Model-Driven Useware Engineering

Gerrit Meixner, Marc Seissler, and Kai Breiner

Abstract. User-oriented hardware and software development relies on a systematic development process based on a comprehensive analysis focusing on the users' requirements and preferences. Such a development process calls for the integration of numerous disciplines, from psychology and ergonomics to computer sciences and mechanical engineering. Hence, a correspondingly interdisciplinary team must be equipped with suitable software tools to allow it to handle the complexity of a multimodal and multi-device user interface development approach. An abstract, model-based development approach seems to be adequate for handling this complexity. This approach comprises different levels of abstraction requiring adequate tool support. Thus, in this chapter, we present the current state of our model-based software tool chain. We introduce the use model as the core model of our model-based process, transformation processes, and a model-based architecture, and we present different software tools that provide support for creating and maintaining the models or performing the necessary model transformations.

1 Introduction

Considering the interaction with technical devices such as a computer or a machine control panel, the users actually interact with a subset of these hardware and software components, which, in their entirety, make up the user interface [1]. Unfortunately, today's developers often disregard the most important component of

Gerrit Meixner
German Research Center for Artificial Intelligence (DFKI), Trippstadter Str. 122, 67663, Kaiserslautern, Germany
e-mail: Gerrit.Meixner@dfki.de

Marc Seissler
University of Kaiserslautern, Center for Human-Machine-Interaction, Gottlieb-Daimler Str. 42, 67663, Kaiserslautern, Germany
e-mail: Marc.Seissler@mv.uni-kl.de

Kai Breiner
University of Kaiserslautern, Software Engineering Research Group, Gottlieb-Daimler Str. 42, 67663, Kaiserslautern, Germany
e-mail: Breiner@cs.uni-kl.de

H. Hussmann et al. (Eds.): MDD of Advanced User Interfaces, SCI 340, pp. 1–26.
springerlink.com

an interactive system – the user – because of their inability to put themselves into the position of a user. Since usability, which is perceived in a subjective way, depends on various factors such as skills or experience, the user interface will be perceived by each user in a completely different way.

Moreover, in a highly competitive market that brings forth technically and functionally more and more similar or equal devices, usability as an additional sales argument secures a competitive advantage. In order to put stronger emphasis on users' and customers' needs, wishes, working styles, requirements, and preferences, and in order to consider them right from the beginning in all phases of the device development process, the responsible professional organizations in Germany, i.e., GfA, GI, VDE-ITG, and VDI/VDE-GMA, coined the term "Useware" for the above-mentioned subset and intersection of hardware and software, back in 1998 already [2].

The development of user interfaces for interactive software systems is a time consuming and therefore costly task, which is shown in a study [3]. By analyzing a number of different software applications, it was found that about 48% of the source code, about 45% of the development time, about 50% of the implementation time, and about 37% of the maintenance time is required for aspects regarding user interfaces. Petrasch argues that the time effort needed for implementing user interfaces – even 15 years after the study by Myers et al. [3] – is still at least 50% [4]. He justifies that the spread of interactive systems as well as their requirements have drastically increased over the last years. To be able to enforce the development of user interfaces more efficiently, a methodical procedure with an early focus on user and task requirements was seen as necessary.

Therefore, the systematic Useware Engineering Process, which calls for a comprehensive user, task, and use context analysis preceding the actual development, was developed [5]. Later in the Useware Engineering Process, interdisciplinary teams composed, for instance, of computer scientists, mechanical engineers, psychologists, and designers, continue developing the respective device in close collaboration with the ordering customer and its clients by constantly providing prototypes even in the very early development phases, thereby facilitating continuous, parallel evaluation (as depicted in Fig. 1).

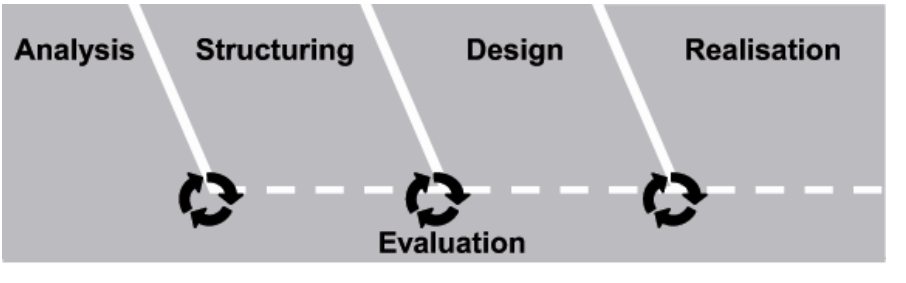

Fig. 1 Useware Engineering Process

The development process is determined by the procedure of ISO 13407 (user centered design) and follows the policy of ISO 9241-110 (dialog principles). In

the analyzing phase, the characteristics and behaviors of current and prospective users are defined using different methods, i.e., interviews, observations, task analysis, or surveys. At first, individual task models as well as communalities and differences between the user groups are derived from the requirements and behaviors while the system is used. Additionally, such issues as environmental and working conditions, team and labor organizations, as well as domain-specific context are explored during the analysis. Data elicited during the analysis can be entered, saved, analyzed, and exported using appropriate tools [6]. First of all, the structuring phase concentrates on harmonization and manual conflation of the individual task models and user requirements in order to obtain a common, system comprehensive, platform-independent use model. This model describes, e.g., what kinds of tasks can be performed or are allowed, for example for user groups at specific locations with specific devices. On the basis of classification, prioritization, and temporal relation of the tasks, an abstract operating structure is specified initially and saved in the XML-based Useware Markup Language [6]. The structure evaluation is an important and determining part of this phase. It guarantees the conformity of the harmonized structure and temporal sequences with the mental models of the users. It is already possible to simultaneously generate first preliminary models as well as executable use models for evaluation goals based on the use models and to test them by having the user use the respective software tools [7]. After the structuring phase, the actual design takes place. With the help of the user requirements and the results of the analysis, concepts of visualization, navigation, and interaction are chosen and combined appropriately. The design of coarse mask layouts is finally followed by the fine design of the ergonomically designed user masks, with the focus being on providing efficient support for the user as well as information brokering in a quick and systematic way in order to offer the user adequate decision-making aids. In parallel to the fine design, the realization starts, meaning the concrete implementation of the developed concepts into a user interface using the selected hard- and software platforms. The parallel evaluation represents a continuation of the analysis phase [8], since the development results are tested and evaluated continuously with structural or executable prototypes in every phase of the development process by representative users. To ensure that each user evaluation is taken into account, the user interfaces are improved iteratively. Adjustments to the use model can be made by returning from later phases to earlier ones, for example.

Any process is useless if developers do not adhere to it or accidently execute the process in a wrong way. To support the correct execution of the Useware Engineering Process, tools are indispensable. Furthermore, the constantly increasing number of heterogeneous platforms (PC, smart phone, PDA, etc.) is another reason why user interfaces have to be kept consistent with each other in relation to user experience on such target platforms in order to guarantee intuitional handling and thus ensure usability [9]. Since usability is a subjectively experienced non-functional requirement, there is no such thing as a best user interaction concept. In order to reduce the recurrent development effort of individual solutions for specific platforms, modalities, or even context of use, a model-based approach – which facilitates reusability – can be taken, focusing on the needs and

requirements of the users. This allows developers to generate prototypes of the user interface on the basis of the use model from the very beginning of their work.

In the subsequent sections, we will first discuss related work, followed by a section that focuses on supporting user interface developers during development time. Then we will introduce the CAMELEON Reference Framework as a meta architecture for model-based user interface development and subsequently present our derived model-based architecture. Furthermore, we will introduce useML 2.0 and the graphical useML-editor "Udit". Targeting the transition from the use model to the abstract user interface design, we present the Dialog and Interface Specification Language (DISL). Furthermore, we will introduce the User Interface Markup Language (UIML), which represents the concrete user interface. Additionally, we will introduce two mapping approaches: from useML to DISL and from DISL to UIML. Finally, we provide an outlook on currently developed as well as planned extensions to the tool chain.

2 Related Work

Many problems and limitations of current model-based architectures are a consequence of focusing too much on just one model [10]. For example, MECANO [11] and TRIDENT [12], [13] are architectures that do not integrate different discrete models.

TADEUS [14], [15] is an architecture and development environment for the development of user interfaces considering the application functionality as well as task modeling aspects. The dialog and the presentation model are generated from the task model on the basis of predefined classifications. The focus of TADEUS is on modeling application functionality. Regarding temporal operators, TADEUS only integrates a sequence operator.

