Meta-Modeling Runtime Models

Grzegorz Lehmann[1], Marco Blumendorf[1], Frank Trollmann[1], Sahin Albayrak[1],

[1] DAI-Labor, Technische Universität Berlin, Ernst-Reuter-Platz 7, 10587 Berlin, Germany
{Grzegorz.Lehmann, Marco.Blumendorf, Frank.Trollmann, Sahin.Albayrak}@dai-labor.de

Abstract. Runtime models enable the implementation of highly adaptive applications but also require a rethinking in the way we approach models. Metamodels of runtime models must be supplemented with additional runtime concepts that have an impact on the way how runtime models are built and reflected in the underlying runtime architectures. The goal of this work is the generalization of common concepts found in different approaches utilizing runtime models and the provision of a basis for their meta-modeling. After analyzing recent works dealing with runtime models, we present a meta-modeling process for runtime models. Based on a meta-metamodel it guides the creation of metamodels combining design time and runtime concepts.

Keywords: Meta-modeling, Models@Runtime, runtime models, meta-metamodel.

1 Introduction

(Self-)Adaptive applications are required to adapt dynamically at runtime, often to situations unforeseeable at design time. Application code generated from design time models fails to provide the required flexibility, as the design rationale held in the models is not available at runtime. To tackle this issue the use of runtime models (or models@run.time) has been proposed. Runtime models enable the reasoning about the decisions of developers when they are no longer available. Additionally, they provide appropriate abstractions from code-level details of the applications at runtime.

Although the idea of utilizing models at runtime is not new, there is still a lack of common understanding and suitable methodologies for the definition of runtime models. Moving the models from design time to runtime raises questions about the connection of the models to the runtime architecture, about synchronization and valid modifications of the models at runtime or the identification of model parts specified at design time and those determined at runtime.

The goal of this work is the generalization of common concepts found in different approaches utilizing runtime models and the provision of a basis for their meta-modeling. The approach brings:

- A common understanding of runtime models and their concepts
- Means for comparing and discussing about different runtime models
- Basis for achieving future interoperability
- Basis for the definition of a meta-modeling process for runtime models

The next section presents some exemplary works dealing with runtime models (2.1) and discusses their common properties (2.2). In section 3 our approach to meta-modeling runtime models is described. Section 4 concludes this paper.

2 Related Work

Model-driven engineering is a promising approach to the development of complex systems and applications. Since its emergence, model-based development aims at expressing different aspects of application on different levels of abstraction within different models. Utilizing formal models takes the design process to a computer-processable level, on which design decisions become understandable for automatic systems. The principles of model-driven architectures [9] have been successfully applied in different domains, e.g. the user interface engineering domain, where application code is generated from models.

Modern context-sensitive applications are required to adapt dynamically to context of use situations unforeseeable at design time. This requirement leads to the recent extension of model utilization's scope from design time to runtime.

2.1 Approaches Utilizing Models at Runtime

Models are utilized at runtime in different domains and for different purposes. This section analyzes exemplary approaches from several fields, ranging from model-based simulation and validation, adaptive and self-managing systems, to executable and reconfigurable models. Depending on the application domain the models fulfill different roles, but some shared similarities can be identified.

[12] describes the Cumbia platform, as a runtime system for executable runtime models, aiming at the provisioning of reusable monitoring and control tools. Integrating the execution logic and semantics behind the evolution of the model over time as part of the model leads to self-contained executable models. Cumbia's models are based on the idea of open objects, consisting of an *entity*, a state machine describing the entity's lifecycle and a set of actions triggered by the transitions of the state machine. Cumbia identifies four types of runtime model information:

- Structure of models - the static information about the application
- State of the elements in the models
- Historical information - the trace of model elements' state during the execution
- Derived information - additional information not directly included in the model but derived from it, e.g. by means of calculations

A slightly different approach to application monitoring is presented in [1]. The authors show how state machine logic can be embedded in object-oriented code. A runtime environment extracts the annotated state chart information at runtime and executes it. This way the runtime environment provides control of the application, enables the logging of its workflow and debugging of events. In the implementation, Java code is connected to the state charts by means of special classes, interfaces and annotations. Rather than being created and manipulated at design time, the state machine model is extracted from code at runtime.

