5th International Workshop

Models@run.time

In conjunction with MODELS 2010

OSLO, NORWAY, October 3-8, 2010

http://www.comp.lancs.ac.uk/computing/users/bencomo/MRT10/

5th Workshop on Models@run.time at MODELS 2010

Oslo, Norway, October 5th 2010

Proceedings

Editors

Nelly Bencomo
Gordon Blair
Franck Fleurey
Cédric Jeanneret

Organization Committee

Nelly Bencomo
Lancaster University, UK

Gordon Blair
Lancaster University, UK

Franck Fleurey
SINTEF, Norway

Cédric Jeanneret
University of Zurich, Switzerland

Program Committee

Uwe Assman
Dresden, Germany

Betty Cheng
Michigan State University, USA

Fabio M. Costa
Federal University of Goias, Brazil

Jeff Gray
University of Alabama at Birmingham, USA

Jozef Hooman
Embedded Systems Institute, Netherlands

Paola Inverardi
Università dell'Aquila, Italy

Flavio Oquendo
University of Brittany, France

Arnor Solberg
SINTEF, Norway

Thaís Vasconcelos Batista
Federal University of Rio Grande do Norte, Brazil

Franck Chauvel
Peking University, China

Peter J. Clarke
Florida International University, USA

Holger Giese
Universität Postdam, Germany

Oystein Haugen
SINTEF, Norway

Gang Huang
Peking University, China

Jean-Marc Jezequel
IRISA, France

Rui Silva Moreira
Universidade Fernando Pessoa, Portugal

Mario Trapp
Frauenhofer IESE, Germany

Additional Reviewers

Rasmus Adler
Frauenhofer IESE, Germany

Andreas Svendsen
SINTEF, Norway

Yali Wu
Florida International University, USA

Andrew Allen
Florida International University, USA

Thomas Vogel
Universität Postdam, Germany

Xiaorui Zhang
SINTEF, Norway

Preface

Welcome to the 5th Workshop on Models@run.time at MODELS 2010!

This document contains the proceedings of the 5th Workshop on Models@run.time that will be co-located with the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS). The workshop will take place in Oslo, Norway, on the 5th of October, 2010. The workshop is organized by Nelly Bencomo, Gordon Blair, Franck Fleurey and Cédric Jeanneret.

From a total of 15 papers submitted 4 full papers, 6 posters were accepted. This volume gathers together all the 10 papers accepted at Models@run.time 2010. After the workshop, a summary of the workshop will be published to complement these proceedings.

We would like to thank a number of people who contributed to this event, especially the members of the program committee and additional reviewers who provided valuable feedback to the authors. We also thank to the authors for their submitted papers, making this workshop possible.

We are looking forward to having fruitful discussions at the workshop!

September 2010

Nelly Bencomo
Gordon Blair
Franck Fleurey
Cédric Jeanneret

Content

Session 1: Fundamental Concepts

Session 2: Evaluation and Experimentation

Poster Session: Applications

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.

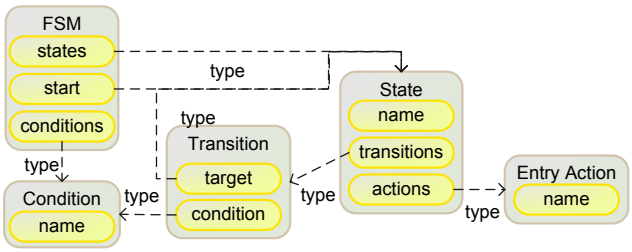


Fig. 1. Metamodel of finite state machines consisting of *States* with *Entry Actions* and *Transitions* bound to *Conditions*.

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.

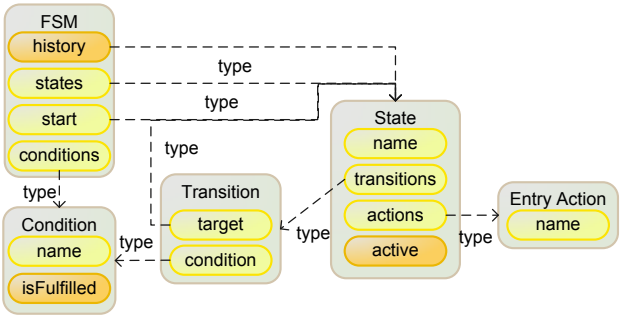


Fig. 2 Finite state machine metamodel with the orange marked descriptive elements *history*, *isFulfilled* and *active*, holding the state of the FSM at runtime.

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*.

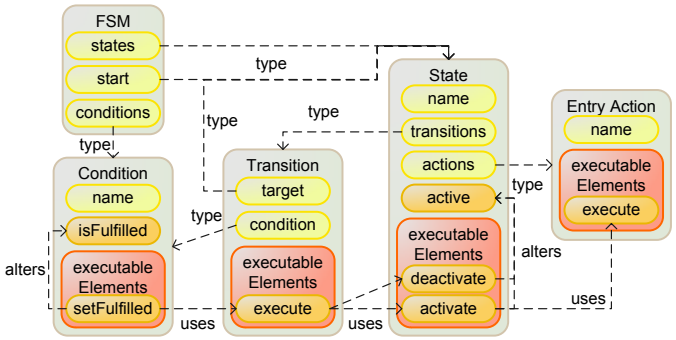


Fig. 3. Finite state machine metamodel with *DescriptionModificationElements setFulfilled*, *execute*, *activate* and *deactivate*.

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.

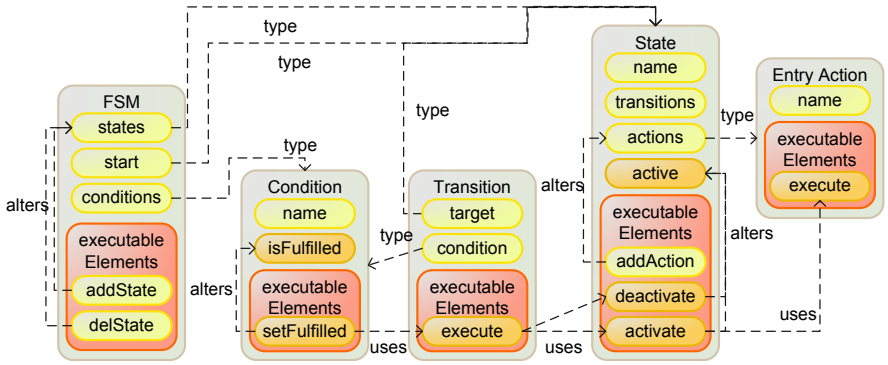


Fig. 4. FSM metamodel with *SpecificationModificationElements addState*, *delState*, *addAction*.

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.

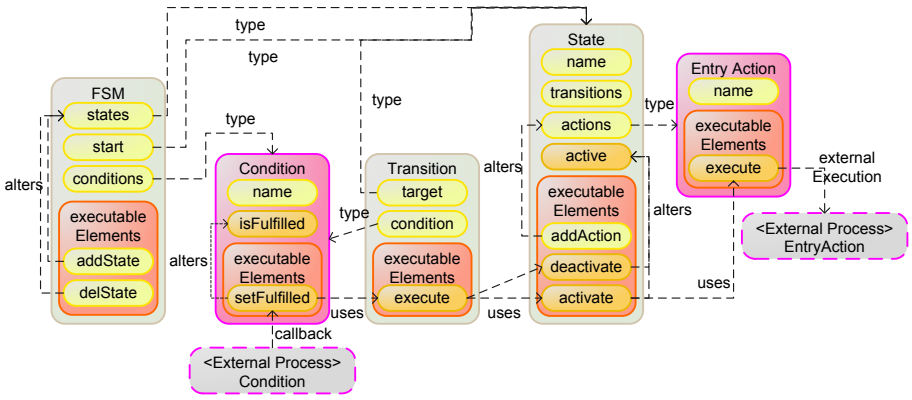


Fig. 5. FSM metamodel with *Condition* and *EntryAction* proxies handling the causal connection.

