Executable Models for Human-Computer Interaction

Marco Blumendorf, Grzegorz Lehmann, Sebastian Feuerstack, Sahin Albayrak
DAI-Labor, TU-Berlin
Ernst-Reuter-Platz 7, D-10587 Berlin
firstname.lastname@DAI-Labor.de

Abstract. Model-based user interface development is grounded on the idea to utilize models at design time to derive user interfaces from the modeled information. There is however an increasing demand for user interfaces that adapt to the context of use at runtime. The shift from design time to runtime means, that different design decisions are postponed until runtime. Utilizing user interface models at runtime provides a possibility to utilize the same basis of information for these postponed decisions. The approach we are following goes even one step further. Instead of only postponing several design decisions, we aim at the utilization of stateful and executable models at runtime to completely express the user interaction and the user interface logic in a model-based way.

Keywords: human-computer interaction**,** model-based user interfaces, runtime interpretation

1. Introduction

Model-based software development is becoming more and more popular these days and has been identified as suitable to deal with the increasing complexity of software systems developers have to cope with. While UML made the idea of modeling popular by providing a common language to exchange concepts between developers, the Meta-Object Facility (MOF) and the Model-Driven Architecture (MDA) of the Object Management Group (OMG) provide the key concepts for the widespread utilization of model-based software engineering. However, with the advent of technologies like UML Actions or the Business Process Modeling Language (BPML) the focus of the modeling approaches shifts from static systems to dynamic systems and executable models. 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 [4]). Executable models support this approach by providing the logic that defines the dynamic behavior as part of the model. Their structure will be explained in more detail in the remainder of this paper.

The ability to model complex software systems has recently also regained more attention as a technology capable of handling the increasing complexity of user interfaces (UIs). Rising demands for dynamic UIs that adapt to the context-of-use and

thus user preferences, multiple devices, the surrounding environment or even multiple modalities, induce the need for new ways to express such characteristics. Model-based approaches as described in [14, 11] address these challenges by utilizing models to support the user interface development process and provide the means to derive multiple consistent user interfaces from a (sometimes multi-level) UI model. Additionally approaches that utilize UI models at runtime [7, 10] addressed specific development issues. There is however still the lack of a well accepted common User Interface Description Language (UIDL) as different approaches focus on different aspects. UsiXML currently seems to be the most feasible candidate for such a language.

In this paper we present an approach that facilitates the development of User Interface Management Systems that address:

- supporting different UIDLs and models by introducing a common meta-layer
- the consideration of the predictive as well as the effective context of use [5]
- the specification of syntax and semantics as part of a model
- support for the easy extension of systems based on the coupling of multiple models
- the unification of design models and the runtime data structures of interactive systems

The model-based approach we describe in the following therefore facilitates the utilization of "executable" user interface models at runtime. Although we propose a set of models, the general system allows the utilization of various UIDLs on different levels of abstraction. The approach therefore addresses the definition of a meta-meta-model providing building blocks for meta-models that also contain the model semantics. Furthermore the system allows the developer to monitor, maintain, manipulate and extend interactive applications at runtime and thus manage the continuously changing requirements of user interface development.

After introducing the current state of the art in the next section, we give an introduction to the idea of executable models, providing the possibility to combine syntax and semantic with state information to support direct model execution. Next we present a meta-meta-model, distinguishing definition-, situation- and execution parts our executable models are comprised of. Following that section, we give an overview of the meta-models and the mapping meta-model we utilize for the UI development and the underlying concepts. We then introduce the architecture of our runtime system and elaborate on the possible applications of the approach. We describe how the development process can be supported by the ability to directly modify the models at runtime using Eclipse and EMF, which also allows runtime inspection, modification and debugging of the models.