Another approach for model-based (rapid) development of software is discussed in [7]. The authors propose a layered debugging architecture for their model-based applications. In an example the authors extend the UML state diagram metamodel with elements holding dynamic runtime information. The metamodel is thus split into a static and a dynamic part. However the categorization of design and runtime information is not further generalized.

The utilization of models at runtime is also common in approaches dealing with model-based design and adaptation of large, (self-) adaptive systems, like [14], [5] and [6]. The configuration of the systems and the possible adaptations are held in models at runtime. Adaptations are performed on the running system by transforming the models of the system.

In the ALIVE approach [14] executable code is generated from application models by means of transformations. If an adaptation is necessary at runtime, the models are modified and the executable code is regenerated. A monitoring mechanism assures that the application is paused for the time of adaptation and restarted when the new executable code is loaded.

In [5] an adaptation model holds information about possible variants of the system, constraints expressing valid configurations of the system and rules defining when adaptations should be performed. A context model represents the environment of the application and is the basis for the adaptation rules. Sensors deployed in the environment and in the system assure that the information in the models is up-to-date.

In the Rainbow framework [6] the architecture monitors and adapts the system through abstract models. The system layer consists of probes and effectors. The former observe and measure system states. The latter carry out the adaptations performed on the model level in the system. On the architecture layer, adaptation operators and strategies are provided. Operators determine the reconfiguration action that can be performed on the system. Strategies describe how operators need to be applied to achieve certain system properties.

The idea of utilizing models at runtime drives the design of executable models and languages. Kermeta, presented in [11], extends the Essential Meta Object Facility (EMOF) with action semantics. The composition of an existing meta language with an action metamodel results in an executable meta-language, enabling the definition of domain specific languages with precisely defined operational semantics. The Kermeta metamodel enhances the EMOF metamodel with typical action expressions (e.g. Conditional, Assignment, Loop).

[10] present Kermeta at RunTime (K@RT), a framework for adaptive software systems reconfigurable at runtime. K@RT supervises component-based systems by maintaining a reference model at runtime. The model provides a high-level view of the system. Modifications performed on the model are propagated into the underlying running system by automatically generated reconfiguration scripts. The authors propose a generic and extensible Metamodel for Runtime Models that represents component-based systems at runtime and aims at abstracting a running system. Composed of three packages (type, instance and implementation) and compatible with the Service Component Architecture (SCA), the metamodel enables the description of component-based software structures.

[8] propose FAME as a polyglot library capable of maintaining the connection of models and code at runtime. FAME enables the adaptation of software at runtime

through modifications of the models and even the meta-models by means of a set of basic operations (Get, Set, Create, Delete). FAME is capable of maintaining the causal connection between models and several programming languages, e.g. Smalltalk, Ruby or Java (with some limitations).

This presented some exemplary works utilizing runtime models. The next section discusses what the common properties of the different approaches are and what definitions can be used to generalize runtime models.

2.2 Generalizing Runtime Models

Although many approaches utilize models at runtime, none known to us does explicitly deal with the issues of creating metamodels of runtime models. Most works in the area of runtime models focus on defining special adaptation (e.g. as transformations executable at runtime) or system models (e.g. component networks), rather than looking at the common characteristics of runtime models.

An analysis of model dynamics and executability has been performed in [3]. The therein proposed classification of model elements in executable models comes nearest to a meta-metamodel. The authors differentiate three parts of dynamic models:

- Definition part – is the static part of a model, defined at design time
- Situation part – includes all elements describing the dynamic state of a model during its execution, and finally the
- Execution part – specifying the transitions of the model from one state to another, in other words its execution logic

The proposed classification has been a good starting point for our work, but, because of its focus on executable models, it does not fully apply to runtime models. For example, not every runtime model must have a definition part defined at design time. There are surely runtime models built up completely at runtime. Thus we have searched for a different basis for classifying runtime models.