GLADIS+ [16] and ADEPT [17] are architectures for model-based development on the basis of task models. The overall usability of these architectures is rather low. These architectures make use of classical formal models, such as entity relation (ER) models, to drive the greatest possible degree of automation regarding the user interface design process. As a consequence, only user interfaces with poor visual presentation can be generated [18].

In MOBI-D [19], an informal textual representation of the task and domain model is used to start the development process. MOBI-D is rather a set of tools than an architecture, and generates (semi-)automatic user interfaces [20]. MOBI-D cannot be used for developing multi-platform or multi-modality user interfaces.

One of the most recent architectures and XML-based development environments for multi-modal user interfaces is the Transformation Environment for inteRactivE Systems representAtions (TERESA) [21], [22]. Basically, TERESA consists of a task- and presentation model. On the basis of an abstract description of the task model in the ConcurTaskTree (CTT) notation [23], a developer is able to (semi-)automatically develop platform-specific task models, abstract presentation models, concrete presentation models, and finally HTML source code [24]. TERESA was developed as a monolithic development environment with an integrated simulator for evaluating models. The focus of TERESA, based on the task

model specified with CTT, is on supporting developers by offering different tools [25], [20]. Besides the task model, the developers need to specify further design decisions in order to transform the task model into a presentation model (specified with TeresaXML [26]). Interaction tasks in CTT do not contain the necessary semantics for transforming tasks into abstract interaction objects fully automatically [20], [9]. Furthermore, transformation processes are integrated directly into the source code of TERESA, which reduces the flexibility of the transformation processes in terms of extension, modification, and maintenance [27]. In TERESA, finite state machines are used to describe the dialog model, which is therefore quite limited in its expressiveness [28]. Recent work has been about the development of MARIA [29], the successor of TERESA.

Similar to TERESA, DYGIMES (Dynamically Generating Interfaces for Mobile and Embedded Systems) [20] is an architecture for the development of user interfaces based on different XML-compliant languages. DYGIMES aims at simplifying the development process by clearly separating the user interfaces from the application functionality. Furthermore, DYGIMES aims at reducing the complexity of the different models used. The focus of DYGIMES is on the automatic generation of a dialog and presentation model from a task model specified with CTT at runtime. The dialog and presentation model is described with SeescoaXML (Software Engineering for Embedded Systems using a Component-Oriented Approach) [25]. Task models are also specified with CTT, which needs additional abstract UI descriptions [20] to transform the task model into a dialog and presentation model. Luyten adapts the Enabled Task Sets (ETS) approach from Paternò [23] and introduces an optimized ETS-calculation algorithm [9]. After ETS calculation, designers can specify spatial layout constraints, which allow expressing how the single UI building blocks are grouped and aligned at the user interfaces. Finally, the generated user interfaces are rendered by a light-weight runtime environment running, for example, on the target mobile device.

3 Useware Engineering at Development Time

In this section, we will give a short overview of the CAMELEON reference framework – which is a well-established refinement framework for the systematic development of user interfaces on the basis of different models. In accordance with the refinement steps of this particular framework, we will introduce our architecture as a concrete instantiation of the CAMELEON reference framework.

3.1 CAMELEON – A Reference Framework

For many years, there has been much intensive research on using model-based development methodologies in the development of user interfaces [30]. These methodologies are very similar to model-based approaches in the domain of software engineering. Key aspects like model abstraction and using transformations to automatically generate further models or source code (e.g., used in Model Driven Architecture (MDA) in software engineering) are also important factors in the development of consistent user interfaces [7].

The CAMELEON reference framework was developed by the EU-funded CAMELEON project [31]. It describes a framework that serves as a reference for classifying user interfaces that support multiple targets, or multiple contexts of use on the basis of a model-based approach. The framework covers both the design time and runtime phases of multi-target user interfaces. Furthermore, the CAMELEON reference framework provides a unified understanding of context sensitive user interfaces rather than a prescription of various ways or methods for tackling different steps of development.

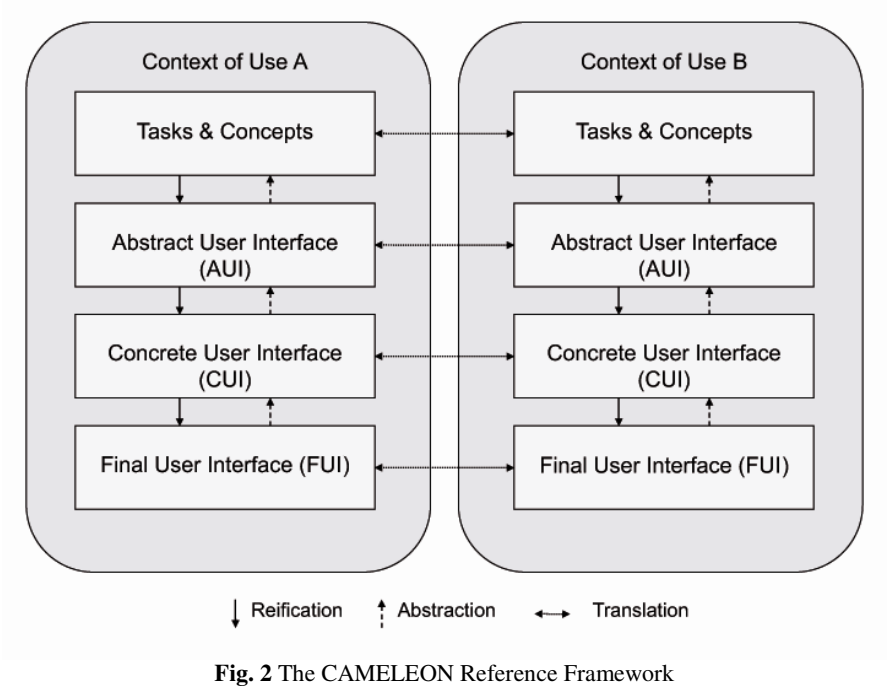

Fig. 2 The CAMELEON Reference Framework

As depicted in Fig. 2, the framework describes different layers of abstraction, which are important for model-based development of user interfaces, and their relationships among each other [32]:

- The **Task and Concepts level** considers, e.g., the hierarchies of tasks that need to be performed in a specific temporal order in order to achieve the users' goals (during the interaction with a user interface).
- The **Abstract User Interface** (AUI) expresses the user interface in terms of abstract interaction objects (AIO) [12]. These AIOs are independent of any platform or modality (e.g., graphical, vocal, haptic). Furthermore, AIOs can be grouped logically.
- The **Concrete User Interface** (CUI) expresses the user interface in terms of concrete interaction objects (CIO). These CIOs are modality

dependent but platform independent. The CUI defines more concretely how the user interface is perceived by the users.

- The **Final User Interface** (FUI) expresses the user interface in terms of platform-dependent source code. A FUI can be represented in any programming language (e.g., Java) or mark-up language (e.g., HTML). A FUI can then be interpreted or compiled.

Between these levels, there are different relationships: *reification* (forward engineering), *abstraction* (reverse engineering), and *translation* (between different contexts of use).

Fig. 3 shows an example (a simple graphical log-in screen) of the different layers of the CAMELEON reference framework. Starting with the "task & concepts" layer modeling the log-in task, the AUI, CUI, and FUI layers can be (semi-) automatically derived via transformations.

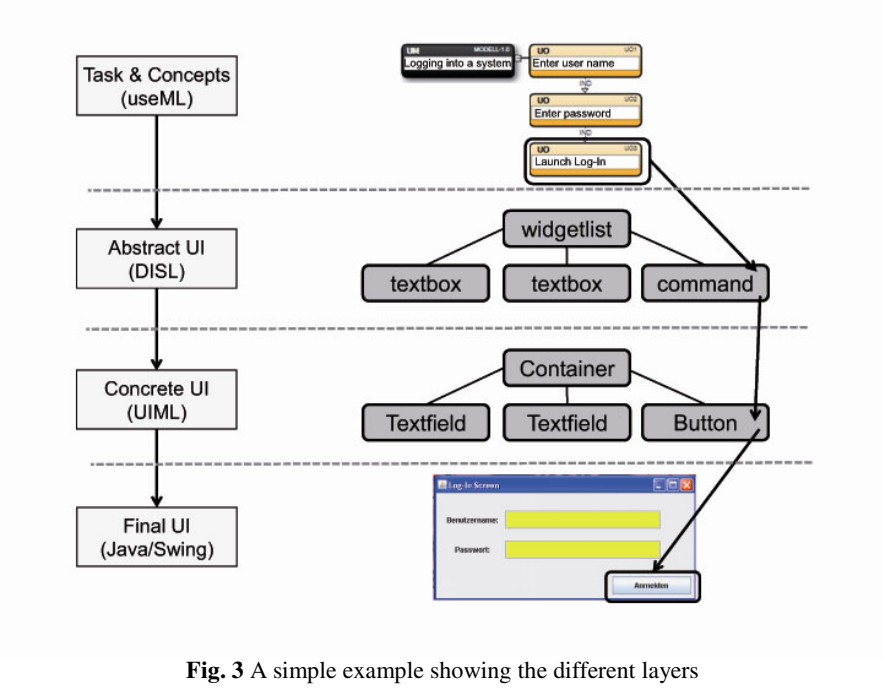

Fig. 3 A simple example showing the different layers

3.2 An Architecture for the Model-Based Useware Engineering Process

Different models are required in model-based development of user interfaces. The entity of models used is known as "interface model" and consists of different abstract and concrete models [30]. Abstract models are, e.g., the user model (represents different user groups, for example), the platform model (specifies target