2. State of The Art

The recent shift towards model-based software development aims at solutions to cope with the increasing complexity of current and future systems. While UML made the idea of modeling popular by providing a common language to exchange concepts between developers, MOF and MDA provide the key concepts for the utilization of

model-based software engineering. Technologies like Executable UML, UML Actions or BPML focus on the shift from static- to dynamic systems and executable models. These developments also influence user interface research. The current state of the art in model-based user interface development shows the need for a common language [11] and a tendency towards a common understanding of the new challenges and approaches [2, 1]. However, there are also approaches to build architectures, tools and methodologies to support the designer during the development as well as the creation of adaptive user interfaces and their adaptation at runtime. [10] for example deals with the execution of CTT-based user interface models and [7] presents a runtime system that targets the creation of context-aware user interfaces.

Sottet et al. [19] propose keeping the models alive at runtime to make the design rationale available. This means, that the final UI code should not be generated at design-time, but at runtime, taking the context adaptations into account. Demeure et al. [8] presented the Comets, which are prototypical user interface components capable of adaptations due to the application of models at runtime. Preserving the models at runtime opened the possibility for the implementation of plasticity-enabling features like their Meta-UI. Yet, the black-box nature of the Comets seems problematic at runtime, as the system has no indications about a Comet's inner state. Clerckx et al. [6] extend the DynaMo-AID design process by context data evaluated at runtime, supporting UI migration and distribution. Their approach allows the designer to define context-dependent information in the models. However, although the models are then interpreted dynamically, their adaptation at runtime is not possible. To support the linking of multiple models, Sottet et al. [20] propose to model transformations which should also be available at runtime. However, none of the solutions we are aware of enables to identify the common components of multiple models and links between the models, which could pave the road to interoperability between different UIDLs. In our approach, we utilize executable models to derive user interfaces at runtime. We define a meta-meta-model and conceptually introduce a mapping meta-model. This allows us to connect different models and concepts to build advanced user interfaces.

3. Executable Models

Recent developments in the model-based user interface development community show the increasing importance of models as a basis for development support and also as basis at runtime. Currently there is still a focus on the usage of static models, providing (only) a snapshot of the system under study at a given point in time. Research in model-driven engineering of user interfaces has brought up various approaches to use models for the derivation of user interfaces for different purposes. However, future interactive systems are required to adapt to different contexts at runtime and thus deriving multiple UIs at design time does not seem to be feasible anymore. Keeping the model(s) at runtime allows postponing design decisions to runtime and thus performing adaptations to the runtime circumstances rather than predicting all possible context situations at design time. We think that the executable models approach introduced in this section can support a more extensive usage of

models at runtime. In contrast to common static models, executable models provide the logic that defines the dynamic behavior as part of the model, which makes them complete in the sense that they have "everything required to produce a desired functionality of a single problem domain" [12]. 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. We can thus describe dynamic executable models as *models that provide a complete view of the system under study over time*.

3.1 A Meta-Meta-Model

Combining the initial state of the system, the dynamic model elements that change over time and the processing logic in one model, leads to the need to clearly distinguish between the different elements. We thus distinguish between definition-, situation- and execution elements in the following. A similar classification has also been identified by Breton and Bézivin [4].

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 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.

Execution Elements define the interpretation process of the model, in other words the transitions from one state to another. In this sense execution elements are procedures or actions altering the situation elements of a model. Execution 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.

Distinguishing these elements leads to the meta-meta-model of dynamic executable models depicted in Figure 1. The meta-meta-model provides a more formal view of executable models and summarizes the common concepts the models are based on. It is positioned at M3 layer in the MOF Metadata Architecture [15] (see also Figure 4). The clear separation of the elements provides clear boundaries for the designer, only working with the definition elements and the system architect, providing the meta-models. A definition element as the basic element finally aggregates situation- and execution elements that describe and change situations for a given definition element. Using such models in a prescriptive way (constructive rather than descriptive