In our view, the key for classifying and generalizing elements of runtime models lies in their causal connection. In [2] a model@run.time has been defined as a *causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective*. A runtime model provides up-to-date information about the system under study (SUS) and enables to perform adaptations of the system by means of model modifications.

In [13] and [4] the classification of descriptive and specification (also called prescriptive) models is discussed. According to [13] a model is descriptive if *all statements made in the model are true for the SUS*. On the other side a specification model prescribes how the system should be: *a specific SUS is considered valid relative to this specification if no statement in the model is false for the SUS*. Favre [4] proposes to use the term or truth to distinct if the model or the system *has the truth*. In case of runtime models, both the system and the runtime model have (parts of) the truth. Due to their causal connection, runtime models describe systems with their states and, at the same time, specify how the systems should behave.

The importance of the causal connection can be observed in the approaches presented in section 2.1. Most of them posses means for connecting the runtime models with the system under study, although the description/specification ratios

strongly differ. In works focusing on model executability, e.g. [11], the models have an either strong or sole prescriptive role. In self-adaptive systems, like [5] and [6], the utilized runtime models mostly have both, descriptive and specification, parts. On the other end, when runtime models are used for debugging and monitoring of applications (e.g. [1]), the descriptive character dominates.

Another common property of runtime models is that they evolve over time. The modifications of the models can be performed in different ways, e.g. by means of transformations, predefined operations or by special tools. Depending on whether the prescriptive or descriptive part of the model is modified, the changes have different consequences. Modifications of the prescriptive elements (e.g. performed by an adaptation engine) lead to changes in the system. Modifications of the descriptive parts of runtime models are mostly triggered by the system (e.g. probes in [6]) - whenever the system changes, its representation in the model must also change.

The identified typical properties of runtime models lead to requirements posed on their metamodels. Metamodels of runtime models must provide modeling constructs enabling the definition of:

- prescriptive part of the model specifying how the system should be
- descriptive part of the model specifying how the system is, i.e. the state of the SUS at runtime (similar to the situation part defined in [3])
- valid model modifications of the descriptive parts, executable at runtime
- valid model modifications of the prescriptive parts, executable at runtime
- causal connection in form of information flow between the model and its SUS

The following sections present a meta-modeling process addressing the above requirements.

3 Meta-Modeling Runtime Models

This part presents a process guiding the meta-modeling of runtime models (sections 3.1-3.4). Section 3.5 describes the meta-metamodel underlying this process.

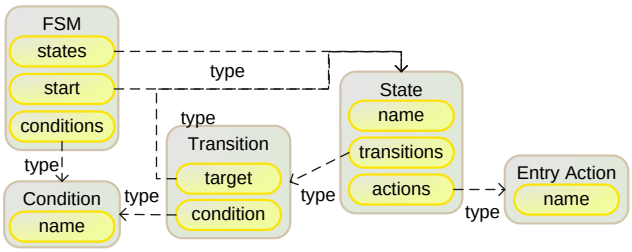

Fig. 1. Metamodel of finite state machines consisting of *States* with *Entry Actions* and *Transitions* bound to *Conditions*.

For illustration purposes, the process is applied to a simplified finite state machine (FSM) metamodel, depicted in Fig. 1. The metamodel defines a finite state machine element *FSM* consisting of *states*, of which one *State* is the *start* state. *States* are connected with each other via *Transitions*. The *FSM* provides *conditions* bound to transitions. Additionally each *State* can be associated with entry actions (*EntryAction* elements) executed upon the activation of the state.

The presented metamodel describes typical design-time models, with no runtime concepts included. It can be used to statically describe state machines but provides limited utility at runtime. However, in our example scenario we wish to use the FSM models both at design- and runtime. At design-time we wish to specify the behavior of software components in form of FSMs. At runtime we want to execute, monitor and inspect the state of the FSM models.