platforms), the context model (describes the context of use), as well as the task model (describes tasks and actions of the user). The visualization of the user interface is defined by the presentation model. It specifies how visual, haptic, or voice interaction objects of the user interfaces are specified. The dialog model is the link between the task model and the presentation model. It describes the operating sequence, the starting point, the goal, and the results that control the process of the user interface. Furthermore, the presentation model and the dialog model are divided into an abstract and a concrete model part. Especially the abstract presentation and dialog models are characterized by a lack of references to specific modalities and platforms. As a result, transformations into any modality or platform can be realized.

The CAMELEON reference framework is the starting point for developing and integrating our own model-based architecture. This reference framework leaves open aspects regarding the practical composition of models and how to use them in user-centered development processes (such as the Useware Engineering Process). Therefore, we adapted the framework and developed our own model-based architecture, which integrates perfectly into the different phases of the Useware Engineering Process (see Fig. 4).

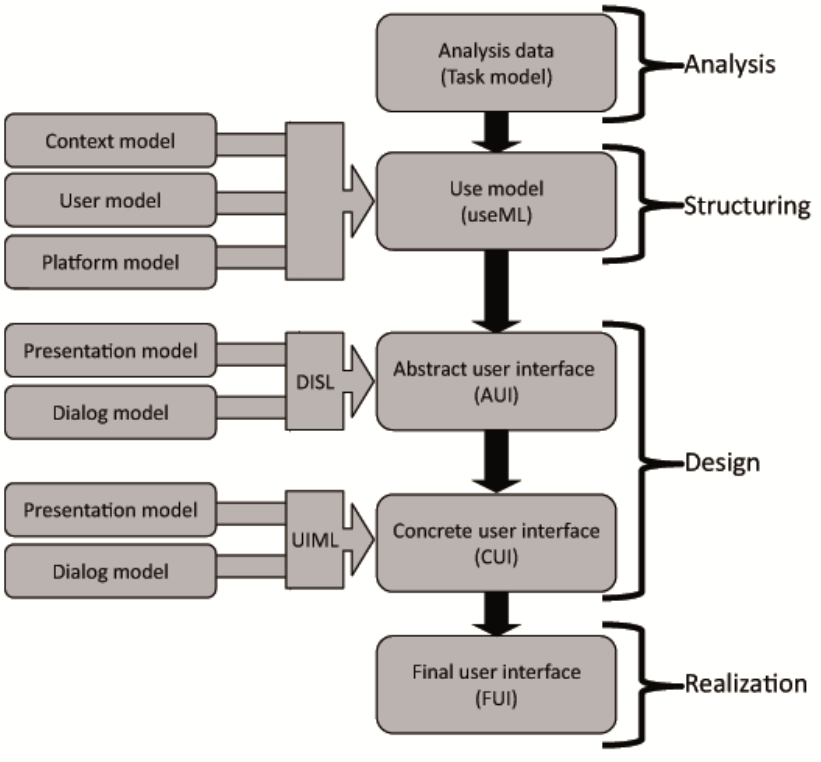

Fig. 4 Schematic of the model-based architecture

The first step consists of a survey of elicited analysis data, from which the task models of the individual users can be extracted. After harmonizing the analysis data, task models are combined (manually) during the structuring phase into a singular use model (see sections 3.3 and 3.4), which also integrates other abstract models (e.g., context, user, and platform model). Together, the analysis and structuring phases can be mapped to the "task & concepts" layer of the CAMELEON reference framework. The abstract user interface is built on the basis of the abstract presentation model and the abstract dialog model, which can be described using the Dialog and Interface Specification Language (DISL) (see section 3.5). With DISL, it is possible to describe platform- and modality-independent user interfaces during the design phase (abstract user interface). With the standardized User Interface Markup Language (UIML), concrete graphical user interfaces (design phase) can be described (see section 3.6). In our architecture, UIML covers the concrete presentation model as well as the concrete dialog model. By making use of an appropriate generic vocabulary [33], it is possible to transform UIML into a final user interface (realization phase) by generating source code or other markup languages (such as HTML) directly.

Although based on models and transformation processes, it is also necessary to integrate further models, transformations, tools, etc. into a holistic view and put them into order. As Schaefer shows, the overall architecture of model-based user interface processes consists of a number of further important components [28]. For the efficient development of user interfaces, respectively for the interactive processing of models, development teams additionally need software support, e.g., a model-based tool chain. This tool chain integrates model transformation engines (see sections 3.7 and 3.8), model editors, knowledge bases, as well as databases. Moreover, an execution environment is required, consisting of layout generators (e.g., for ordering graphical elements), HCI patterns (for reusing the designers' expert knowledge), and source code generators.

Finally, an application skeleton of the user interface can be generated via source code generators in the preferred programming language. This application skeleton can then be extended by functional characteristics until a complete vertical application prototype in a particular development environment is finished. This vertical application prototype can then again be tested iteratively by the users of the interactive system.

3.3 The Useware Markup Language 2.0

The Useware Markup Language (useML) 1.0 [34] was developed to support the user- and task-oriented Useware Engineering Process with a modeling language representing the results of the initial task analysis. Accordingly, the use model abstracts platform-independent tasks into use objects (UO) that make up a hierarchically ordered structure (see Fig. 5). Each element of this structure can be annotated by attributes such as eligible user groups, access rights, and importance. Further annotations and restrictions can be integrated by extending a dynamic part of the use model (e.g., for integrating information from the platform or context model). This functionality makes the use model more flexible than many other

task models and their respective task modeling languages (cf. section 2). Furthermore, the leaf tasks of a use model are described with a set of elementary use objects (eUO) representing atomic interactive tasks: inform, trigger, select, enter, and change**.** In contrast to other task modeling languages such as CTT [23] (see section 2), an eUO refines an interaction task, i.e., an eUO can be mapped directly to a corresponding abstract interaction object in the abstract user interface.

The basic structure of the use model has not been changed since 2004 [34], but the development of a taxonomy for task models and its application to the use model have revealed certain shortcomings and potentials for enhancing the use model extensively [35]. All these enhancements have been incorporated into useML 2.0 as introduced below.

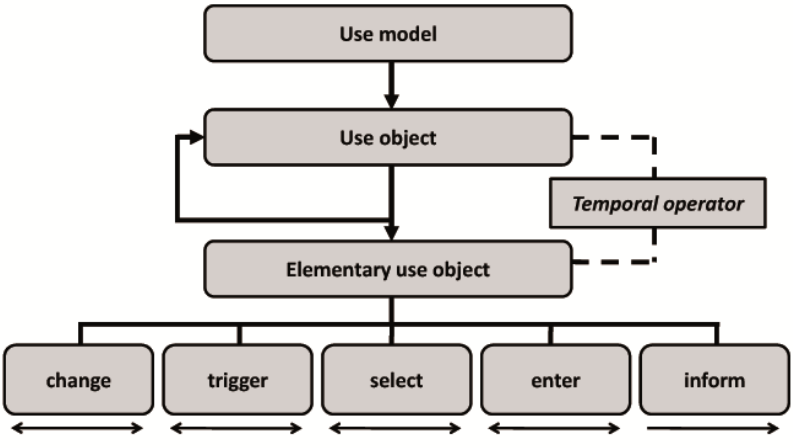

Fig. 5 Schematic of the use model

According to [36], the use model must differentiate between interactive user tasks (performed via the user interface) and pure system tasks requiring no active intervention by the user. System tasks encapsulate tasks that are fulfilled solely by the system – which, however, does not imply that no user interface must be presented, because the user might decide, for example, to abort the system task, or request information about the status of the system. Interactive tasks usually require the user(s) to actively operate the system, but still, there can be tasks that do not have to be fulfilled or may be tackled only under certain conditions. In any case, however, interactive tasks are usually connected to system tasks and the underlying application logic, which has been addressed recently by the newly introduced differentiation of user tasks and system tasks in useML 2.0.

