101–108

[Richter 2006] Richter, K.: Transformational Consistency. In: *CADUI'2006 Computer-AIDED Design of User Interface V*, 2006

[Robbie Schaefer 2006] Robbie Schaefer, W. M. Steffen Bleul B. Steffen Bleul: Dialog Modelling for Multiple Devices and Multiple Interaction Modalities. In: *5th International Workshop on Task Models and Diagrams for UI design (TAMODIA'2006)*. Hasselt, Belgium, October 2006

[Rohr et al. 2006] Rohr, M. ; Boskovic, M. ; Giesecke, S. ; Hasselbring, W.: Model-driven Development of Self-managing Software Systems. In: *Proceedings of the Workshop "Models@run.time" at the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML'06) 2006*, 2006

[Rossi et al. 2005] Rossi, G. ; Gordillo, S. ; Lyardet, F.: Design Patterns for Context-Aware Adaptation. In: *SAINT-W '05: Proceedings of the 2005 Symposium on Applications and the Internet Workshops (SAINT 2005 Workshops)*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2263–7, S. 170–173

[Rubin et al. 1998] Rubin, P. ; Vatikiotis-Bateson, E. ; Benoit, C.: Audio-Visual Speech Processing (Special Issue). In: *Speech Cummunication* 26 (1998), S. 1–161

[Salber et al. 1999] Salber, D. ; Dey, A. K. ; Abowd, G. D.: The Context Toolkit: Aiding the Development of Context-Enabled Applications. In: *CHI*, 1999, 434-441

[Sanchez et al. 2008] Sanchez, M. ; Barrero, I. ; Villalobos, J. ; Deridder, D.: An Execution Platform for Extensible Runtime Models. In: *3rd Int. Workshop on Models at Runtime at MoDELS'08*, 2008

[Sandor et al. 2001] Sandor, C. ; Reicher, T. ; München, T. U.: CUIML: A Language for the Generation of Multimodal Human-Computer Interfaces. In: *In Proceedings of the European UIML conference*, 2001

[Schneider and Becker 2008] Schneider, D. ; Becker, M.: Runtime Models for Self-Adaptation in the Ambient Assisted Living Domain. In: *3rd Int. Workshop on Models at Runtime at MoDELS'08*, 2008

[da Silva 2001] Silva, P. P.: User Interface Declarative Models and Development Environments: A Survey. In: *Interactive Systems. Design, Specification, and Verification, 8th International Workshop, DSV-IS 2001, Glasgow, Scotland, Springer-Verlag Berlin.* Bd. 1946. Limerick, Ireland, 2001, S. 207–226

[Sottet et al. 2006a] Sottet, J.-S. ; Calvary, G. ; Favre, J.-M.: Mapping Model: A First Step to Ensure Usability for sustaining User Interface Plasticity. In: Pleuss, A. (Ed.) ; Bergh, J. V. (Ed.) ; Hussmann, H. (Ed.) ; Sauer, S. (Ed.) ; Boedcher, A. (Ed.) ; ACM/IEEE (Org.): *Model Driven Development of Advanced User Interfaces (MDDAUI 2006)* Bd. 214 ACM/IEEE, 2006, S. 51–54

[Sottet et al. 2006b] Sottet, J.-S. ; Calvary, G. ; Favre, J.-M.: *Models at Runtime for sustaining User Interface Plasticity.* 2006

[Sottet et al. 2007a] Sottet, J.-S. ; Calvary, G. ; Coutaz, J. ; Favre, J.-M.: A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces. In: *Proc. Engineering Interactive Systems 2007*, 2007

[Sottet et al. 2007b] Sottet, J.-S. ; Ganneau, V. ; Calvary, G. ; Coutaz, J. ; Demeure, A. ; Favre, J.-M. ; Demumieux, R.: Model-Driven Adaptation for Plastic User Interfaces. In: Baranauskas, M. C. C. (Ed.) ; Palanque, P. A. (Ed.) ; Abascal, J. (Ed.) ; Barbosa, S. D. J. (Ed.): *Human-Computer Interaction - INTERACT 2007, 11th IFIP TC 13 International Conference, Rio de Janeiro, Brazil, September 10-14, 2007, Proceedings, Part I* Bd. 4662, Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–74794–9, 397-410

[Stanciulescu 2008] Stanciulescu, A.: *A Methodology for Developing Multimodal User Interfaces of Information Systems*, Université Catholique de Louvain, Diss., 2008

[Stocq and Vanderdonckt 2004] Stocq, J. ; Vanderdonckt, J.: A Domain Model-Driven Approach for Producing User Interfaces to Multi-Platform Information Systems. In: *AVI '04: Proceedings of the working conference on Advanced visual interfaces.* New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–867–9, 395–398

[Stork and Hennecke 1996] Stork, D. G. (Ed.) ; Hennecke, M. E. (Ed.): *Speechreading by Humans and Machines.* Springer, 1996