In the following the metamodel is extended with runtime concepts so it enables the definition of FSM runtime models. The meta-modeling process consists of four steps; each of the following subsections is dedicated to one of the subsequent steps.

3.1 Identify the Prescriptive and Descriptive Parts

To use the FSM models at runtime we must first identify elements of the models, which describe the runtime state of the system under study. At runtime, *Conditions* of a FSM become fulfilled and lead to the execution of the associated *Transitions*, which then activate *target* states. The example metamodel is therefore extended by adding an *active* attribute to the *State* and an *isFulfilled* attribute to the *Condition*. These descriptive attributes, marked orange in Fig. 2, hold the state of a FSM at runtime.

The distinction between the prescriptive and descriptive elements is necessary to clearly separate parts of a model altered in order to change the behavior of the system from the parts storing the runtime state of the system. In the example FSM metamodel, a state and the conditions of its transitions belong to the specification part, but whether a state has been activated or a condition fulfilled belongs to the descriptive part and is determined at runtime.

The differentiation between prescriptive and descriptive elements cannot be based on their type or class, but depends on the relationship of the element to other elements. Model elements of a specific type may in some cases be descriptive elements and in other cases prescriptive elements. It only counts whether the element is aggregated in a prescriptive- or descriptive field.

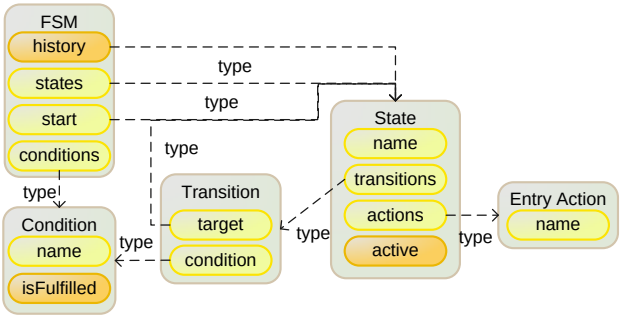

Fig. 2 Finite state machine metamodel with the orange marked descriptive elements *history*, *isFulfilled* and *active*, holding the state of the FSM at runtime.

In case of the example runtime FSM models, the state and transition hierarchy is defined by the model developer at design time. The states composing the FSM are thus prescriptive elements (e.g. elements held in *FSM.states*, *FSM.start* or *Transition.target*). However, an FSM may also store a *history* list of states activated in the past. The history is a result of runtime execution of the model and thus belongs

to its descriptive part. This way, model elements of type *State* are either prescriptive or descriptive depending on their relationship to other model elements. As shown in Fig. 2, *States* are descriptive elements, if they are part of *FSM.history*, or prescriptive elements, if they belong to the design-time state network specification (*FSM.states*). The latter are defined by the developer, the former are determined at runtime.

3.2 Modifications of Descriptive Elements

In the previous section the example metamodel has been enhanced with descriptive elements that enable to describe the state of a FSM model at runtime. In the next step of the meta-modeling process, available operations that can be performed on the descriptive part of the model must be identified. The example FSM metamodel is thus enhanced with operations, which describe the transitions of FSM models from one state to another (i.e. the FSM execution logic). We refer to these operations as *DescriptionModificationElements*.

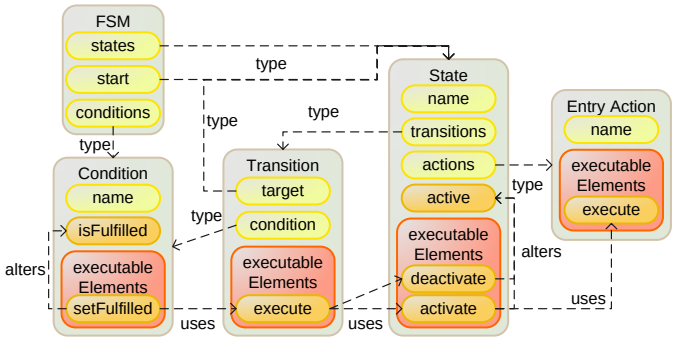

Fig. 3. Finite state machine metamodel with *DescriptionModificationElements setFulfilled*, *execute*, *activate* and *deactivate*.