To specify that a certain task is optional, the semantics of the use objects and the elementary use objects has been enhanced to reflect their importance. Their respective user actions can now be marked as “optional” or “required”.

Similarly, only useML 2.0 can attribute cardinalities to use objects and elementary use objects. These cardinalities can specify minimum and maximum frequencies of utilization, ranging from 0 for optional tasks up to ∞. Further, respective logical and/or temporal conditions can now be specified, as well as invariants that must be fulfilled at any time during the execution (processing) of a task. Except for useML 2.0, only few task modeling languages are able to specify both logical and temporal conditions. Consequently, temporal operators (see [7]) have been added to useML, which is the most important and most comprehensive enhancement in version 2.0. These operators allow for putting tasks on one hierarchical level into certain explicitly temporal orders; implicitly, temporal operators applied to neighboring levels of the hierarchical structure can form highly complex, temporal expressions. In order to define the minimum number of temporal operators that allows for the broadest range of applications, the temporal operators of 18 task modeling languages were analyzed and compared [37]. Among others, Tombola [38], VTMB [39], XUAN [40], MAD [41], DIANE+ [42], GTA [43], and CTT [23] were examined closely. Based on their temporal operators' relevance and applicability in a model-based development process, the following binary temporal operators were selected for useML 2.0:

- **Choice** (CHO): Exactly one of two tasks will be fulfilled.
- **Order Independence** (IND): The two tasks can be accomplished in any arbitrary order. However, when the first task has been performed, the second one has to wait for the first one to be finalized or aborted.
- **Concurrency** (CON): The two tasks can be accomplished in any arbitrary order, even in parallel at the same time (i.e., concurrently).
- **Deactivation** (DEA): The second task interrupts and deactivates the first task.
- **Sequence** (SEQ): The tasks must be accomplished in the given order. The second task must wait until the first one has been fulfilled.

Since the unambiguous priority of these four temporal operators is crucial for the connection of the use model with a dialog model, their priorities (i.e., their order of temporal execution) have been defined as follows [9]:

Choice > Order Independence > Concurrency > Deactivation > Sequence

3.4 The Graphical useML 2.0-Editor

Editors, simulators, and model transformation tools are needed that allow creating, testing, as well as processing the user interface models. To address these demands, the useML-Editor (Udit) was introduced [7], which allows the graphical editing of useML 2.0 models. Udit enables the developer to create and manipulate use models easily and quickly via a simple, graphical user interface. It further provides a validation mechanism for ensuring the correctness of a use model and the integrity of a use model to be loaded from a useML file. In case of problems, Udit shows appropriate warnings, hints, and error messages (use models created and saved using Udit are always valid).

Using project-specific conditions and constraints, useML provides an external schema attribute definition that can be changed at any time, without necessitating changes to the core useML schema. For example, user group names, personas and roles, locations, device types and specifications, the devices' function models, etc., are highly variable. While these conditions and constraints are specified in an external XML file in useML, Udit provides a schema editor to edit them quickly and easily.

As can be seen in Fig. 6 (left part), the basic elements of the useML 2.0 specification, i.e., use model (root element, black), use objects (orange), active task elementary use objects (green), and elementary use objects of the passive "inform" type (blue), are displayed in different colors. This facilitates the developer's orientation and navigation, especially when a developer works with complex use models. Collapsing and expanding sub-trees of the use model is also possible. The temporal operators are displayed as part of the connection line between two neighboring (elementary) use objects.

Udit has been designed to support the features of the recently revised useML 2.0. Since then, the initial version of Udit has been consistently enhanced. Udit now implements the transformation process from the use model to the abstract user interface. An integrated filter mechanism can be used to automatically derive a specific use model from the basic use model. This specific use model can be refined by the developer, or it can be automatically exported. Additional features, such as drag&drop functionality, a model validation tool, and a zoom function, have been incorporated.

To visualize the dynamic behavior of the use model, a simulator has been integrated in Udit, which can be used to evaluate the behavior of the developed use model. As depicted in Fig. 6 (right part), the simulator is split into four main screens: On the left side of the window, the simulated use model is displayed. The eUOs that are enabled for execution are highlighted in this use model and listed in a box, located in the right upper window. Each of the listed eUOs can be executed by pressing the corresponding "execute" button, which triggers the simulator to load a new set of executable eUOs. Additional features of the simulator include an execution history and a window for displaying conditions.

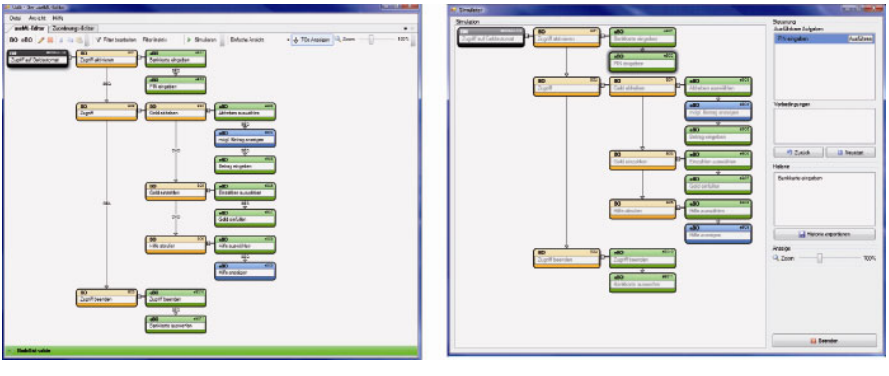

Fig. 6 Udit 2.0 - The useML Editor (left) and Simulator (right)

3.5 The Abstract User Interface

The abstract user interface is modeled with the Dialog and Interface Specification Language (DISL) [44], which was developed at the University of Paderborn (Germany) as a modeling language for platform- and modality-independent user interfaces for mobile devices. DISL focuses on scalability, reactivity, easy usability for developers, and low demands on processing power and memory consumption. An important precondition to the systematic development of user interfaces is the strict separation of structure, presentation, and behavior of a user interface. Since the User Interface Markup Language (UIML) [45] facilitates not only this separation, but also – by employing a XML-based file format – the easy creation of interpreters and development tools, UIML was used as a basis for the development of DISL. Therefore, the basic structure and the syntax of UIML were partially adapted. However, two UIML properties that shall be presented here in more detail did not fulfill the purpose of DISL. These are UIML's limited behavior model and its dependence on platform specifications.

UIML allows for the event-based behavior description of user interfaces. Events like pressing a key can lead to changes in the state of the respective user interface. Therefore, it is possible to specify the behavior of a user interface as a finite state machine in UIML. This is intuitive for simple user interfaces. In bigger projects, the developer is likely to lose track of the exponentially growing number of state transitions. In the past, this has been the reason why mechanisms and notations were introduced that significantly reduce the complexity of the state space, for example by employing parallel state transitions as in [46]. This, however, requires storing complex user interface states, such as "menu item 1 selected AND switch set to C". Instead of storing numerous complex states, DISL introduces state variables, resulting in state transitions being calculated from relevant state variable values at the occurrence of certain events. This also allows for setting state variables arbitrarily during a state transition. Finally, DISL also provides means for specifying time-dependent transitions, which is of high relevance for mobile applications where reactive user interfaces are to be designed even in unreliable networks, e.g., when a waiting period times out and an alternative interaction method must be provided to the user.

The second significant difference between DISL and UIML is the consequent abstraction of the DISL modeling language from any target platforms and modalities, which makes DISL a pure dialog modeling language. In UIML, on the contrary, abstract descriptions of the user interfaces are possible, but mapping between abstract items and concrete target platform items – the so-called "vocabulary" – is mandatory. DISL, however, uses only purely abstract interaction objects (AIO, see [10]); it is up to the implementation to either interpret AIOs directly on the target device (as presented, for example, in [44]), or to convert the abstract specification into a modality-dependent code using (external) transformations (see section 3.8). This supports DISL's objective of being scalable, since the abstract interaction objects possess only the minimal set properties that must be available on many systems. Fig. 7 shows as a proof of concept a simple interface for a media player modeled with DISL. The left part of Fig. 7 shows an emulated

Siemens M55 mobile phone, whereas the right part of Fig. 7 shows a real Siemens M55 mobile phone. Both mobile phones – the emulated and the real one – render the corresponding DISL document. The generated UI is functional but not very appealing; however, the AIOs could later be augmented during the transformation phase, e.g., by incorporating HCI patterns as design knowledge, in order to generate better interfaces on the respective end device.

Fig. 7 Simple User Interface for a Siemens M55 mobile phone generated from DISL

Adopting DISL into the Useware Engineering Process and linking it to the use models, finally completes the transformation-based, holistic Useware Engineering Process, as illustrated in [28]. For the development of DISL itself, not the whole user interface development process was taken into account, but, on purpose, only the dialog modeling and the presentation, either through direct interpretation on an end device or through transformation into a target format.