[Streitz et al. 1999] Streitz, N. A. ; Geissler, J. ; Holmer, T. ; Konomi, S. ; Müller-Tomfelde, C. ; Reischl, W. ; Rexroth, P. ; Seitz, P. ; Steinmetz, R.: i-LAND:

An Interactive Landscape for Creativity and Innovation. In: *CHI '99: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM, 1999. – ISBN 0–201–48559–1, S. 120–127

[Szekely 1996] Szekely, P. A.: Retrospective and Challenges for Model-Based Interface Development. In: Bodart, F. (Ed.) ; Vanderdonckt, J. (Ed.): *DSV-IS*, Springer, 1996. – ISBN 3–211–82900–8, S. 1–27

[Tandler 2004] Tandler, P.: The BEACH Application Model and Software Framework for Synchronous Collaboration in Ubiquitous Computing Environments. In: *J. Syst. Softw.* 69 (2004), Nr. 3, S. 267–296. – ISSN 0164–1212

[Tandler et al. 2001] Tandler, P. ; Prante, T. ; Müller-Tomfelde, C. ; Streitz, N. ; Steinmetz, R.: Connectables: Dynamic Coupling of Displays for the Flexible Creation of Shared Workspaces. In: *UIST '01: Proceedings of the 14th Annual ACM Symposium on User interface Software and Technology*. New York, NY, USA : ACM Press, 2001. – ISBN 1–58113–438–X, S. 11–20

[Vanderdonckt 2005] Vanderdonckt, J.: A MDA-Compliant Environment for Developing User Interfaces of Information Systems. In: Pastor, O. (Ed.) ; Cunha, J. F. (Ed.): *CAiSE* Bd. 3520, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–26095–1, 16–31

[Vanderdonckt et al. 2007] *Kapitel* 4. In: Vanderdonckt, J. ; Calvary, G. ; Coutaz, J. ; Stanciulescu, A.: *Multimodality for Plastic User Interfaces: Models, Methods, and Principles*. 2007, S. 75–105

[Vandervelpen and Coninx 2004] Vandervelpen, C. ; Coninx, K.: Towards Model-based Design Support for Distributed User Interfaces. In: *NordiCHI '04: Proceedings of the third Nordic Conference on Human-Computer Interaction*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–857–1, S. 61–70

[Vandervelpen et al. 2005] Vandervelpen, C. ; Vanderhulst, G. ; Luyten, K. ; Coninx, K.: Light-Weight Distributed Web Interfaces: Preparing the Web for Heterogeneous Environments. In: *ICWE*, 2005, S. 197–202

[Vo et al. 1995] Vo, M. T. ; Houghton, R. ; Yang, J. ; Bub, U. ; Meier, U. ; Waibel, A. ; Duchnowski, P.: Multimodal learning interfaces. In: *Proceedings of the ARPA Spoken Language Systems Technology Workshop*, 1995

[Vo and Waibel 1993] Vo, M. T. ; Waibel, A.: A multimodal human-computer interface: combination of speech and gesture recognition. In: *Adjunct proceedings of InterCHI'93*, 1993

[Vo and Wood 1996] Vo, M. T. ; Wood, C.: Building an application framework for speech and pen input integration in multimodal learning interfaces. In: *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on* 6 (1996), May, S. 3545–3548 vol. 6

[Wahlster 2002] Wahlster, W.: SmartKom: Fusion and Fission of Speech, Gestures, and Facial Expressions. In: *Proc. of the 1st International Workshop on Man-Machine Symbiotic Systems*, 2002

[Wahlster 2003] Wahlster, W.: Towards Symmetric Multimodality: Fusion and Fission of Speech, Gesture, and Facial Expression. In: *KI 2003: Advances in Artificial Intelligence*, 2003

[Wahlster 2006] Wahlster, W. (Ed.): *SmartKom: Foundations of Multimodal Dialogue Systems*. Springer, 2006

[Waibel et al. 1995] Waibel, A. ; Vo, M. T. ; Duchnowski, P. ; Manke, S.: MULTIMODAL INTERFACES. In: *AIRJ'94*, 1995

[Wang 1995] Wang, J.: Integration of eye-gaze, voice and manual response in multimodal user interface. In: *IEEE International Conference on Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century.* Bd. 5, 1995, S. 3938–3942 vol.5

[Weber et al. 2005] Weber, W. (Ed.) ; Rabaey, J. M. (Ed.) ; Aarts, E. (Ed.): *Ambient Intelligence*. Springer, 2005

[Weiser 1993] Weiser, M.: Some computer science issues in ubiquitous computing. In: *Commun. ACM* 36 (1993), Nr. 7, S. 75–84. – ISSN 0001–0782

[Wu et al. 1999] Wu, L. ; Oviatt, S. ; Cohen, P.: Multimodal integration-a statistical view. In: *Multimedia, IEEE Transactions on* 1 (1999), Dec, Nr. 4, S. 334–341. – ISSN 1520–9210