Fig. 3 pictures the FSM metamodel with *DescriptionModificationElements* altering the state of FSMs at runtime. The *State* type has been enhanced with the *DescriptionModificationElements activate* and *deactivate*, which *alter* the *active* attribute of *States*. Activation of a *State* leads to the execution of its entry actions, so the *activate* operation *uses* the *execute* operation of *EntryAction*. *States* become activated and deactivated by executed transitions. Transitions are triggered by the fulfillment of the associated conditions.

The *DescriptionModificationElements* represent procedures or actions altering the elements of conforming runtime models. Through them a metamodel provides the ability to insert new information about the system into the models in a well-defined manner, even at runtime. For example, the *DescriptionModificationElement setFulfilled* makes it possible to inform an FSM model about a condition fulfilled in the system under study.

At this point of the process the FSM metamodel enables the definition of runtime models with state information and execution logic as alteration of this information (*DescriptionModificationElements*). The next step deals with the identification of *SpecificationModificationElements* that enable the modification of the prescriptive

part of the conforming FSM models. We refer to the modifications of prescriptive elements as adaptations, because they change the behavior of the system under study.

3.3 Modifications of Prescriptive Elements

One of the main purposes of runtime model utilization is the adaptation of the modeled application to varying context situations by means of model modifications. However, arbitrary reconfiguration of application models very soon leads to inconsistencies and can destroy the integrity of the adapted models.

The definition of possible model adaptations is an integral part of the meta-modeling process. It is the task of the meta-modeler to define possible modifications of the conforming models and their impact on the models. Only so can the correctness of the adaptations and the consistency of the adapted models at runtime be assumed.

The meta-modeling of model adaptations can again be exemplified using the FSM metamodel. A possible and often feasible adaptation of a FSM-based application is the adding of special states or entry actions. Such adaptations can, for example, be necessary if the context of the application changes and parts of the state network must be replaced with alternatives.

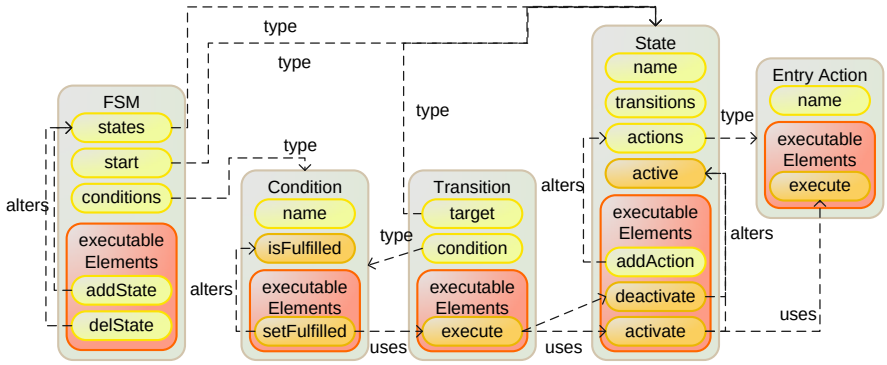

Fig. 4. FSM metamodel with *SpecificationModificationElements addState*, *delState*, *addAction*.

To enable the adding and removing of states in a finite state machine at runtime, the example metamodel is extended with *SpecificationModificationElements addState* and respectively *delState*. Fig. 4 shows the FSM metamodel with the new elements. Both *alter* the *states* of the adapted *FSM*. To retain the readability of the figure, we did not draw the *SpecificationModificationElements addTransition* and *delTransition* needed for reconfiguration of the transition network.

The difference between the *Description*- and *SpecificationModificationElements* is essential. While the former only change the model, so it reflects the state of the SUS at runtime (e.g. *activate* or *deactivate* in the example FSM metamodel), the latter have the power to modify the structure and behavior of the SUS (e.g. *FSM.addState* or *FSM.delState*). The *SpecificationModificationElements* have thus a much stronger impact on the models and their adaptation capabilities.