3.6 The Concrete User Interface

The concrete user interface is modeled with the User Interface Markup Language (UIML) [45]. UIML separates presentation components (e.g., widgets and layout), dynamic behavior (e.g., state transitions), and the content of a user interface (see Fig. 8). For instantiating a user interface in UIML, a UIML document and a specific vocabulary are required.

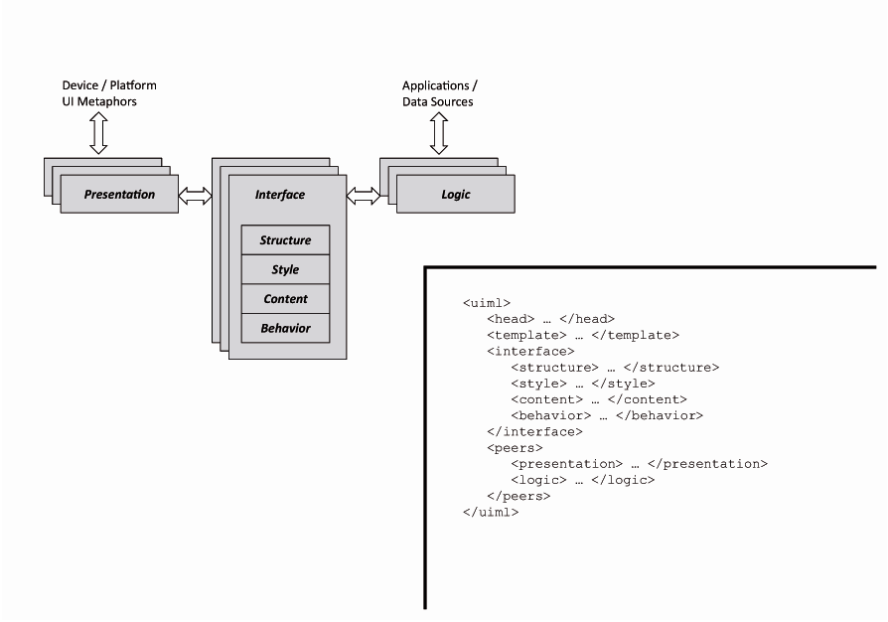

Fig. 8 The UIML Meta-Interface Model

The *interface* section of a UIML document consists of five components: structure, style, layout, content, and behavior:

- *Structure* describes a hierarchy of interaction objects and their relationships.
- *Style* specifies the properties of the components, e.g., text, color, or size.
- *Layout* defines how the components are arranged relative to each other (spatial constraints).
- *Content* separates the content of the interface from the other parts and is referenced in the components' properties.
- *Behavior* describes, for example, interaction rules or actions to be triggered under various circumstances (specifies the dialog model).

While the *interface* section describes the general appearance of the user interface, the *peers* section of a UIML document describes the concrete instantiation of the user interface by providing a mapping onto a platform-specific language (i.e., interface toolkits or bindings to the application logic).

- *Logic* specifies how the interfaces are bound to the application logic.
- *Presentation* describes the mapping to a concrete interface toolkit, such as Java Swing.

Furthermore, a UIML document includes an optional <head>-element for providing meta-information and a concept that allows the reuse of predefined components. These so-called "templates" are predefined interaction objects, which can be easily instantiated with concrete parameters derived from the application data model.

Since syntax and functionality of DISL were still close to UIML, several fundamental enhancements and improvements of DISL were incorporated into the new 4.0 version of UIML [47]. Since May 2009, UIML 4.0 is a standard of the Organization for the Advancement of Structured Information Standards (OASIS). DISL's abstractions accounting for platform and modality independence, however, are adopted by UIML 4.0 because of their fundamentally different mechanism. Still, platform independence of graphical user interfaces can now be achieved using UIML with a generic vocabulary, as demonstrated in [48].

3.7 Transformation of useML 2.0 into DISL

To support the developer in designing user interfaces, an automatic transformation process has been developed [49]. This process adapts the transformation process used in the TERESA development methodology [26] and consists of four phases depicted in Fig. 9. While in the (optional) first phase, the developer manually refines the use model – e.g., for the target platform or target user group – the subsequent phases gradually and automatically transform the use model into an abstract user interface.

While the transformation process introduced in [26] transforms a task model into a final user interface, we explicitly focus on mapping the use model onto an abstract user interface. Since this is compliant with the architecture proposed with the CAMELEON Reference Framework [31], it has the advantage that the generated user interface is independent from the later modality or platform.

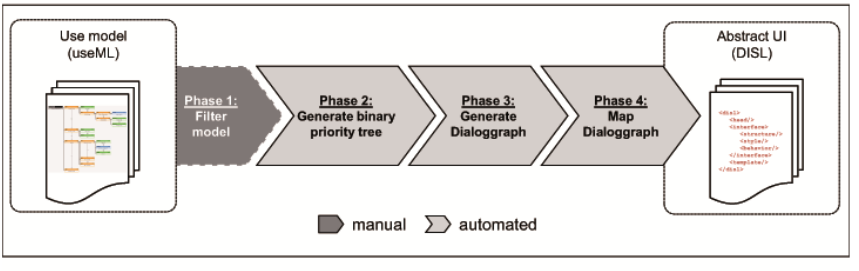

Fig. 9 The Transformation Process

Phase 1: Filtering the Use Model

In the optional first phase, developers can annotate the use model by applying assertions to the single UOs and eUOs. These assertions can be used to specify, for example, on which device a task can be executed or which user group is allowed to execute the task. After the developer has annotated the use model, filters can be set to generate the "system task-model" [26]. Whereas a standard set of assertions is specified in a separate XML-schema in useML, this schema can be individually extended with project-specific assertions.

Phase 2: Generating the Binary Priority Tree

The filtered use model is passed to the second process phase as input for the subsequent automated transformation steps. To simplify interpretation and to solve ambiguities between the temporal operators in this phase, the use model is transformed into a binary "priority tree" [25] representation. While in [25] the priority tree is used for grouping tasks with identical temporal operator priority on the same hierarchical level, a binary version of the priority tree has been used. The binary version of the priority tree has exactly two UOs – respectively eUOs – on each level of the tree. This significantly reduces the number of cases that have to be considered when generating the dialog graph in the next phase. In the left part of Fig. 10, a binary priority tree for a simple "pump" use model is depicted.

The hierarchical structure of the binary priority tree is derived from the temporal operator priorities. A recursive algorithm starts at the root level of the use model and selects those UOs that have the temporal operator with the highest priority. These two UOs are grouped with a new "abstract UO" that replaces both UOs. After that, the algorithm loops until only two UOs are left on the current level. Then the algorithm recursively descends into the next hierarchy level and starts grouping the children. The algorithm terminates when only two UOs/eUOs are left on each hierarchy level.

Phase 3: Generating the Dialog Graph

A dialog graph is generated based on the binary priority tree in the third phase of the transformation process. The dialog graph represents the dynamical character of the use model derived from the semantics of the temporal operators.

eUOs that can be executed by the user at the same point in time are grouped within the states of the dialog graph. Consequently, a state of the dialog graph represents an "Enabled Task Set" (ETS) [23]. The right part of Fig. 10 shows the corresponding dialog graph for the previously mentioned binary priority tree of a pump.

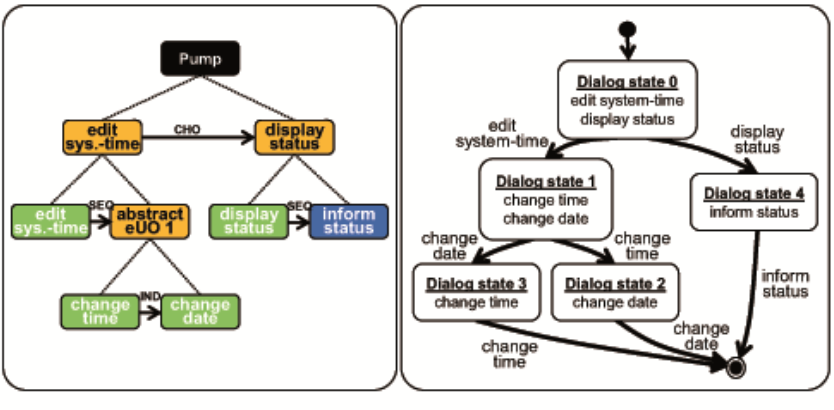

Fig. 10 The binary priority tree (left) and the generated dialog graph (right) of a simple pump use model

For each eUO of a dialog state, there is a corresponding directed transition that is labeled with the eUO and connected to a successor state. These transitions are used to describe the navigation between the single dialog states. The user can navigate through the dialog graph by executing one of the eUOs of a dialog state.