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.

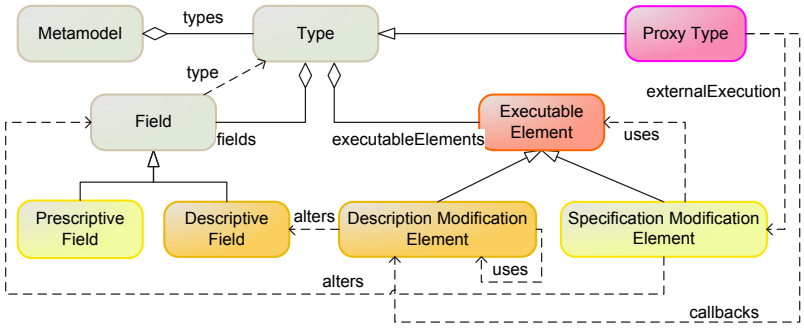


Fig. 6. Meta-metamodel of runtime models.

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.

Toward Megamodels at Runtime

Thomas Vogel, Andreas Seibel, and Holger Giese

Hasso Plattner Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam, Germany
prename.surname@hpi.uni-potsdam.de

Abstract. In model-driven software development a multitude of development models that are related with each other are used to systematically realize a software system. This results in a complex development process since these models and the relations between these models have to be managed. Similar problems appear when following a model-driven approach for managing software systems at runtime. A multitude of runtime models that are related with each other are likely to be employed simultaneously, and thus they have to be maintained at runtime. While for the development case *megamodels* have emerged to address the problem of managing development models and relations, the problem is rather neglected for the case of runtime models by applying ad-hoc solutions. Therefore, we propose to utilize concepts of megamodels in the domain of runtime system management. Based on existing work in the research field of runtime models, we demonstrate that different kinds of runtime models and relations are already employed simultaneously in several approaches. Then, we show how megamodels help in structuring and maintaining runtime models and relations in a model-driven manner while supporting a high level of automation. Finally, we present two case studies exemplifying the application and benefits of megamodels at runtime.

1 Introduction

According to France and Rumpe, there are two broad classes of models in *Model-Driven Engineering* (MDE): *development models* and *runtime models* [1]. Development models are employed during the model-driven development of software. Starting from abstract models describing the requirements of a software, these models are systematically transformed and refined to architectural, design, and implementation models until the source code level is reached.

In contrast, a runtime model provides a view on a running software system that is usually used for managing the system at runtime. Therefore, a runtime model serves as a basis and interface for monitoring, analyzing, and adapting a running system, which is realized by causally connecting the model and the system [1, 2]. Most approaches, like [3–6], employ *one* causally connected runtime model that reflects a running software system. While it is commonly accepted that developing complex software systems using *one* development model is not practicable, we argue that the whole complexity of managing a running software system cannot be covered by one runtime model defined by one metamodel. This is also recognized by Blair et. al who state "that in practice, it is likely that

multiple [runtime] models will coexist and that different styles of models may be required to capture different system concerns" [2, p.25].

At the *2009 Workshop on Models@run.time* we presented an approach for using multiple runtime models at different levels of abstraction simultaneously for monitoring and analyzing a running system [7]. Each runtime model defined by a different metamodel abstracts from the system and focuses on a specific concern, like architectural constraints or performance. At the workshop, our approach raised questions and led to a discussion about simultaneously coping with these models since concerns that potentially interfere with each other are separated in different models [8]. For example, any adaptation being triggered due to the performance state of a running system, which is reflected by one runtime model, might violate architectural constraints being reflected in a different model. Thus, there exists relations, like trade-offs or overlaps, between different concerns or models, which have to be considered for runtime management.

A similar issue appears during the model-driven development of software. A multitude of development models and relations between those models have to be managed. An example is the *Model-Driven Architecture* (MDA) approach that considers, among others, transformations of platform-independent to platform-specific models [9]. Thus, different development models are related with each other, and if changes are made to any model, the related models have to be updated by synchronizing these changes or by repeating the transformation. In this context *megamodels* have emerged as one means to cope with the problem of managing a multitude of development models and relations. The term megamodel is known since Jean Bézivin et al. and Jean Marie Favre published their ideas on modeling MDA and MDE, respectively [10, 11]. Both authors basically agree that *a megamodel is a model that contains models and relations between those models or between elements of those models* (cf. [10–13]).

In contrast, the problem of managing multiple models and relations is neglected for the runtime case and to the best of our knowledge there is no approach that explicitly considers this problem beyond ad-hoc and code-based solutions. In this paper, we present categories of conceivable runtime models and possible relations between those models. Based on that, we propose to apply existing concepts of megamodels for managing runtime models and relations. Such an approach provides a high level of automation for organizing and utilizing multiple runtime models and their relations, which supports the domain of runtime system management, e.g., by automated impact analyses across related models.

The rest of the paper is structured as follows. Section 2 discusses the categorization of runtime models and relations. Section 3 describes the application of megamodels at runtime, which is exemplified by two case studies in Section 4. Finally, the paper concludes and gives an outlook on future work in Section 5.

2 Runtime Models and Relations Between Them

In this section, we present categories of conceivable runtime models and relations between them based on the current state of the research field, primarily the past *Models@run.time* workshops [14] and our own work [7, 15–17]. However, we do

not claim that the presented categories are complete or that each category has to exist in every approach. Nevertheless, they indicate that different kinds of runtime models are likely to be employed simultaneously and that these models themselves together with their relations have to be managed at runtime.

2.1 Categories of Runtime Models

Each of the already mentioned approaches [3–6] employs one runtime model reflecting the running system. In contrast, our approach [7] provides multiple runtime models simultaneously, each of which reflects the running system and is specified by a distinct metamodel. Nevertheless, almost all of the other approaches also maintain additional model artifacts at runtime. These artifacts do not reflect the running system, but they are used for runtime management.

In the case of *Rainbow* [6], such artifacts are invariants that are checked on the runtime model, and adaptation strategies that are applied if invariants are violated. Morin et al. [4] even have in addition to an architectural runtime model reflecting the running system, a feature model describing the system's variability, a context model describing the system's environment, and a so called reasoning model that can take the form of event-condition-action (ECA) rules describing which feature should be (de-)activated on the architectural model depending on the context model. Thus, even if only one causally connected runtime model is used for reflecting the running system, several other models are employed at runtime. For the following categories as depicted in Figure 1, we consider any conceivable *Runtime Models* regardless whether they reflect a running system or not. The models are categorized according to their purposes and what they represent. Runtime models of all categories are usually instances of *Runtime Metamodels* conforming to *Runtime Meta-Metamodels*, which leverages typical MDE techniques, like model transformation or validation, to the runtime.