After identifying the runtime elements of a runtime model, defining the valid modification of both its descriptive and prescriptive parts, the meta-modeler has to

deal with one final runtime concept. The next section describes the last step of the meta-modeling process, which is the identification of the causal connection between the runtime model and its system under study.

3.4 Identify the Causal Connection

The connection between a runtime model and its system under study is referred to as the *causal connection*. The concept expresses the interrelation or causal loop between the model that represents a system and a system that must act according to the model. During the meta-modeling process the causal connections between the conforming runtime models and their systems under study must be identified.

Meta-modeling the causal connection comprises the definition of both directions of communication between the runtime models and their SUS. The influence of the model on the system and the synchronization of the model, based on the occurrences in the system, must be specified. It is thus essential to identify, how descriptive and prescriptive elements of the models communicate with the SUS.

The approaches described in section 2.1 present different ways of handling the causal connection. In Rainbow [6] the *effectors* are responsible of adapting to system to the current structure of the model. *Probes*, or *sensors* in [5], assure the information flow in the opposite direction – from the system and its environment into the model. We generalize such elements by the term of *proxy* elements.

A proxy element fulfills the role of an interface between the runtime model and its system under study. To enable the explicit definition of proxies within metamodels we use the proxy type. It enables the classification of model elements connected to entities outside of the model.

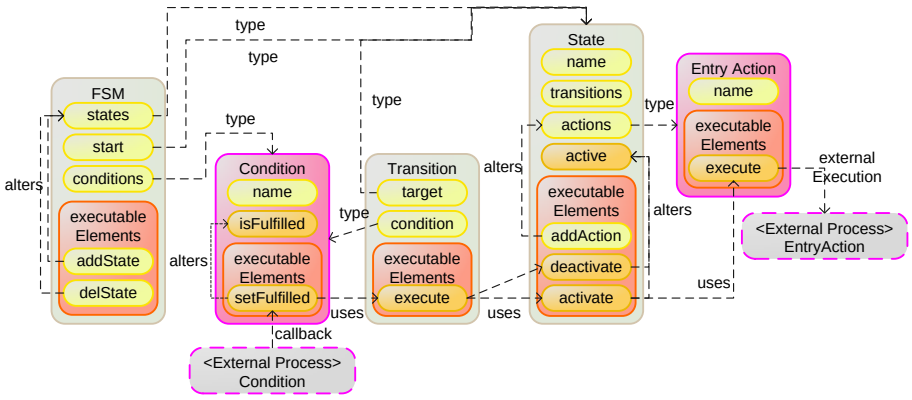

Fig. 5. FSM metamodel with *Condition* and *EntryAction* proxies handling the causal connection.

The information flow between the proxy elements and the outside world can be bidirectional. On the one side proxies synchronize the descriptive elements of the model with the state of the SUS, and on the other side they adapt the system according to the prescriptive part of the model. To achieve the first the proxies are provided with *DescriptionModificationElements*. For the model-SUS synchronization the proxies forward calls of *SpecificationModificationElements* to the SUS.

In the example FSM metamodel two proxy types have been identified: *Condition* and *EntryAction*. An FSM model must become aware of condition fulfillment occurring in the SUS. Therefore, as shown in Fig. 5, the *Condition* proxies expose the *setFulfilled* operation to external condition processes. This way, whenever a condition is fulfilled, external components inform the FSM model using the *setFulfilled* element. The *EntryAction* proxies do not expose any operations to the external processes, but trigger action execution in external processes outside of the model.

The identification of proxy elements enables an explicit and clear definition of the boundaries of runtime models. The communication between the model and the system via *Description-* and *SpecificationModificationElements* ensures that the synchronization occurs in a metamodel conformant way and does not interfere with the execution logic of the model. In the FSM example, the proxy elements causally connect the models with running systems through well-defined interfaces. The *Condition* proxies ensure that the FSM model reflects the state of the SUS at runtime. The *EntryAction* proxies enable the model to influence the SUS upon state changes.