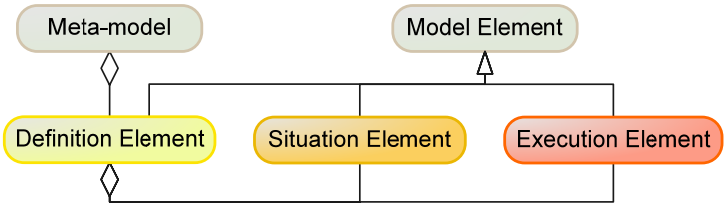

Figure 1: Meta-Meta-Model of Dynamic Executable Models

modeling) allows defining systems that evolve over time, reason about the past and predict future behavior. Dynamic models are often used to build self-adaptive applications, as for example Rohr et al. [17] describe. In this context, the role of the models is often that of monitoring the system. In the following we illustrate the implications of the meta-meta-model by introducing the realization of a CTT-based task-meta-model as executable meta-model using the Eclipse Modeling Framework.

3.2 Modeling with EMF

For our current implementation we have utilized the Eclipse Modeling Framework (EMF), which is a modeling and code generation framework integrated into the Eclipse IDE. EMF provides means to define meta-models, create models and appropriate editors. Beyond that, for each meta-model EMF is capable of generating Java class structures representing it. These can then be enriched by a programmer just as usual Java code can. This way it is possible to add execution logic into the meta-model in form of Java code fragments.

ECore is the meta-model of EMF and thus the meta-meta-model of all models defined in EMF. It resides on the same layer as the meta-meta-model of the executable models. Choosing EMF as the implementation technology makes it necessary to map definition-, situation- and execution elements - the entities of our meta-meta-model - to entities in the ECore meta-meta-model. In our approach, the definition elements are represented by EClasses in ECore. 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. Finally, the execution elements are in ECore expressed as EOperations, which allows adding execution logic into a meta-model in form of Java code fragments. In Java the execution logic is defined within methods and these are represented by EOperations within ECore.

3.3 Executable Task Models

In the following we use the task model as an example to illustrate the executable models, the usage of the meta-meta-model and the realization with EMF. The task model we use is based on the CTT notation which is well known in model-based UI development. Task models are also known to be executable [10] and define the tasks

the user has to accomplish and their temporal relations. They thus provide an overview of the workflow of the application. To be able to utilize the CTT-based task model for our purposes we extended the static part of the CTT meta-model with the state information needed to reflect the state of the execution in the model. We introduce attributes for each task, identifying the state of the task and thus the situation elements of the model.

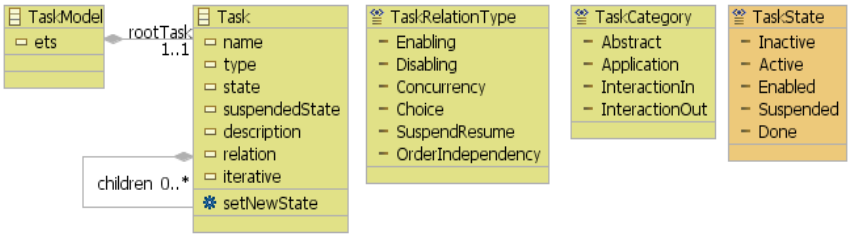

Figure 2: The Task-Meta-Model in EMF

Figure 2 shows the EMF meta-model structure for task models. As one can see in the graphic, every task model is comprised of a root task with a set of children 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. During execution at runtime - 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). This allows to explicitly store the execution state of the model as part of the model. During execution the Enabled Task Set (ETS) is derived and then each task in this set is set to state “enabled”. Once the task is completed it is set to “done”. Using this interpretation we distinguish InteractionIn (user input) and InteractionOut (system output) and application tasks (backend call without user intervention) to model the workflow of the application.

This example illustrates how the task model and its execution logic can be embedded into a single executable model while keeping design time and runtime information separate, but also making the runtime state of the model explicit.

3.4 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-meta-model of executable models describes the building blocks of such models. We exemplified this principle using executable task models.