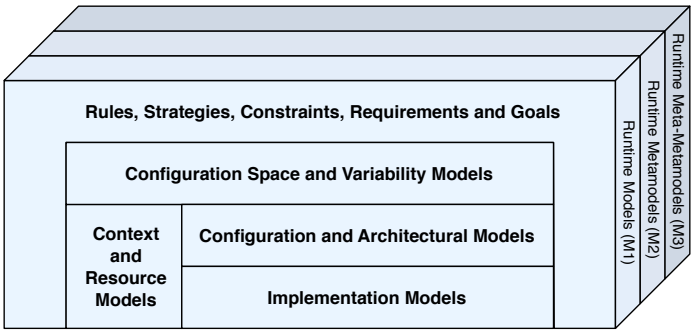


Fig. 1. Categories of Runtime Models

Implementation Models are similar to models used in the field of reflection to represent and modify a running system through a causal connection. Thus, these models are dynamic as they evolve consistently with a running system. Such models are based on the solution space of a system as they are coupled to the system's implementation and computation model [2]. Examples are models used in reflective programming languages, which represent the building blocks of

the languages [18, 19], or models that are directly coupled to platforms or technologies like *CORBA* [20]. Therefore, these models are rather platform-specific and at a low level of abstraction. Examples of such models are class or object diagrams and scenario-based sequence diagrams covering the interaction between objects or generally traces of a system [18, 21, 22]. Moreover, behavioral models in the form of statecharts, state machines, or generally automatons are used to reflect the current state of objects or of a running system [23–25].

Configuration and Architectural Models are at a higher level of abstraction than *Implementation Models*, but they usually also provide causally connected representations of running systems. Such a model reflects the current configuration of a system and it is the core model for monitoring and adapting the system. Since software architectures are considered as an appropriate abstraction level for performing adaptations, such models often provide architectural views on a running system [3–7, 17]. Thus, these models are often similar to component diagrams, which are often annotated or enhanced with elements or attributes to address non-functional properties, like performance or reliability [6, 7]. Therefore, these models are also the basis for analysis either by directly performing the analysis on them or by transforming them to specific analysis models, like queuing networks in the case of performance management. At a even higher level of abstraction, process or workflow models are also feasible to describe a running system from a business-oriented view [26]. Moreover, model types of the *Implementation Models* category, like statecharts or sequence diagrams, are also conceivable in this category, but at a higher level of abstraction. For example, a sequence diagram would consider the interactions between component instances instead of the interactions between objects.

In general, models of this category are rather related to problem spaces and they abstract from the implementation models and from underlying technologies to provide platform-independent views. This corresponds to the view of Blair et al. [2] on runtime models. With respect to a self-adaptive system, these models enable the self-awareness of the system at an appropriate level of abstraction, which is used as a basis for the feedback loop, i.e., for monitoring and analyzing the system, and for planning and executing adaptations on the system.

Context and Resource Models describe the operational environment of a running system. This comprises the context of a system, which is "any information that can be used to characterise the situation of an entity", while "an entity is a person, place, or object that is considered relevant to the interaction between a user and an application" [27, p.5] or in general to the operation of the application. Especially for a context-aware system, which is a system that adapts its behavior to changes in its context, the context has to be observed by sensors and described by a model. A simple example for a context is the user's location, which can be used in mobile systems to find services, like restaurants, in the vicinity of the user. To represent a context, a variety of models can be used: semi-structured tags and attributes, object-oriented or logic-based models [28], or some form of variables, like key value pairs [4, 28]. Even feature models have been proposed for modeling context [29].

Moreover, the operational environment consists of resources a running system requires for operation. These are logical resources, like any form of data, or physical resources, like hardware. An example for a resource model reflects the hardware infrastructure, like computing nodes and network links among nodes, on which the system is running. Therefore, such a resource model provides information whether any adaptation of a system is feasible based on the currently provided resources, like on which node a subsystem can be deployed.

Configuration Space and Variability Models specify potential variants of a running system, while *Configuration and Architectural Models* reflect the currently running variant of the system. Therefore, models of this category describe a system at the type level to span the system's configuration space and variability. Considering a component-based system, the configuration space is defined by the available types of components that can be instantiated and deployed to a running system. Thus, adaptation points in a running system and possible adaptation alternatives can be identified using these models.

Examples for models in this category are component type diagrams [16, 17], feature models originating from dynamic software product lines [4, 30, 31], or aspect models describing variants of a system and instances of these aspects are woven into configuration or architectural models for adapting the system [4, 32].

Rules, Strategies, Constraints, Requirements and Goals may refer to any model from the other categories and, therefore, their levels of abstraction are similar to the levels of the referred models. However, considering requirements or goals at runtime aims at higher levels of abstraction, even above the level of software architectures [33]. Models in this category define, among others, when and how a running system should be adapted. According to Fleurey and Solberg [34] there are two general approaches to specify adaptations. First, adaptation rules or reconfiguration strategies usually in some form of ECA rules describe when (periodically or at the occurrences of context or system events) and under which conditions, a system is adapted by performing reconfiguration actions. The second approach is based on goals a running system should achieve, and adaptation aims at optimizing the system with respect to these goals. This optimization process is based on utility functions to find the best or at least an appropriate target system configuration fulfilling the goals. Both approaches use models reflecting the current system, context, and resources to search the configuration space for a variant that is appropriate for the current state. For example, a goal-based optimization model is used in [4, 31, 35], and adaptation rules or reconfiguration strategies are used in [6, 29, 36].

Moreover, constraints on models of the other categories regarding functional and non-functional properties are used for runtime validation and verification purposes, and for guiding adaptations. If a constraint is violated, an adaptation can be triggered, as it is done in [6], or constraints may exclude certain kind of adaptations. Constraints can be expressed, among others, in the *Object Constraint Language* (OCL), like in [7] to check architectural constraints, or formally in some form of *Linear Temporal Logic* (LTL), like in [23] to verify adaptive systems at runtime. Though constraints can be seen as requirements that

are checked at runtime, recently the idea of *requirements reflection* has emerged, which explicitly considers requirements as adaptive runtime entities [33]. Thus, requirements models, like goal models, become runtime models that have to be related to *Configuration and Architectural Models* since any changes at the requirements level have to be reflected in the running system, and vice versa.

The presented categories show that different kinds of runtime models are possible and even employed simultaneously. Which categories are used, and which kind of and how many models for each of the used categories are employed is specific to each approach. This depends, among others, on the purposes of an approach (which functional and non-functional concerns are of interest, which management activities, like monitoring, analysis or adaptation, are supported, etc.) and on the domain of the managed system (embedded, mobile, or server-side systems, or even IT infrastructures, etc.). Based on the model categories, conceivable relations between runtime models are presented in the next section.

2.2 Relations Between Runtime Models or Model Elements

In the following, we outline exemplars of relations between runtime models to motivate the need for runtime management of relations together with the models.

As already mentioned, models of the category *Rules, Strategies, Constraints, Requirements and Goals* may refer to models of the other categories. For example, goal modeling approaches refine a top-level goal to subgoals recursively until each subgoal can be satisfied by an agent being a human or a software component [37]. Having a goal model at runtime, it is of interest which component of a running system actually satisfies or fails in satisfying a certain goal. Therefore, goals being reflected in a goal model refer to corresponding components of *Configuration and Architectural Models* such that a goal model and an architectural model are related at runtime. Moreover, goal satisfaction can be influenced by the current context of a system, such that goals and elements of a context model and, therefore, the goal model and *Context Model* are related with each other.