We have presented a meta-modeling process, which identifies and makes explicit the runtime concepts necessary for the utilization of models at runtime. The next section sums up the ideas behind this process in form of a meta-metamodel.

3.5 Meta-Metamodel

Defining metamodels of runtime models requires a meta-modeling language that provides means for the expression of the described runtime concepts within the metamodels. Meta-modeling languages are defined in form of special metamodels, so called meta-metamodels. We thus present a meta-metamodel, which provides necessary constructs for formalizing metamodels of runtime models.

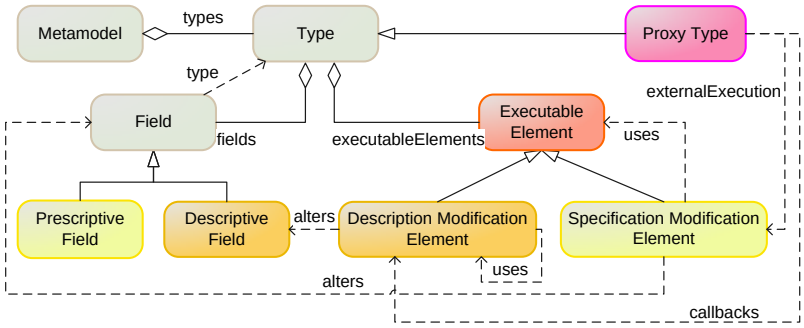

Fig. 6. Meta-metamodel of runtime models.

The meta-metamodel, shown in Fig. 6, prescribes that each conforming metamodel defines *Types* composed of *Fields* and *ExecutableElements*. *Fields* represent relationships between types (often referred to as attributes, associations, references, etc.) and are classified as either *Prescriptive-* or *DescriptiveFields*. Intuitively, model elements held in prescriptive fields are prescriptive elements and those held in descriptive fields are descriptive elements. The differentiation of fields enables the identification of descriptive and prescriptive parts of conforming models during the meta-modeling process.

The *ExecutableElements* represent operations enabling the modification of model elements. Depending on whether the modifications influence the descriptive or the prescriptive part of the model, *ExecutableElements* are refined as either *DescriptionModificationElements* (DME) or *SpecificationModificationElements* (SME). As explained in previous sections, the DMEs encapsulate the state synchronization of the models conforming to the metamodel, whereas the SMEs represent possible model and system adaptations.

The descriptive elements of the model are held in the *DescriptiveFields*. Therefore each DME defines, which *DescriptiveFields* it modifies, using the *alters* association. Associating a DME with other DMEs by means of the *uses* association the meta-modeler expresses that the execution of the DME is composed of or includes the execution of the associated DMEs (as the *State.activate* DME using *EntryAction.execute* in case of the FSM metamodel example).

Performing an adaptation of the model may not only influence its prescriptive part. In most cases it impacts its state as well. For this reason the SMEs can define *alters* and *uses* associations to both types of *Fields* and *ExecutionElements*.

Finally, the special *Proxy* type enables the formalization of the causal connection of runtime models. It classifies model elements connecting the model with its SUS. At runtime a proxy element mediates with an external element through a clearly defined communication interface. The interface is specified in form of *ExecutableElements*, either called during the model adaptation to influence the SUS (*externalExecution*) or available to the proxies to push information about the SUS into the model (*callbacks*).

4 Conclusions and Outlook

On the basis of our experiences with runtime models, we have presented a meta-modeling process. The process identifies core runtime concepts reoccurring in runtime models and helps supplementing traditional, design time models with them. The process and the constructs of the meta-metamodel are sufficient to distinguish the descriptive and prescriptive (specification) parts of runtime models as well as to identify operations for their modification *(ExecutableElements*). Furthermore the causal connection of the runtime model and its system under study can be described using the *Proxy* type. This way the meta-metamodel covers all aspects of meta-modeling runtime models identified in section 2.2.