In [9], an algorithm is introduced that has been used in the DYGIMES framework to generate a dialog graph from a CTT task model. Since this algorithm has some shortcomings regarding the generation of parallel states, a new algorithm has been developed that solves these shortcomings and allows the parallel identification of successor states and transitions. The developed recursive algorithm generates the complete dialog graph by *virtually executing* the use model. For this execution, each UO and eUO is flagged with an *execution status* that denotes whether the object is currently executable or has already been executed.

The algorithm is divided into two subsequent phases: *Top-down analysis* for identifying the current dialog state and *bottom-up updating* for determining the new use model execution status.

In the top-down analysis, the use model is searched for executable eUOs. For this purpose, the use model is traversed from the root node of the use model to the leaves, which are represented by the eUOs. The semantics of the temporal operators and the execution status of the UOs are used to decide which branch of the binary priority tree has to be descended recursively. The algorithm terminates when the leaves of the use model have been reached and the executable eUOs have been identified. The result of one top-down-analysis is a unique dialog state that represents one ETS.

Following the top-down analysis, for each eUO stored in the identified dialog state, the successor dialog state as well as the transition to the successor dialog state has to be generated. This is where the identified eUOs are “virtually executed”. Each eUO of the previously identified dialog state is selected and labeled by the algorithm as “executed”. When the execution status of the selected eUO has been changed in the use model, the execution status of all other UOs/eUOs has to be updated. Beginning with the parent UO of the executed eUO, the tree nodes are recursively updated from the leaves up to the root of the use model. This is why this recursive algorithm is referred to as bottom-up updating.

To generate the whole dialog graph, these two recursive algorithms are nested within each other. When all eUOs in the binary priority tree are marked as “executed”, the dialog graph has been generated.

Phase 4: Mapping the Dialog Graph

The final phase of the transformation process implements the mapping from the generated dialog graph onto a dialog model.

In contrast to the TERESA approach, a modality-independent target mapping language has been used for this mapping to support the generation of multi-modality and multi-platform user interfaces. For the specification of this abstract user interface we use DISL. Since DISL was initially designed for mobile devices, it supports a concept where the user interface is split into several modular interfaces. This concept is used in our transformation process for the presentation mapping. Here, the states of the dialog graph are mapped onto DISL interfaces.

Afterwards, each eUO of a dialog state is mapped onto its corresponding abstract interaction object. In Fig. 11, a mapping of the previously generated dialog graph of a pump is depicted.

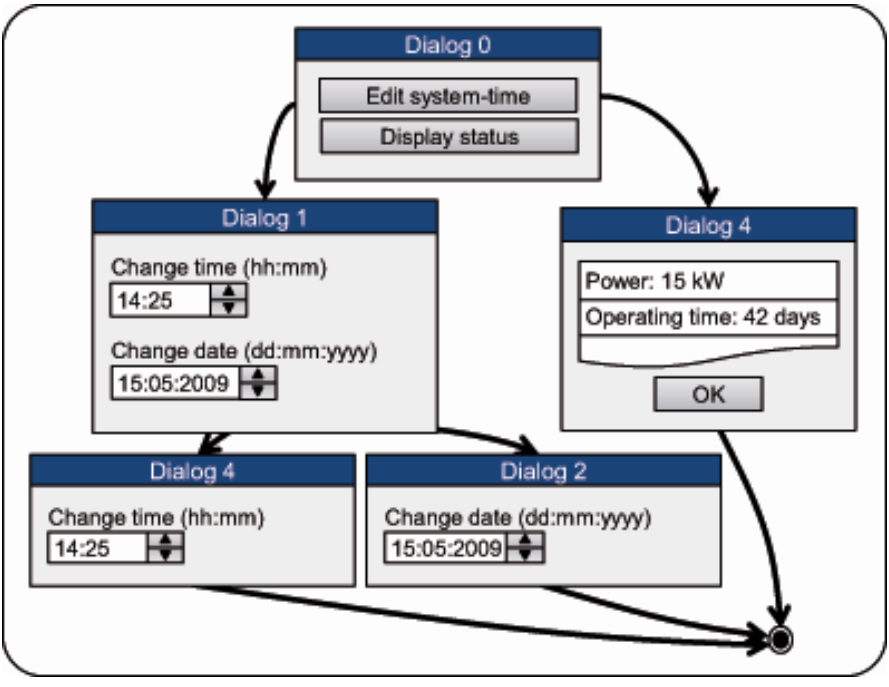

Fig. 11 The mapped dialog graph rendered as a GUI

Since the transitions of the dialog graph are used to move between the dialog states, they represent a dynamical aspect of the user interface. Therefore, the transitions are represented in the behavior part of the DISL user interface. By using a concept of rules and transitions for each AIO, a transition is specified that is fired when the user interacts with this AIO. This transition executes a "restructure" command, which triggers the DISL renderer to activate the new interface.

3.8 Transformation of DISL into UIML

According to the CAMELEON Reference Framework, the next step in a model-based user interface development process is the transformation of the abstract user interface into a modality-depended but platform-independent concrete user interface. Since UIML can be used to describe the user interface in terms of a platform-independent model, the language is used as the target mapping language. Therefore, in this step, the dialogs and interaction objects, as well as the behavior of the abstract UI have to be mapped onto the corresponding UIML elements.

In both languages (DISL and UIML), the user interface is separated into *structure*, *style,* and *behavior* descriptions. Starting by analyzing the user interface structure, the widgets that have to be mapped are identified in the first transformation step. This mapping is specified in a look-up table that expresses the relationship between the abstract DISL interaction objects and the concrete UIML widgets. One mapping can be, for example, that an abstract DISL *command* interaction object is mapped onto a UIML *button* widget, which is commonly used in graphical user interfaces to trigger a function. In the style section of the DISL and UIML user interfaces, the widget's properties that have an impact on the presentation of the user interface are specified. Therefore, DISL properties such as widget texts, descriptions, and visibility attributes are mapped onto the corresponding UIML properties.

The behavior section is primarily used to specify the actions that have to be executed when the user interacts with the user interface. These user interactions may result in opening a new window or changing a set of interaction objects. In DISL, the user interface behavior is described with a set of *rules*, which allow the specification of conditions, and a set of *transitions*, which specify a set of actions that are executed when the according rule is evaluated as true. Additionally, DISL supports the definition of events that allow the specification of time-triggered transitions. As in DISL, the behavior description in UIML is also specified using rules, conditions, and, accordingly, the actions to be executed when a condition is evaluated to true. Because of this similarity, the behavior can be mapped rather statically between those languages.

While those three categories are used in both languages to describe the UI, the languages have different characteristics that have an influence on the transformation process:

One aspect with an impact on the transformation is that an explicit classification of the interaction objects is used in DISL. The abstract interaction objects are classified as *output*, *interaction,* and *collection interaction objects*. While these classes have an impact on the interaction objects semantics, they are not expressed as an explicit property in the interaction objects style definition. Since UIML does not use such a widget classification, those properties have to be expressed as explicit properties in the UI style definition. For example, the *textfield* interaction object is an output element used for presenting large non-editable texts to the user [28]. Since there is no dedicated output-only widget in UIML, this property has to be expressed in the widget *style* definition by specifying the *editable* attribute of the respective *text* widget.

Another aspect that has to be considered in the mapping process is that the properties specified in the style description also have an influence on the mapping of the widgets. For example, in DISL there is an *incremental* property in the style definition that is used to specify a *variablebox* that accepts numerical input values. If this property is set, the variable box has to be mapped onto a *spinner* widget in UIML, while otherwise a *text* widget is used during the structure mapping phase.

The third aspect that has an influence on the transformation is that besides the explicit properties in the style section of the DISL document, additional information can be used in the mapping process. In DISL, some widget properties are

specified in the behavior section. For example, DISL does not support the specification of a minimum or maximum value for a variable box that is set to be incrementally changeable. To emulate these boundaries, they have to be specified in the behavior section of the DISL document by using two variables and a set of rules and transitions that ensure that the values cannot exceed the limits. While this behavior can be mapped onto a UIML style property, the rules and transitions must be omitted when mapping the behavior section of the two languages.

While this phase of the transformation process benefits from the similar structure of the AUI and the CUI models, ambiguities in the mapping process may occur. These ambiguities stem from the fact that the abstract user interface model has to be enriched with additional, modality-specific information not contained in the source model. Since UIML is used as the target language for describing the CUI, attributes for the graphical user interfaces – such as widget size, layout, and color – have to be specified in this mapping that are, by definition, not contained in the AUI model.