Another exemplar describes an instance-of relation between *Configuration Models* and *Configuration Space Models*. For example, a configuration space is defined by the types of available components and an actually running system consists of instances of these types. At runtime, this relation is useful for navigating from configuration model elements to corresponding configuration space model elements to find potential variability points for adaptations. Regarding the same dimension of abstraction, *Implementation Models* can be seen as refinements of *Configuration and Architectural Models* as they describe how a configuration and architecture is actually realized using concrete technologies. Thus, refinement relations are conceivable between models of these two categories.

Another relation can reflect the deployment or resource utilization of a system by means of relating *Architectural Model* elements and *Resource Model* elements, or in other words, which components of a running systems are deployed on which nodes and are consuming which resources. *Context and Resource Models* can also refer to *Configuration Space and Variability Models* since the configuration space

and variability of a system can be influenced by the current context or resource conditions. For example, a certain variant is disabled due to limited resources.

Besides relations between models of different categories, there can also exist relations between models of the same category. For example, in [7] several *Architectural Models* are employed reflecting the same system, but providing different views. However, these views overlap with each other, which can be considered as a kind of relation between these models. Furthermore, each model focuses on a certain non-functional concern, like performance, and any adaptation optimizing one concern might interfere with another concern. Thus, overlaps, trade-offs or conflicts between concerns respectively between the models are conceivable.

Finally, considering the levels of models, metamodels, and meta-metamodels, there exists conformance and instance-of relations between models of those levels.

The presented exemplars show that runtime models are usually not independent from each other, but they rather compose a network of models. Therefore, besides the runtime models also the relations between those models have to be managed at runtime. The concrete relations emerging in an approach depend, among others, on the purposes of the approach, the domain of the system and especially on the models that are employed.

3 Megamodels at Runtime

As it turned out in the previous sections, for runtime management different kinds of models and relations between models emerge. In such scenarios, it is important that these relations are maintained at runtime because this makes the relations explicit and, therefore, amenable for reasoning or analysis purposes. For example, an impact analysis is leveraged when knowing which models are related with each other. Then, the impact of any model change to related models can be analyzed by following transitively the relations and propagating the change. Moreover, relations can be classified, for example in critical and non-critical ones, and for certain costly analyses only the critical relations may be considered.

Nevertheless, relations to other models are usually not covered originally by all models because they were not foreseen when designing the corresponding modeling languages. Thus, a language for explicitly specifying all kinds of relations between various models is required for supporting the management of runtime models. Rather than applying ad-hoc and code-based solutions to relate models with each other, megamodels provide a language that supports the modeling of arbitrary models and relations between those models. Therefore, the management of models and relations itself is done in a model-driven manner enabling the use of existing MDE techniques for it. In general, megamodels for the model-driven development serve organizational and utilization purposes that should also be leveraged at runtime. Organizational purposes are primarily about managing the complexity of a multitude of models. Therefore, megamodels help in organizing a huge set of different models together with their relations by storing and categorizing them. According to Bézivin et al., megamodels act as some kind of registry for models [12] or even as a global map for the information assets of a company [10]. Likewise, megamodels can serve as a means to organize and

maintain runtime models and their relations in the domain of runtime system management since several models and relations can be simultaneously employed at runtime (cf. Sections 2.1 and 2.2).

Utilization purposes of megamodels are primarily about navigation and automation. Megamodels can be the basis for navigating through models by using relations between models. Thus, starting from a model, all related models can be reached in a model-driven manner instead of using mechanisms at a lower level of abstraction like programming interfaces. Having the conceivable relations between runtime models in mind (cf. Section 2.2), navigating between models at runtime is essential for a comprehensive system management approach.

Automation aims at increasing the efficiency by treating relations between models as executable units that take models as input and produce models as output. Thus, a megamodel can be considered as an executable process, and additional automations for executing a megamodel can be defined on top of a megamodel. For example, a megamodel can be used to automatically analyze the impact of model changes to other related models. Therefore, relations can be used to synchronize model changes to related models and these synchronized models are then analyzed to investigate the impact of the initial changes. This can be used at runtime to validate a planned adaptation on different models before the system is actually adapted. Finally, automation also considers the maintenance of models and relations, which should be automated as far as possible since models and relations are often both dynamic and they change over time.

Having outlined the application of megamodels at runtime, the following section presents two case studies exemplifying megamodels at runtime.

4 Case Studies

In this section, we outline two case studies from our previous works and how these case studies benefit from the application of megamodels at runtime.

4.1 IT Service Management

In [16] we presented a model-driven configuration management system (CMS) for advanced IT service management (ITSM) by applying several MDE techniques. The core of a model-driven CMS is a configuration management database (CMDB) that stores an as-is and a to-be *Configuration Model* of a managed system. Configuration models consist of configuration items and relations between items, while items are manageable units of a managed system, like servers or applications. On top of a model-driven CMS, we realized three simplified ITIL processes by using MDE techniques, namely, change management, release & deployment management, and service asset & configuration management.

The service asset & configuration management process is responsible for providing an up-to-date as-is *Configuration Model* in a CMDB. Furthermore, key performance indicators (KPIs) are implemented to provide more control on this process. An example KPI is the degree of discrepancy between the to-be and the as-is configuration models, which is the number of covered configuration items in both models divided by the number of items in the to-be configuration model.

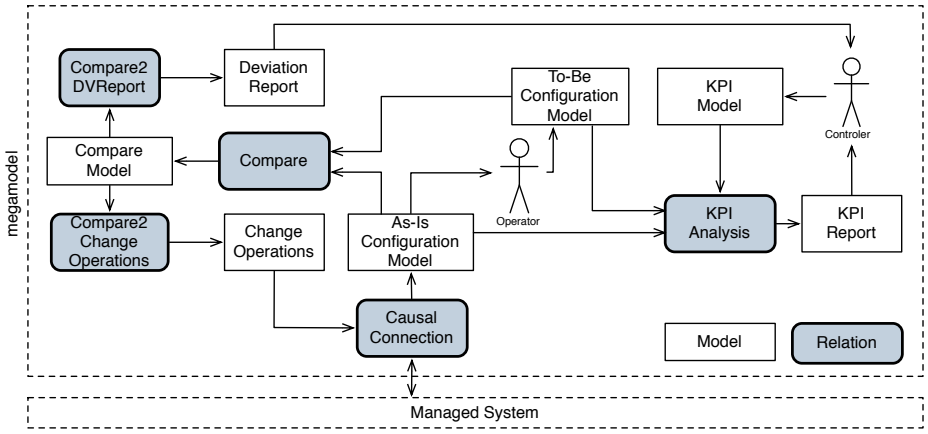


Fig. 2. A simplified megamodel example for ITSM

The change management process provides capabilities to define changes to a managed system based on models. Thus, an operator models changes directly on the as-is configuration model and then stores it as a to-be configuration model, which is further used by the release & deployment process to perform the modeled changes. Therefore, a set of change operations are automatically derived by comparing the defined to-be with the as-is configuration model.

Such a CMS can be appropriately captured by a megamodel, which is shown as a simplified example for ITSM in Figure 2. Additional actors are integrated for indicating manual interventions. The megamodel shows the models used in this system and the relations between these models. The *As-Is* and the *To-Be Configuration Models* belong to the category of *Configuration and Architectural Models*, and they are both used for the *KPI Analysis*. This analysis evaluates the KPIs specified as rules in the *KPI Model*, which therefore belongs to the model category of *Rules*, and the analysis results are described in a *KPI Report*.