We have utilized our approach to create a large set of metamodels, ranging from the FSM metamodel presented in this paper to metamodels from the user interface engineering domain (task, UI, layout or context metamodels). Our implementation is based on the popular Eclipse Modeling Framework (EMF). To assure a possibly high compatibility of our models with EMF we define our metamodels as plain EMF metamodels enhanced with some special annotations (e.g. annotating that an attribute expressed in Ecore is a *DescriptiveField*). The use of annotations makes our metamodels readable and usable for EMF tools (which simply ignore our custom annotations) and at the same time enables to extract the additional information about the runtime concepts of the conforming models.

We have defined the meta-metamodel in form of an Ecore metamodel and created transformations between annotated metamodels and the meta-metamodel. This approach enables to define metamodels of runtime models with full advantages of EMF tools and work with the meta-metamodel as with a plain EMF metamodel.

In the future we will explore the possibilities of using the meta-metamodel to achieve interoperability between different runtime model approaches (across technological spaces). We are working on additional metamodel transformations that will enable us to transform metamodels from technological spaces other than Ecore into the format of the meta-metamodel. We are also working on a reconfiguration metamodel, defined on the basis of the meta-metamodel. Combined with the transformations it will enable us to reconfigure and adapt runtime models from different technological spaces in one reconfiguration model.

References

[1] Moritz Balz, Michael Striewe, and Michael Goedicke. Embedding State Machine Models in Object-Oriented Source Code. In: *3rd Int. Workshop on Models@run.time*, 2008.

[2] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@run.time. In *Computer*, 42(10), 2009.

[3] Erwan Breton and Jean Bézivin. Towards an understanding of model executability. In *Proc. of the International Conference on Formal Ontology in Information Systems*, 2001.

[4] J. Favre. Foundations of Model (Driven) (Reverse) Engineering -- Episode I: Story of The Fidus Papyrus and the Solarus, In *Post-Proc. of Dagstuhl Seminar on Model Driven Reverse Engineering*, 2004.

[5] Franck Fleurey, Vegard Dehlen, Nelly Bencomo, Brice Morin, and Jean-Marc Jézéquel. Modeling and validating dynamic adaptation. In *3rd Int. Workshop on Models@run.time*, 2008.

[6] S.-W.; Huang A.-C.; Schmerl B.; Steenkiste P. Garlan, D.; Cheng. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In *Computer*, 37(10), 2004.

[7] Philipp Graf and Klaus D. Müller-Glaser. Gaining insight into executable models during runtime: Architecture and mappings. In *IEEE Distributed Systems Online*, 8(3), 2007.

[8] Adrian Kuhn and Toon Verwaest. Fame, a polyglot library for metamodeling at runtime. In *3rd Int. Workshop on Models@run.time*, 2008.

[9] Joaqin Miller and Jishnu Mukerji. *Model Driven Architecture (MDA)*. Object Management Group, omg document ormsc/2001-07-01 edition, 2001.

[10] Brice Morin, Olivier Barais, and Jean-Marc Jézéquel. K@rt: An aspect-oriented and model-oriented framework for dynamic software product lines. In *3rd Int. Workshop on Models@run.time*, 2008.

[11] Pierre A. Muller, Franck Fleurey, and Jean M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proc. of the 8th International Conference on Model Driven Engineering Languages and Systems*, 2005.

[12] Mario Sanchez, Ivan Barrero, Jorge Villalobos, and Dirk Deridder. An execution platform for extensible runtime models. In *3rd Int. Workshop on Models@run.time*, 2008.

[13] Ed Seidewitz. What models mean. *IEEE Software*, 20(5), 2003.

[14] Athanasios Staikopoulos, Sébastien Saudrais, Siobhán Clarke, Julian Padget, Owen Cliffe, and Marina De Vos. Mutual dynamic adaptation of models and service enactment in alive*. In *3rd Int. Workshop on Models@run.time*, 2008.