Various strategies can be used to obtain this information and to use it as input in the transformation process. One approach is to derive the information from other models. Platform models [30] are used to specify information about the input and output capabilities of the target device and are therefore suitable as an input source. But while it has been shown that these models are suitable for partially deriving the layout [9], they are still not sufficient for automatically deriving a high fidelity user interface. Aspects such as the corporate identity of a user interface usually still have to be applied by the developer herself. Therefore, tools are needed in this transformation phase that allow manual intervention in the transformation phases.

Today, different languages – e.g., ATL [50], RDL/TT [51], and XSLT [52] – and tools are used for model transformation. A review of several transformation systems can be found in [53]. Since the models used in these transformation processes are based on XML, XML-based transformation languages and tools are needed. XSL Transformation (XSLT) is standardized by the W3C group and one of the most widespread languages for transforming XML-based documents. XSLT uses templates to match and transform XML documents into other XML structures, respectively output formats. Since XSLT is one of the most popular XML based transformation languages and offers a large set of generic transformation processors, it is used for the implementation of the DISL-to-UIML transformation process. To enable convenient handling of the XSLT templates, an additional tool has been built that allows selecting the input model (DISL), the transformation template (XSLT), and the output folder for transforming DISL into UIML. Following the generation of the CUI, the source code for the final user interface can be generated, or the user interface can be interpreted by a renderer. In order to enable early feedback of the resulting user interface, an UIML renderer has been integrated into the transformation tool DISL2UIML. Fig. 12 shows a screenshot of the DISL2UIML tool. In the left part, a developer can choose a DISL document and the transformation template (XSLT). After starting the transformation, the resulting UIML document is visualized in the right part of DISL2UIML. In the next step, UIML.net can be started for rendering the transformed UIML document.

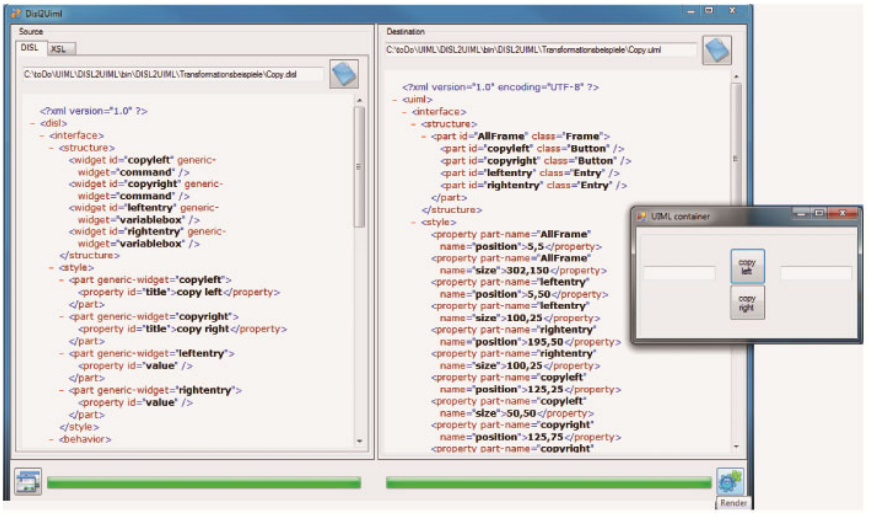

Fig. 12 Screenshot of DISL2UIML showing the transformation software as well as the rendered UIML document

This renderer is an enhanced and extended version of UIML.net [54] capable of rendering UIML 4.0 compliant user interfaces. Different vocabularies such as Gtk#, System.Windows.Forms and System.Windows.Forms for the Compact .Net Framework can be used to present the user interface with a different look and feel.

Besides offering tools for model transformation, tools for authoring and enhancing models are crucial. In recent years, a set of UIML authoring environments have been introduced that allow the design of multi-platform user interfaces. Gummy [55] is an authoring environment that supports the graphical editing of UIML 3.0 documents by offering a toolbox with a set of predefined widgets. After selecting the target platform, the tool loads the appropriate UIML widgets that are available on that platform. Jelly [56] has been recently introduced as a tool for designing multi-platform user interfaces. Although Jelly relies on a proprietary UI description language, it adapts the UIML structure. Harmonia Inc. LiquidApps [57] is the only known commercial UIML authoring environment that supports editing UIML 3.0 compliant documents. The tool has a graphical WYSIWYG editor for designing the user interface using drag&drop functionality. Besides adding and aligning the widgets, the behavior of the user interface can be specified in a different view. After the user interface has been specified, the code for the final user interface can be automatically generated by the tool for several target languages, e.g., Java, C++ Qt, and Web apps.

After generating the code for the final user interface, language-specific tools can be used by the developer to incorporate final design decisions and compile the user interface.

4 Summary and Outlook

In this chapter, we have presented the current status of our model-based user interface development environment. After discussing related work, we introduced the CAMELEON Reference Framework as a meta-architecture for model-based user interface development and subsequently presented the model-based architecture we derived from it. Furthermore, we introduced useML 2.0 and the graphical useML editor "Udit", which is targeted at the structuring phase of the Useware Engineering Process. Additionally, we introduced the Dialog and Interface Specification Language (DISL) and the User Interface Markup Language (UIML) as well as mapping processes from useML to DISL and from DISL to UIML.

Currently, we are investigating efforts to optimize the transformation approach from DISL to UIML by integrating platform models. For specifying platform models, we have analyzed the User Agent Profile (UAProf), which is a specification for capturing the capabilities of a mobile phone, including, e.g., screen size, multimedia capabilities, and character set support. Information about input and output capabilities, in particular, is relevant for transforming the abstract user interface into a concrete user interface. During this transformation process, information about target constraints is essential.

Furthermore, existing tools have to be extended and new tools have to be developed. Especially tools for tweaking the transformation process of useML to DISL have to be developed in conjunction with new DISL renderers, e.g., for the Apple iPhone or for the Google Android platform.

References

[1] Shneiderman, B.: Designing the user interface - strategies for effective human-computer interaction. Addison-Wesley, Boston (2005)

[2] Zuehlke, D., Wahl, M.: Hardware, Software – Useware. Elektronik 23, 54–62 (1999)

[3] Myers, B., Rosson, M.B.: Survey on User Interface Programming. In: Proc. of the 10thAnnual CHI Conference on Human Factors in Computing Systems, pp. 195–202 (1992)

[4] Petrasch, R.: Model Based User Interface Design: Model Driven Architecture und HCI Patterns. GI Softwaretechnik-Trends 27(3), 5–10 (2007)

[5] Zuehlke, D., Thiels, N.: Useware engineering: a methodology for the development of user-friendly interfaces. Library Hi Tech 26(1), 126–140 (2008)

[6] Meixner, G., et al.: Raising the Efficiency of the Use Context Analysis in Useware Engineering by Employing a Support Tool. In: Lee, S., Choo, H., Ha, S., Shin, I.C. (eds.) APCHI 2008. LNCS, vol. 5068, Springer, Heidelberg (2008)

[7] Meixner, G., et al.: Udit – A Graphical Editor For Task Models. In: Proc. of the 4thInternational Workshop on Model-Driven Development of Advanced User Interfaces (MDDAUI). CEUR Workshop Proceedings, Sanibel Island, USA, vol. 439 (2009)

[8] Boedcher, A.: Methodische Nutzungskontextanalyse als Grundlage eines strukturierten USEWARE-Engineering-Prozesses, Fortschritt-Berichte pak 14, Technische Universität Kaiserslautern, Kaiserslautern (2007)