Furthermore, model relations can be mapped to operations that are automatically executed, e.g., the *Compare* relation is implemented by an EMF Compare[1] operation or the *Compare2Change Operations* is a model transformation. Thus, whenever changes occur, i.e., the *To-Be Configuration Model* is modified, *Change Operations* are automatically derived and performed on the system, while the *KPI Analysis* observes the progress of performing the changes to the system.

4.2 Self-Adaptive Software

In the field of self-adaptive software, we presented an approach that employs several runtime models simultaneously for monitoring [7] and adapting [17] a system. This is outlined in Figure 3. A running *Managed System* is reflected by an *Implementation Model* and both are causally connected. However, the implementation model is platform-specific, complex, at a low level of abstraction, and related to the solution space of the system. Therefore, abstract runtime models

[1] Eclipse Modeling Framework Compare, `http://wiki.eclipse.org/EMF_Compare`

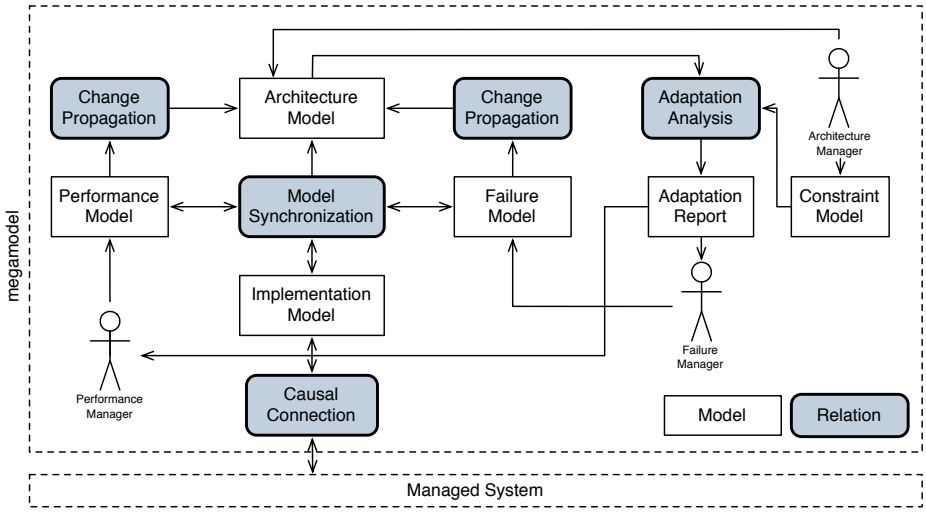


Fig. 3. A megamodel example for self-adaptive software

are derived from the implementation model using incremental and bidirectional *Model Synchronization* techniques. These abstract models can be causally connected to the system via the implementation model, and they belong to the category of *Configuration and Architectural Models*. Each of these abstract models focuses on a specific concern of interest, which leverages models related to problem spaces. An *Architecture Model*, a *Performance Model*, and a *Failure Model* are derived focusing on architectural constraints, performance, and failures of the system, respectively. Thus, specific self-management capabilities are supported by distinct models, like self-healing by the failure model or self-optimization by the performance model. Consequently, specialized autonomic managers, like a *Performance Manager* working on the performance model, can be employed.

However, adaptations performed by a certain manager due to a certain concern might interfere with other concerns covered by other managers. For example, adaptations based on the performance model, like deploying an additional component to balance the load, might violate architectural constraints covered by the *Architecture Model*, like the affected component can only be deployed once.

Since each concern is covered by a different model, megamodels can be used to describe relations, like interferences or trade-offs, between different models or concerns. Moreover, the coordination between different managers can be modeled with megamodels, which can be enacted at runtime to balance competing concerns, as outlined by the following scenario. Before any adaptation proposed by the performance or failure manager who change the performance or failure model, respectively, is executed on the system by triggering the *Model Synchronization*, the changes are automatically propagated to the architecture model (cf. *Change Propagation* relations in Figure 3). Then, the architecture manager takes the updated architecture model and the *Constraint Model* to analyze and validate the proposed adaptations (*Adaptation Analysis*). The resulting *Adap-*

tation Report is sent to the manager proposing the adaptation and it instructs either the execution of the proposed adaptation on the system or the rollback of the corresponding model changes depending on the analysis results.

Both presented case studies exemplified potential use cases for megamodels at runtime and benefits of megamodels for advanced system management approaches using multiple runtime models simultaneously.

5 Conclusion and Future Work

In this paper we have shown that the issue of complexity in the domain of model-driven development, caused by the amount of models and their relations, is also a problem in the domain of runtime system management and runtime models. Since for the latter domain this problem is rather neglected, we addressed it by presenting a categorization of runtime models and potential relations that can exist between models of the same or different categories. Based on that, we showed that megamodels are an appropriate formalism to manage runtime models and their relations. This has been exemplified by two case studies outlining the benefits in the domain of runtime system management by providing a high level of automation for organizing and utilizing runtime models and relations.

As future work, we plan to elaborate our categorization to incorporate other preliminary classifications comparing development and runtime models [1, 38] and describing dimensions of runtime models [2]. This includes possible categorizations of relations between runtime models. Finally, to evaluate this proposal, we will investigate the application of our megamodel approach designed for the development and deployment time [39] to the domain of runtime management.

References

1. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: Proc. of the ICSE Workshop on Future of Software Engineering (FOSE), IEEE (2007) 37–54
2. Blair, G., Bencomo, N., France, R.B.: Models@run.time: Guest Editors' Introduction. Computer **42**(10) (2009) 22–27
3. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating Synchronization Engines between Running Systems and Their Model-Based Views. In: Models in Software Engineering, Workshops and Symposia at MODELS 2009. Volume 6002 of LNCS. Springer (2010) 140–154
4. Morin, B., Barais, O., Jézéquel, J.M., Fleurey, F., Solberg, A.: Models@Run.time to Support Dynamic Adaptation. Computer **42**(10) (2009) 44–51
5. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: Proc. of the 20th Intl. Conference on Software Engineering (ICSE), IEEE (1998) 177–186
6. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer **37**(10) (2004) 46–54
7. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental Model Synchronization for Efficient Run-Time Monitoring. In: Models in Software Engineering, Workshops and Symposia at MODELS 2009. Volume 6002 of LNCS. Springer (2010) 124–139
8. Bencomo, N., Blair, G., France, R., Munoz, F., Jeanneret, C.: 4th International Workshop on Models@run.time. In: Models in Software Engineering, Workshops and Symposia at MODELS 2009. Volume 6002 of LNCS. Springer (2010) 119–123
9. Mellor, S.J., Scott, K., Uhl, A., Weise, D.: MDA Distilled: Principles of Model-Driven Architecture. Addison-Wesley, Boston (2004)
10. Bézivin, J., Gérard, S., Muller, P.A., Rioux, L.: MDA components: Challenges and Opportunities. In: 1st Intl. Workshop on Metamodelling for MDA. (2003) 23–41
11. Favre, J.M.: Foundations of Model (Driven) (Reverse) Engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In: Language Engineering for Model-Driven Software Development. Number 04101 in Dagstuhl Seminar Proceedings, IBFI, Schloss Dagstuhl (2005)
12. Bézivin, J., Jouault, F., Valduriez, P.: On the Need for Megamodels. In: Proc. of the OOPSLA/GPCE Workshop on Best Practices for Model-Driven Software Development. (2004)