Looking at current model-based approaches [2, 11] there is a clear trend to provide multiple models for the different aspects (e.g. levels of abstractions) rather than a single model. We introduce an approach, combining multiple models to create user interfaces at runtime, in more detail in section 5. Such relations between models are

not reflected by the meta-meta-model, as executable models are first of all self-contained to ensure executability. The next section thus introduces a mapping meta-model, that allows to express the relations between multiple models. The model itself is executable as well, and provides the required event hooks in the execution logic to interconnect multiple models. The mapping meta-model is positioned on layer M2 of the MOF architecture [15] (see also Figure 4).

4. Mapping-Meta-Model

The mapping model connects multiple executable models and allows to define relations between their elements based on the structures given by the meta-meta-model. The mappings defined in this model are the glue between the models of our multi-model architecture. The mapping meta-model as well as the other related meta-models is thereby located at M2 layer of the MOF architecture. Providing an extra meta-model solely for mappings also enables to benefit from tool support and removes the problem of mappings hard-coded into the architecture, as has been already advised by Puerta and Eisenstein [16]. The mapping meta-model allows the definition of the common nature of the mappings and helps ensuring extensibility and flexibility. A mapping relates models by relating elements of the models whereas the models are not aware of their relation. An example of a mapping meta-model, consisting of a fixed set of predefined mapping types only, can also be found in UsiXML described by Limbourg [11]. Sottet et al. [18] have defined a mapping meta-model, which can also be used to describe transformations between model elements at runtime. However, in contrast to their approach we put a stronger focus on 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 two or more models. The information provided by the mappings can be used to synchronize elements if the state of the source elements changes. Mellor et al. [13] also 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). Our mapping model contains mappings of the latter kind. Focusing on runtime aspects, we see a mapping as a possibility to alter an existing target model, based on changes that happen to the related source model. In contrast to the most common understanding of mappings the mappings we utilize 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 meta-model is provided in **Figure 3** and combines mapping types and mappings. Mapping types are the main elements of the mapping meta-model, as they provide predefined types of mappings that can be used to define the actual mappings between elements on M1 layer. A mapping type thereby consists of two definition elements as well as of 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

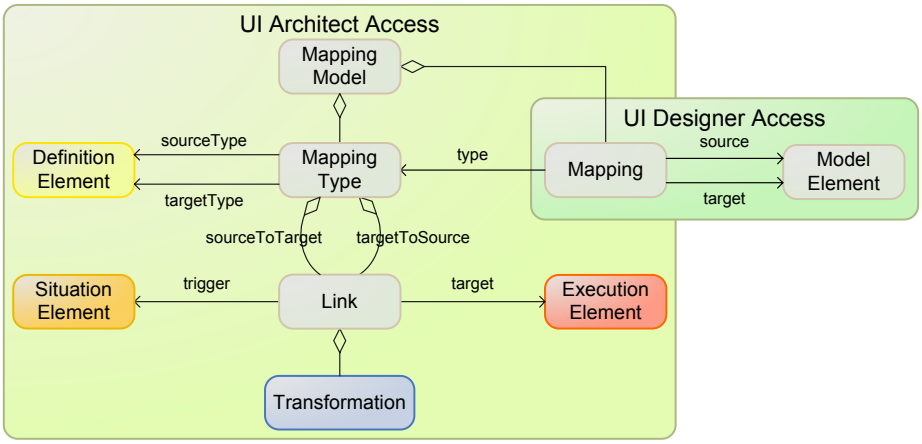

Figure 3: Mapping Meta-Model

situation element, an execution 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 execution logic is thus the logical target of the link. 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 meta-model 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 now provides a set of available mapping types with predefined logic, defined on the meta-model level. Thus to relate two models, the user interface designer extends this initial model by creating new mappings that reference one of the available mapping types. 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 our meta-meta-model we were able to define the mapping meta-model independent from the concrete meta-models 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 meta-models.