[9] Luyten, K.: Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development. PhD thesis, TransnationaleUniversiteit Limburg (2004)
[10] Pribeanu, C., Vanderdonckt, J.: Exploring Design Heuristics for User Interface Derivation from Task and Domain Models. In: Proc. of the 4th International Conference on Computer-Aided Design of User Interfaces, pp. 103–110 (2002)
[11] Puerta, A.: TheMecano Project: Enabling User-Task Automation During Interface Development. In: Proc. of the Spring Symposium on Acquisition, Learning and Demonstration: Automating Tasks for Users, pp. 117–121 (1996)
[12] Vanderdonckt, J., Bodart, F.: Encapsulating Knowledge for Intelligent Automatic Interaction Objects Selection. In: Proc. of the 1st Annual CHI Conference on Human Factors in Computing Systems, pp. 424–429 (1993)
[13] Bodart, F., et al.: Key Activities for a Development Methodology of Interactive Applications. In: Benyon, D., Palanque, P. (eds.) Critical Issues in User Interface Systems Engineering, pp. 109–134. Springer, Heidelberg (1995)
[14] Schlungbaum, E.: Knowledge-based Support of Task-based User Interface Design in TADEUS. In: Proc. of the 16th Annual CHI Conference on Human Factors in Computing Systems (1998)
[15] Stary, C.: TADEUS: seamless development of task-based and user-oriented interfaces. IEEE Transactions on Systems, Man, and Cybernetics 30(5), 509–525 (2000)
[16] Ricard, E., Buisine, A.: Des tâches utilisateur au dialogue homme-machine: GLADIS++, une demarche industrielle. In: Proc. of Huitièmes Journées-surl'Ingénierie de l'Interaction Homme-Machine (1996)
[17] Markopoulos, P., et al.: On the composition of interactor specifications. In: Proc. of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface, London (1996)
[18] Abed, M., et al.: Using Formal Specification Techniques for the Modeling of Tasks and the Generation of Human-Computer User Interface Specifications. In: Diaper, D., Stanton, N. (eds.) The Handbook of Task Analysis for Human-Computer Interaction, pp. 503–529. Lawrence Erlbaum Associates, Mahwah (2003)
[19] Puerta, A.: A Model-Based Interface Development Environment. IEEE Software 14(4), 40–47 (1997)
[20] Clerckx, T., Coninx, K.: Integrating Task Models in Automatic User Interface Generation. EDM/LUC Diepenbeek, Technical Report TR-LUC-EDM-0302 (2003)
[21] Mori, G., et al.: Tool Support for Designing Nomadic Applications. In: Proc. of the 8th International Conference on Intelligent User Interfaces, pp. 141–148 (2003)
[22] Paternò, F., et al.: Authoring pervasive multi-modal user interfaces. International Journal on Web Engineering and Technology 4(2), 235–261 (2008)
[23] Paternò, F.: Model-based design and evaluation of interactive applications. Springer, London (1999)
[24] Mori, G., et al.: Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions. IEEE Transactions on Software Engineering 30(8), 507–520 (2004)
[25] Luyten, K., et al.: Derivation of a Dialog Model from a Task Model by Activity Chain Extraction. In: Proc. of the 10th International Workshop on Interactive Systems: Design, Specification and Verification (2003)

[26] Berti, S., et al.: The TERESA XML language for the Description of Interactive Systems at Multiple Abstraction Levels. In: Proc. of the Workshop on Developing User Interfaces with XML: Advances on User Interface Description Languages, pp. 103–110 (2004)
[27] Limbourg, Q., Vanderdonckt, J.: Addressing the Mapping Problem in User Interface Design with UsiXML. In: Proc. of the 3rd International Workshop on Task Models and Diagrams for User Interface Design (2004)
[28] Schaefer, R.: Model-Based Development of Multimodal and Multi-Device User Interfaces in Context-Aware Environments. C-LAB Publication, 25. Shaker Verlag, Aachen (2007)
[29] Paternò, F., et al.: MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Transactions on Computer-Human Interaction (TOCHI) 16(4), 1–30 (2009)
[30] Puerta, A., Eisenstein, J.: Towards a General Computational Framework for Model-Based Interface Development Systems. In: Proc. of the 4th International Conference on Intelligent User Interfaces (1999)
[31] Calvary, G., et al.: A Unifying Reference Framework for Multi-Target User Interfaces. Interacting with Computers 15(3), 289–308 (2003)
[32] Cantera Fonseca, J.M., et al.: Model-Based UI XG Final Report, W3C Incubator Group Report, May 4 (2010), http://www.w3.org/2005/Incubator/model-based-ui/XGR-mbui-20100504/ (accessed June 1, 2010)
[33] Ali, M.F., et al.: Building Multiplatform User Interfaces With UIML. In: Seffah, A., Javahery, H. (eds.) Multiple User Interfaces – Cross-Platform Applications and Context-Aware Interfaces, pp. 95–118. John Wiley & Sons, Chichester (2004)
[34] Mukasa, K., Reuther, A.: The Useware Markup Language (useML) - Development of User-Centered Interface Using XML. In: Proc. of the 9th IFAC Symposium on Analysis, Design and Evaluation of Human-Machine-Systems, Atlanta, USA (2004)
[35] Meixner, G., Goerlich, D.: Eine Taxonomie für Aufgabenmodelle. In: Proc. of Software Engineering 2009, Kaiserslautern, Germany. LNI P, vol. 143, pp. 171–177 (2009)
[36] Bomsdorf, B., Szwillus, G.: From task to dialogue: Task based user interface design. SIGCHI Bulletin 30(4), 40–42 (1998)
[37] Meixner, G.: Entwicklung einer modellbasierten Architektur für multimodale Benutzungsschnittstellen, Fortschritt-Berichte pak 21, Technische Universität Kaiserslautern, Kaiserslautern (2010)
[38] Uhr, H.: TOMBOLA: Simulation and User-Specific Presentation of Executable Task Models. In: Proc. of the International HCI Conference, pp. 263–267 (2003)
[39] Biere, M., et al.: Specification and Simulation of Task Models with VTMB. In: Proc. of the 17th Annual CHI Conference on Human Factors in Computing Systems, pp. 1–2. ACM Press, New York (1999)
[40] Gray, P., et al.: XUAN: Enhancing UAN to capture temporal relationships among actions. In: Proc. of the Conference on People and Computers, vol. IX, pp. 301–312. Cambridge University Press, Cambridge (1994)
[41] Scapin, D., Pierret-Golbreich, C.: Towards a method for task description: MAD. In: Proc. of the Conference Work with Display Units, pp. 27–34 (1989)
[42] Tarby, J.C., Barthet, M.F.: The Diane+ method. In: Proc. of the 2nd International Conference on Computer-Aided Design of User Interfaces, pp. 95–120 (1996)

[43] Van Der Veer, G., et al.: GTA: Groupware task analysis – modeling complexity. Acta Psychologica 91, 297–322 (1996)
[44] Mueller, W., et al.: Interactive Multimodal User Interfaces for Mobile Devices. In: Proc. of the 37th Annual Hawaii International Conference on System Sciences, Hawaii, USA (2004)
[45] Abrams, M., et al.: UIML: An Appliance-Independent XML User Interface Language. In: Proc. of the 8th International World Wide Web Conference, Toronto, Canada, pp. 1695–1708 (1999)
[46] Curry, M.B., Monk, A.F.: Dialogue Modeling of Graphical User Interfaces with a Production System. Behaviour and Information Technology 14(1), 41–55 (1995)
[47] Helms, J., et al.: User Interface Markup Language (UIML) Version 4.0. (2009), http://docs.oasis-open.org/uiml/v4.0/cd01/uiml-4.0-cd01.pdf
[48] Ali, M.F., et al.: Building Multi-Platform User Interfaces with UIML. In: Proc. of the 4th International Conference on Computer-Aided Design of User Interfaces, pp. 255–266 (2002)
[49] Seißler, M., Meixner, G.: Entwicklung eines Transformationsprozesses zur modellbasierten Entwicklung von multimodalen Benutzungsschnittstellen. In: Proc. of the 8th Berliner Werkstatt Mensch-Maschine-Systeme, Berlin, Germany (2009)
[50] Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
[51] Schaefer, R., et al.: RDL/TT - A Description Language for the Profile-Dependent Transcoding of XML Documents. In: Proc. of the International ITEA Workshop on Virtual Home Environments (2002)
[52] W3C Consortium, XSL Transformations (XSLT) Version 1.0. W3C Recommendation, November 16 (1999)
[53] Schaefer, R.: A survey on transformation tools for model based user interface development. In: Proc. of the 12th International Conference on Human-Computer Interaction: Interaction Design and Usability, Beijing, China, pp. 1178–1187. Springer, Heidelberg (2007)
[54] Luyten, K., Coninx, K.: UIML.Net: an Open UIML Renderer for the.Net Framework. In: Jacob, R.J.K., Limbourg, Q., Vanderdonckt, J. (eds.) Computer-Aided Design of User Interfaces, vol. IV, pp. 259–270. Kluwer Academic Publishers, Dordrecht (2006)
[55] Meskens, J., et al.: Gummy for multi-platform user interface designs: shape me, multiply me, fix me, use me. In: Proc. of the Working Conference on Advanced Visual Interfaces 2008, Napoli, Italy, pp. 233–240. ACM, New York (2008)
[56] Meskens, J., et al.: Jelly: A Multi-Device Design Environment for Managing Consistency Across Devices. In: Proc. of the Working Conference on Advanced Visual Interfaces 2010, Rome, Italy. ACM, New York (2010)
[57] Harmonia Inc. LiquidApps® - A Powerful Enterprise Mashup Solution, http://www.liquidappsworld.com/ (accessed June 1, 2010)