13. Barbero, M., Fabro, M.D.D., Bézivin, J.: Traceability and Provenance Issues in Global Model Management. In: ECMDA-TW'07: Proc. of 3rd Workshop on Traceability. (2007) 47–55
14. Workshop on Models@run.time, http://www.comp.lancs.ac.uk/~bencomo/MRT/ (2006-2009)
15. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Model-Driven Architectural Monitoring and Adaptation for Autonomic Systems. In: Proc. of the 6th Intl. Conference on Autonomic Computing and Communications (ICAC), ACM (2009) 67–68
16. Giese, H., Seibel, A., Vogel, T.: A Model-Driven Configuration Management System for Advanced IT Service Management. In: Proc. of the 4th Intl. Workshop on Models@run.time. Volume 509 of CEUR-WS.org. (2009) 61–70
17. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: Proc. of the 5th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), ACM (2010) 39–48
18. Jouault, F., Bézivin, J., Chevrel, R., Gray, J.: Experiments in Run-Time Model Extraction. In: Proc. of 1st Intl. Workshop on Models@run.time. (2006)
19. Kuhn, A., Verwaest, T.: FAME - A Polyglot Library for Metamodeling at Runtime. In: Proc. of the 3rd Intl. Workshop on Models@run.time, Technical Report COMP-005-2008, Lancaster University (2008) 57–66
20. Costa, F., Provensi, L., Vaz, F.: Towards a More Effective Coupling of Reflection and Runtime Metamodels for Middleware. In: Proc. of 1st Intl. Workshop on Models@run.time. (2006)
21. Gjerlufsen, T., Ingstrup, M., Wolff, J., Olsen, O.: Mirrors of Meaning: Supporting Inspectable Runtime Models. Computer **42**(10) (2009) 61–68
22. Maoz, S.: Using Model-Based Traces as Runtime Models. Computer **42**(10) (2009) 28–36
23. Goldsby, H.J., Cheng, B.H., Zhang, J.: AMOEBA-RT: Run-Time Verification of Adaptive Software. In: Models in Software Engineering: Workshops and Symposia at MoDELS 2007. Volume 5002 of LNCS. Springer (2008) 212–224
24. Maoz, S.: Model-Based Traces. In: Proc. of the 3rd Intl. Workshop on Models@run.time, Technical Report COMP-005-2008, Lancaster University (2008) 16–25
25. Höfig, E., Deussen, P.H., Coskun, H.: Statechart Interpretation on Resource Constrained Platforms: a Performance Analysis. In: Proc. of the 4th Intl. Workshop on Models@run.time. Volume 509 of CEUR-WS.org. (2009) 99–108
26. Sanchez, M., Barrero, I., Villalobos, J., Deridder, D.: An Execution Platform for Extensible Runtime Models. In: Proc. of the 3rd Intl. Workshop on Models@run.time, Technical Report COMP-005-2008, Lancaster University (2008) 107–116
27. Dey, A.K.: Understanding and Using Context. Personal Ubiquitous Comput. **5**(1) (2001) 4–7
28. Schneider, D., Becker, M.: Runtime Models for Self-Adaptation in the Ambient Assisted Living Domain. In: Proc. of the 3rd Intl. Workshop on Models@run.time, Technical Report COMP-005-2008, Lancaster University (2008) 47–56
29. Acher, M., Collet, P., Fleurey, F., Lahire, P., Moisan, S., Rigault, J.P.: Modeling Context and Dynamic Adaptations with Feature Models. In: Proc. of the 4th Intl. Workshop on Models@run.time. Volume 509 of CEUR-WS.org. (2009) 89–98
30. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic Computing through Reuse of Variability Models at Runtime: The Case of Smart Homes. Computer **42**(10) (2009) 37–43
31. Elkhodary, A., Malek, S., Esfahani, N.: On the Role of Features in Analyzing the Architecture of Self-Adaptive Software Systems. In: Proc. of the 4th Intl. Workshop on Models@run.time. Volume 509 of CEUR-WS.org. (2009) 41–50
32. Ferry, N., Hourdin, V., Lavirotte, S., Rey, G., Tigli, J.Y., Riveill, M.: Models at Runtime: Service for Device Composition and Adaptation. In: Proc. of the 4th Intl. Workshop on Models@run.time. Volume 509 of CEUR-WS.org. (2009) 51–60
33. Bencomo, N., Whittle, J., Sawyer, P., Finkelstein, A., Letier, E.: Requirements reflection: requirements as runtime entities. In: Proc. of the 32nd ACM/IEEE Intl. Conference on Software Engineering (ICSE), ACM (2010) 199–202
34. Fleurey, F., Solberg, A.: A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In: Proc. of the 12th Intl. Conference on Model Driven Engineering Languages and Systems (MODELS). Volume 5795 of LNCS., Springer (2009) 606–621
35. Ramirez, A.J., Cheng, B.H.: Evolving Models at Run Time to Address Functional and Non-Functional Adaptation Requirements. In: Proc. of the 4th Intl. Workshop on Models@run.time. Volume 509 of CEUR-WS.org. (2009) 31–40
36. Dubus, J., Merle, P.: Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-based Systems. In: Proc. of 1st Intl. Workshop on Models@run.time. (2006)
37. Cheng, B.H., Sawyer, P., Bencomo, N., Whittle, J.: A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In: Proc. of the 12th Intl. Conference on Model Driven Engineering Languages and Systems (MODELS). Volume 5795 of LNCS., Springer (2009) 468–483
38. Bencomo, N.: On the Use of Software Models during Software Execution. In: Proc. of the ICSE Workshop on Modeling in Software Engineering (MISE), IEEE (2009) 62–67
39. Seibel, A., Neumann, S., Giese, H.: Dynamic Hierarchical Mega Models: Comprehensive Traceability and its Efficient Maintenance. Software and Systems Modeling **9** (2009) 493–528

Applying MDE Tools at Runtime: Experiments upon Runtime Models

Hui Song, Gang Huang *, Franck Chauvel, and Yanchun Sun

Key Lab of High Confidence Software Technologies (Ministry of Education)
School of Electronic Engineering & Computer Science, Peking University, China
{songhui06,huanggang,franck.chauvel,sunyc}@sei.pku.edu.cn

Abstract. Runtime models facilitate the management of running systems in many different ways. One of the advantages of runtime models is that they enable the use of existing MDE tools at runtime to implement common auxiliary activities in runtime management, such as querying, visualization, and transformation. In this tool demonstration paper, we focus on this specific aspect of runtime models. We discuss the requirements of runtime models to enable the use of model-driven tools, and present our tool to help provide such runtime models on the target systems. We apply this tool on a wide range of target systems, modeling the Android mobile system, the Eclipse GUI, the Java class structure, and the JOnAS inner structure. With the help of these runtime models, we perform the runtime management on these systems using classical MDE tools including OCL, QVT, and GMF.

1 Introduction

For a running system, developers often need to retrieve and update its data at runtime. The runtime data depict the system's configuration, structure, state, or environment. By analyzing and changing these runtime data, developers monitor and control the system at runtime to fix system defects, adapt to the changed environment, or meet newly emerged requirements. Take a mobile phone as a sample system, we may care about what wireless network (Wi-Fi) channels are currently available, as well as their signal intensity. We may also need to switch channels when necessary and possible.