4.1 Modeling Mappings with EMF

The EMF implementation of the mappings basically reflects the meta-model illustrated in **Figure 3** and also conforms to the described meta-meta-model 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 us to create standard mappings that refer to the meta-models of the system to design. Once a UI developer creates models according to these meta-models, the pre-defined mappings can directly be used to relate dedicated model elements and thus easily provide the necessary information exchange.

Our implementation of the mapping meta-model is derived from the mapping of our meta-meta-model with the ECore meta-meta-model as introduced in section 3.2. This way it is possible to define mapping types on top of any executable ECore meta-model (M2) used within our architecture. 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. In our implementation we made use of the eventing mechanism provided by EMF in the generated Java code. It enables to register so called adapters to every EObject. 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, its new and previous values. In our prototyping phase we have developed a simple transformation language which we then used to define the transformation elements. Currently we are working on the integration of the ATLAS Transformation Language (ATL)[1] into the mapping meta-model. 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. For this purpose we utilize the reflection mechanisms of the Java language.

5. The Multi-Access Service Platform (v2)

Based on the concepts of executable models and the mappings, we rebuild our previously developed Multi-Access Service Platform (MASP). The MASP is a UIMS that allows the creation of multimodal user interfaces by interpreting models at runtime. We are currently using the system to build adaptive multimodal interfaces for smart home environments as part of the Service Centric Home project[2]. Utilizing executable models as the underlying concepts for the approach lead to a complete redesign of the system. Based on the meta-meta-models and the mapping (meta-) model we selected a set of models to represent the workflow and the interaction with the application as well as context and backend services (Figure 5). The selection and design of the models was also influenced by UsiXML models and the Cameleon reference framework, although we decided to go with a slightly adapted syntax in the first step. Figure 4 shows the components of the MASP in relation to the MOF Meta Pyramid. M1 thereby comprises the loosely coupled models while M2 provides the underlying meta-models. On M2 we also introduced the MASP Core meta-model which provides the means to initially load applications (sets of models) and trigger the execution. The Model contains sessions for the user and application management. Additionally it provides a basic API to access the models, making it easy to build

[1] http://www.eclipse.org/m2m/atl/
[2] www.sercho.de

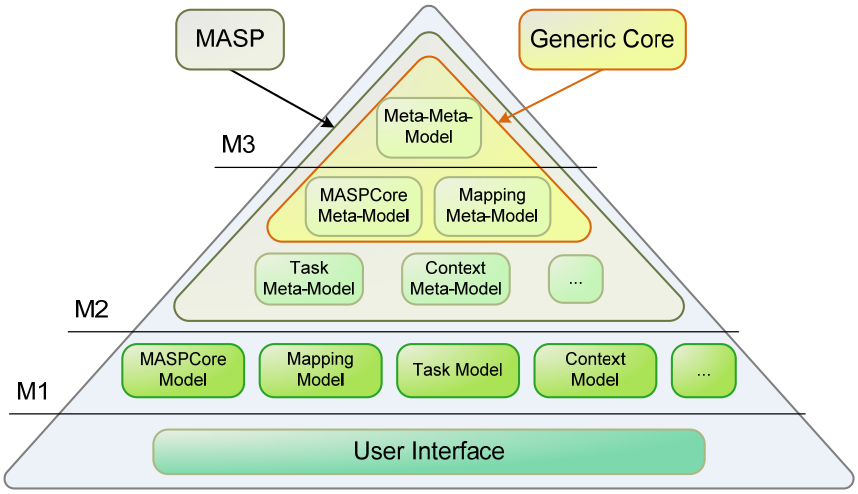

Figure 4: The MASP in Relation to the MOF Meta-Pyramid

software and management tools for the platform. Besides the models and their execution logic, the MASP comprises a channel-based delivery mechanism for the delivery of the created final user interfaces to the interaction devices [3] and integrates several sensors (e.g. an Ubisense ultra wide band localization system) for context recognition.