However, manipulating the runtime data is not an easy task. Currently, most systems only provide low-level APIs for manipulating the runtime data [1], and developers have to write low-level code to invoke the APIs. For example, the code below illustrates how to invoke the Android (a mobile OS) API to print the signal IDs of available Wi-Fi channels.

```
1 WifiManager wm=(WifiManager) this
2     .getSystemService(Context.WIFI_SERVICE);
3 List<ScanResult> srs = wm.getScanResults();
4 for(ScanResult sr in srs)
5   Log.i("Wi-Fi_Signal_ID",sr.ssid);
```

* corresponding author

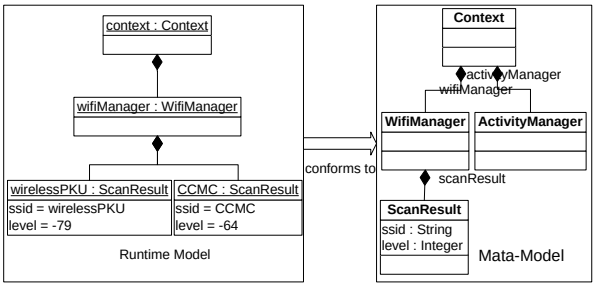


Fig. 1. A runtime model and its meta-model

It is tedious and error-prone to manage the system by directly using the management APIs. First, there lacks explicit definition about the data types. Second, there are different invocation manners for different systems or even different types of data inside the same system. Third, people have to re-implement many common auxiliary management activities on each of the APIs, such as querying, aggregation, visualization, etc.

Runtime model [2, 3, 1] provides a promising way to liberate people from the tedious APIs, and allow them to manipulate the runtime data in a higher abstraction level, utilizing the rich and mature MDE (model-driven engineering) techniques and tools, such as OCL for evaluation or querying, QVT for aggregation and analysis, visualization, etc. Figure 1 illustrates a sample runtime model and its meta-model for the Android system. Developers could use the following OCL rule to query the signal IDs of Wi-Fi channels.

```
self.wifiManager.scanResult->collect(e|e.ssid)
```

To enable the application of existing MDE tools, the runtime model should satisfy the following three requirements. Firstly, the model should be organized in a standardized form. Second, the runtime model must have an explicit meta-model which defines its semantics. Thirdly and most importantly, the model must have a *causal connection* with ever-changing system. That means if the system evolves, the model will change immediately, and if the model is modified, the system will change correspondingly.

These requirements call for a software agent to represent the runtime data as a standard model conforming to a specific meta-model, and to synchronize the model with the runtime data. We name such agents as "synchronizers". For a target system, *runtime model providers*, who are experts of the system and its API, develop such synchronizers, and *runtime model consumers*, usually the common developers, use the runtime model maintained by the synchronizer to manipulate the runtime data, using the MDE tools. Existing approaches on runtime model usually require runtime model providers to develop such synchronizers *by hand*[4, 1].

In this paper, we demonstrate a generative tool, *SM@RT*[1], which generates synchronizers for a wide class of systems. As shown in Figure 2, for a kind of

[1] SM@RT: Supporting Models at Run-Time, the tool and the case studies are available on line: `http://code.google.com/p/smatrt`

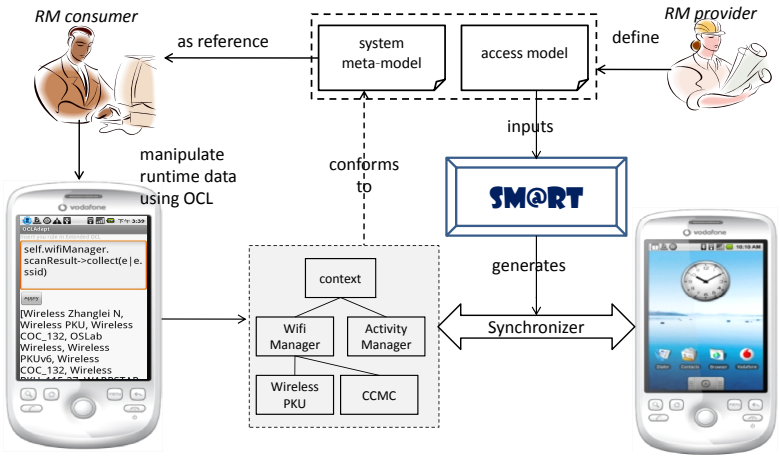


Fig. 2. Tool overview

target systems, like Android, we require the runtime model providers to define a *system meta-model* which specifies the types of the runtime data, and an *access model* which specifies how to manipulate the data through the API. From these two inputs, *SM@RT* automatically generates the synchronizer, which maintains a MOF standard runtime model for a running system instance, and ensures the causal connection between this model and the system's runtime data.

Our contributions can be summarized as follows.

- We propose that runtime models could facilitate the management of systems by enabling the use of existing model-driven techniques at runtime. We also identify the key requirements for such runtime models.
- We provide a generic synchronization solution between runtime models and system data, and based on this solution, we provide a generative tool to construct synchronizers for a wide class of systems.
- We successfully apply this tool on several systems, and undertake several experiments to utilize the provided runtime models for managing the systems. These case studies illustrate how the runtime models facilitate the management of systems by using model-driven tools, and how our SM@RT tool implement such runtime models.

The rest of the paper is structured as follows. Section 2 discusses the requirements of runtime models. Section 3 presents our SM@RT tool to implement such runtime models. Section 4 reports our case studies. Section 5 presents some related approaches and Section 6 concludes this paper.

2 Requirements of Runtime Models

In this section, we discuss what the runtime model should be like in order to facilitate the system management with the help of MDE techniques and tools.

We summarize the following three requirements, considering the feature of both runtime management and the MDE tools.

Standardized. First, the format of the runtime models should conform to some widely accepted modeling framework (the meta-meta-model and the exchange format), such as MOF, fractal, XML, etc. Standardized models provide the runtime model consumers a consistent basis for understanding and manipulating the data. Moreover, since many MDE techniques and tools are defined and implemented on specific modeling standards, they can be directly reused only if the model conforms to the same standard. As the OMG's Meta-Object Facilities (MOF) has became the most accepted standard, with rich tool support, in this paper we only consider the runtime models conforming to the MOF standard.

Explicitly defined. The types of runtime models should be explicitly defined by meta-models. Such meta-models provide an intuitive guidance and a strict constraint for runtime model consumers to understand and reconfigure the runtime models, and are also necessary reference for MDE tools to process the models. According to the MOF standard, a meta-model defines the types of model elements by the *classes*. For each class, the meta-model defines the data type of attributes that can be contained by the elements, and the potential relation between them and the elements of other classes.

Causally connected. Finally, we require the runtime models to have the *causal connection* with the running systems. The management agents monitor and reconfigure the system by reading and writing the model. The causal connection ensures that each time the management agent reads the model, it gets the information representing the current system state, and similarly, each time it writes the model, the information it writes causes the proper system change. Considering the Android example, if the device enters into the scope of a new Wi-Fi service, there will be a new `ScanResult` element appearing in the model immediately, so that the OCL query in Section 1 returns the ID of the new Wi-Fi service. Causal connection is an important feature of runtime models, which distinguishes them from the models used in design and development phases.

Notice that there are multiple levels for causal connection. The above requirement is just a basic one. Advanced usages of runtime models may require the model changes launched by the management agent would be stable as system evolves, or even require the model to hold some predefined constraints. But in this paper, we cares about the minimal requirement to enable MDE tools to be used for runtime management, and leave the advanced work as the task for "using the tool in a correct way".