Figure 5 shows the models we are currently using to develop applications for our approach. The task model defines the temporal relations between the multiple tasks of the application and can thus serve as outline for the interaction. A domain model completes the task model by providing content to the tasks. The model itself on the one hand defines the data structures we are dealing with, but also holds instances of these structures, objects, that become accessible at runtime. The life-time of these objects is determined by the task model again, which also references the objects in the designated tasks [9]. Altering the content of the domain model happens in two ways. On the one hand there are backend services that provide information. These services are on the highest level referenced by the task model in terms of application tasks [9]. A specific description of the service call itself and the referenced objects is provided by a service model. Thus application tasks are mapped to service calls in the service model via the appropriate mappings. The other possibility for new or modified content is the user entering or changing information while interacting with the system. This is realized by the interaction model, related to interaction tasks. Here we distinguish input and output tasks which each identify the interaction on the highest level of abstraction. A reification of the interaction in terms of details is then provided by the interaction model that comprises an abstract interaction description, which is modality independent, and a concrete interaction description, which adds the modality dependent information. Finally, during our work we identified the context model as an important part as soon as the environment, available devices and thus the context of the interaction comes into play. We thus also created a context model, allowing to provide context information. The model is at runtime filled with information delivered by various sensors and allows the creation of mappings that trigger behavior

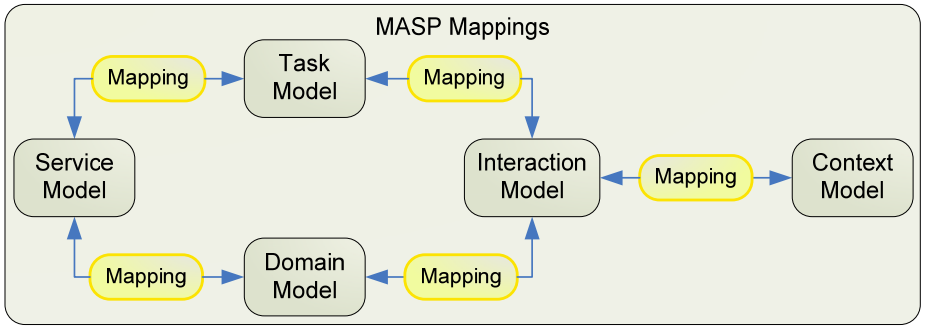

Figure 5: Structure of the runtime system (models and mappings)

or UI adaptations dependent on the context. Finally, our mapping model allows the creation of various mappings between the different parts of the models and thus links all models together. By linking the task model to service and interaction model, the execution of the task model and thus changing task states to "enabled" triggers the activation of service calls and interaction elements. While service calls activate backend functions, active interaction elements are displayed on the screen and allow user interaction. They also incorporate domain model elements in their presentation and allow their manipulation through user input as defined by the mappings. The context model finally also influences the presentation of the interaction elements that are related to context information. Thus, the execution of the task model triggers a chain reaction, leading to the creation of a user interface from the defined user interface model. The structure underlying this approach also opens the possibility to add additional models or change existing models in the future. Although our current approach follows the well accepted Cameleon Reference Framework and thus provides a similar set of models, it provides a meta-layer, allowing to unite multiple modeling languages and approaches.

6. Applications

Utilizing executable models as described in this paper offers various opportunities for future user interface development. We build a couple of prototypes and smaller trials, which showed great potential for issues like context adaptation at runtime, personalization, debugging and hot deployment as well as extensibility of running systems. In the following we report on our results concerning two multimodal applications (a cooking assistant and an energy manager) we (re-)built based on the MASP as well as several smaller proof-of-concept prototypes.

Both applications, the cooking assistant (CA) and the energy manager, target smart home environments and support multimodal interaction. While the CA, we will focus on in the following, runs in the kitchen and supports the user while preparing a meal, the energy manager provides an overview of the energy usage of the home devices and allows to switch devices on and off. The CA is based on three interaction steps. First the user selects a recipe, from recommendations or the results of a search. Afterwards the required ingredients are listed and based on the availability in the

home a shopping list is displayed. Finally the cooking process is guided with step by step instructions. The central model of the CA is the task model, defining the underlying workflow. Based on the task model, related objects have been modeled as domain model and service calls to the backend (e.g. to retrieve the list of recipes or to control kitchen devices) have been defined as service model. Mappings on the one hand relate application tasks to service calls. Thus as soon as an application task becomes active the related service call is executed. On the other hand the domain objects serving as input and output for the service calls are related to these. In a similar way, interaction tasks are related to interaction objects via mappings. Interaction objects thus become activated as soon as an interaction task becomes enabled. This triggers the delivery of the representation of the interaction objects on the interaction device. The interaction devices are thereby identified as part of the usage context and thus the mappings between interaction model and context model provide the foundation for the delivery of the user interface.

In addition to this complete application we also evaluated some additional features in smaller trials. Based on the developed CA, we explored the runtime inspection of the state information of the underlying models as well as extension mechanisms and further capabilities to adapt the UI to the context of use.

Runtime Development – One feature of the Eclipse Modeling Framework underlying our implementation is the possibility to directly connect the models to Java code. We make use of this facility to build an editor that connects to the models of the running system. Thus any changes we make to the model via the editor are directly propagated into the runtime system, as they also trigger the related events. This approach allows to directly inspect and change the running system. As the situation-elements monitor the state of the execution in various details, there is an enormous potential to access and manipulate the complete state of the system. All modeled information is available. This feature simplifies development and debugging a lot, however, in combination with our strictly model-based approach it also allows the customization of the application by the end user if appropriate tools are provided either as additional software or even as part of the application. The loose coupling of the models and the encapsulation of the execution logic as part of the meta-model also allow easily extending or changing the application, even at runtime, which is an important aspect to manage the continuous changes requested from software developers.

Enhancing a Running System – We evaluated the possibilities to enhance (running) systems in another case study, where we replaced one model with another one (conforming to a new meta-model) at runtime. With current task-based approaches we noted that it is rather difficult to model back and forth navigation e.g. between different screens of an application, as dialog modeling is not the responsibility of the task model. Therefore we will transform the task model into a state machine model and enrich it with additional transitions representing the desired dialog navigation. This case study showed that it is possible to replace models of the system without changing the existing models, simply by providing the model and a set of mappings. In the same way the system can also be extended with additional models, which emphasizes the language-spanning aspects of the approach.

7. Summary & Outlook

We presented an approach to utilize dynamic executable models to build user interfaces. Combining definition-, situation- and execution elements provides the means to make all relevant information explicitly accessible and also helps separating the parts of the models relevant for the UI designer. In combination with the mapping model, this approach allows an easy integration of multiple models at runtime to build complex systems. The loose coupling of models also provides a very flexible structure that can easily be extended and adapted to different needs. This also addresses the problem that there are currently no standard or widely accepted UI models. Combined with development and debugging tools this approach allows to inspect and analyze the behavior of the interactive system on a very low level of details. To evaluate the feasibility of the approach to cope with challenges and requirements for the next generation of user interfaces we developed a model-based runtime system for smart home user interfaces. We use task, domain, service and interaction models and mappings between these models at runtime to interpret the modeled information and derive a user interface. As next steps we want to further evaluate the performance of our EMF- and Java-based implementation to optimize the implementation. However, its current implementation shows that the systems perform very well. We also aim at further refining the models we are using. While the combination of different models seems suitable, especially our current interaction model gives room for extensions and enhancements. The possibility to build self-aware systems using executable models is also a fascinating feature that needs further evaluation. Utilizing the models at runtime however, does not solve all problems of model-based user interface development, but it gives possibilities to overcome the technical challenges in addressing these problems.

8. Acknowledgements

We thank the German Federal Ministry of Economics and Technology for supporting our work as part of the Service Centric Home project in the "Next Generation Media" program.

9. References

1. Calvary, G., Coutaz, J., Ganneau, V. Vanderdonckt, J., Demeure, A., Sottet, J.-S. The 4c reference model for distributed user interfaces. In Proc. of 4th IARIA International Conference on Autonomic and Autonomous Systems, 2008.
2. Balme, L., Demeure, A., Barralon, N., Coutaz, J., and Calvary, G. Cameleon-rt: A software architecture reference model for distributed, migratable, and plastic user interfaces. In EUSAI, 2004.
3. Blumendorf, M., Feuerstack, S., and Albayrak, S. Multimodal user interaction in smart environments: Delivering distributed user interfaces. In European Conference on Ambient

Intelligence: Workshop on Model Driven Software Engineering for Ambient Intelligence Applications, 2007.
4. Breton, E. and Bézivin, J.. Towards an understanding of model executability. In FOIS '01: Proc. of the international conference on Formal Ontology in Information Systems, 2001.
5. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., and Vanderdonckt, J. A unifying reference framework for multi-target user interfaces. Interacting with Computers, 15(3), 2003.
6. Clerckx, T., Vandervelpen, C., and Coninx, K. Task-based design and runtime support for multimodal user interface distribution. In Proc. of Engineering Interactive Systems 2007.
7. Coninx, K., Luyten, K., Vandervelpen, C., Van den Bergh, J., and Creemers, B. Dygimes: Dynamically generating interfaces for mobile computing devices and embedded systems. In Luca Chittaro, editor, Mobile HCI, volume 2795 of Lecture Notes in Computer Science, 2003.
8. Demeure, A., Calvary, G., Coutaz, J., and Vanderdonckt, J. The comets inspector: Towards run time plasticity control based on a sematic network. In Proc. of TAMODIA 2006.
9. Feuerstack, S., Blumendorf, B., and Albayrak, S. Prototyping of multimodal interactions for smart environments based on task models. In European Conference on Ambient Intelligence: Workshop on Model Driven Software Engineering for Ambient Intelligence Applications, 2007.
10. Klug, T. and Kangasharju, J. Executable task models. In Proc. of TAMODIA 2005.
11. Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and López-Jaquero, V. Usixml: A language supporting multi-path development of user interfaces. In EHCI/DS-VIS, volume 3425 of Lecture Notes in Computer Science, 2004.
12. Mellor, S. Agile MDA, 2004.
13. Mellor, S., Scott, K., Uhl, A., and Weise, D. MDA Distilled: Principles of Model-Driven Architecture. 2004.
14. Mori, G., Paternò, F., and Santoro, C. Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions. IEEE Trans. Softw. Eng., 30(8), 2004.
15. Object Management Group. Meta Object Facility (MOF) Specification — Version 1.4, April 2002.
16. Puerta, A.R. and Eisenstein, J. Towards a general computational framework for model-based interface development systems. In Intelligent User Interfaces, 1999.
17. Rohr, M., Boskovic, M., Giesecke, S., and Hasselbring, W. Model-driven development of self-managing software systems. In "Models@run.time" at the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML'06) 2006.
18. Sottet, J.-S., Calvary, G., and Favre, J.-M.. Mapping model: A first step to ensure usability for sustaining user interface plasticity. In Model Driven Development of Advanced User Interfaces (MDDAUI 2006), 2006.
19. Sottet, J.-S., Calvary, G., and Favre, J.-M.. Models at runtime for sustaining user interface plasticity. In "Models@run.time" at the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML'06) 2006.
20. Sottet, J.-S., Ganneau, V., Calvary, G., Coutaz, J., Demeure, A., Favre, J.-M., and Demumieux, R. Model-driven adaptation for plastic user interfaces. In INTERACT (1), 2007.