3 The *SM@RT* Tool to Implement Runtime Models

We provide a generitive tool, the *SM@RT*, to help implement runtime models that satisfy the above requirements. Specifically, for a target system with a management API, the tool accepts a MOF meta-model defining the system data, and a description about the management API to access such data. Then it automatically generates a *synchronizer* for the target system, which represents

the system data as a MOF standard model conforming the system meta-model, and maintain the causal connection between the model and the system data.

3.1 Tool Input

To provide a runtime model for a specific system, we need the information about "what kind of data can be manipulated" in this system, and "how to manipulate them through the system's API". The former is defined by the MOF meta-model as discussed in Section 2. For the latter, we defined an API description language to specify how to access (invoke) the API to manipulate each type of the data.

The API access is described as code snippets annotated with their effects on the data. Look over the sample code in Section 1 for invoking the Android API. The first line tells us that from the root system element `this`, whose type is `Context`, how we can get its child named `wifiManager`. The above statement comprises three kinds of information for manipulating the system data, i.e., the manipulation target (an aggregation named `Context.wifiManager`), the manipulation type (`get`), and the action (Lines 1-2 in this code snippet). From this point of view, we define the access model for an API as follows.

$$AccItem : MetaElement \times Manipulation \longrightarrow Code$$

Here $MetaElement$ is the set of all the elements in the system meta-model (classes, attributes, etc.), $Manipulation$ is the set of 9 types of manipulations, including `getting` and `setting` attribute values, `creating` and `deleting` model elements, etc., and $Code$ is a piece of Java code [5].

3.2 Tool Output

The output of *SM@RT* is a "synchronizer" that maintains the causal connection between the runtime model and the running system.

The mechanism inside such synchronizers can be briefly described as "lazy and local refreshment". Specifically, the synchronizer maintains an in-memory MOF standard model, in the form of a set of Java objects implementing the `EObject` interface defined in Eclipse EMF. During runtime, the synchronizer keeps on listening to the external reading and writing operations on this runtime model. For reading operation, the synchronizer calculates what system data are required, collects the data via the management API, and refreshes or complements the model according to the collected data. Similarly, for a writing operation, the synchronizer identifies the modifications on the model, calculates the corresponding changes on the system, and invokes the API to implement the changes. For different kinds of operations (getting, setting, adding etc.) and their target meta-elements (classes, attributes, single or multiple valued associations), the calculation methods are different. We name these methods as the synchronization strategies. We summarized and designed a set of synchronization strategies covering all the potential combinations of operations and meta-elements, as presented in our previous work [5].

3.3 Generating the Synchronizer

SM@RT automatically generates the synchronizers from the API description. The tool has two parts, a *common library* and a *code generation engine*. The common library implements the generic solutions inside the synchronizers, such as maintaining the mapping between model elements and system parts, and the hard-coded synchronization strategies for different kinds of elements and different operations. The code generation engine generates the parts of the synchronizers which are specific to the target system, such as all the standard model operations (depending on the system meta-model) and the effective API invocations to manipulate each kind of system data (depending on the access model). We generate the model operations by directly reusing Eclipse EMF generator, and generate system operations according to the items defined in the access model, using the defined API-invoking code snippets as the body of the system operation. The generated operations follows a strict naming convention, so that the synchronization strategy know the semantical relation between model and system operations, automatically.

4 Demonstration

We demonstrate four case studies for *SM@RT*, using it to provide runtime models for four different target systems, including Android mobile systems, Eclipse SWT windows, Java classes and JOnAS JEE enterprise systems. We describe the Android case in detail, showing how to construct the system-model synchronizer, how to use the MDE tool (the OCL query engine in this case) upon the runtime model, and how the synchronizer works to maintain the runtime model.

4.1 The Android Case

Android is a mobile operating system developed by Open Handset Alliance[2]. It allows developers to write managed code in Java to manipulate (read or write) a device, by invoking the API of a set of Google-developed Java libraries.

Figure 3 shows the system meta-model we define for Android runtime data. In this demonstration, we care about the memory, connections, running tasks and Wi-Fi. We define each type of system data as a class, and define the relations between them as properties. For example, this meta-model tells us that from a root element in type of `Context`, we can first get its `wifiManager`, and then get the manager's `scanResult` to enumerate all the Wi-Fi signals. For each scanned signal, we can get its attributes like `ssid`, `frequency`, etc. This meta-model is not only an input to our synchronizer, but also a guidance for using the runtime model (like writing the OCL query) and a reference of the MDE tool (like the OCL engine).

Figure 4 shows an excerpt of the access model, which defines how to get a `Context`'s `wifiManager`. The annotations (keywords starting with "@") indicates the constitution of the item, i.e. a `AccItem` containing a `MetaElement`,

[2] `http://www.android.com`

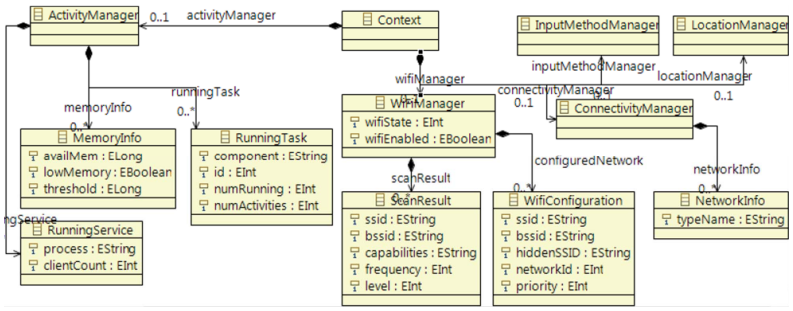


Fig. 3. Tool overview

```
1  @AccItem @MetaElement=Context::wifiManager
2  @Manipulation=Get @Code=@Begin
3    $sys::result=($sys::type)$sys::this
4      .getSystemService($sys::type.WIFI_SERVICE);
5  @End @EndAccItem
```

Fig. 4. Access model for Android

`Manipulation` and a `Code` fragment, and whose values are defined on the right hand side of the equal signs. Inside the code fragment, we define a piece of Java code to say that to get a `wifiManager` (`$sys::result`) from a `Context` instance (`$sys::this`), we should invoke a method named `getSystemService` with a parameter `Context.WIFI_SERVICE`. The entire access model contains 95 items like this, with 431 lines of code (including the structural lines like `"@AccItem"`).

We use these two inputs to generate the synchronizer. The generation result is in the form of Java source code. We compile it as an Android package, and deploy it onto an Android supported mobile phone, the "HTC Magic (G2)".

The generated synchronizer allow the device users to use OCL for querying the device data. Figure 5 shows the snapshots of four scenarios for executing OCL rules on Android. For the first scenario, we want to list the IDs of all the Wi-Fi signals available for the device. Initially, we know that the root element (we refer to it as `self` in the OCL rules) is in type of `Context`. Then we check the system meta-model and find that `Context` has an association named `wifiManager`. The target class `WifiManager` has an multiple-valued association named `scanResult`. And finally, the target class `ScanResult` has an attribute named `ssid`. According to terminology of Wi-Fi technique, we know that we can list the signal IDs by querying out the values of these `ssid`s. We input the OCL rule as shown in Figure 5(a), click the button, and the result is printed under the button. Similarly, Figure 5(b) shows how we print the detailed information of the first Wi-Fi channel. Figure 5(c) shows how we calculate the total number of clients registered on all the running services. Figure 5(d) shows a relatively complex query: We want to see what services have more than one clients listening to them. The OCL rule means "getting the running services, selecting the ones from them

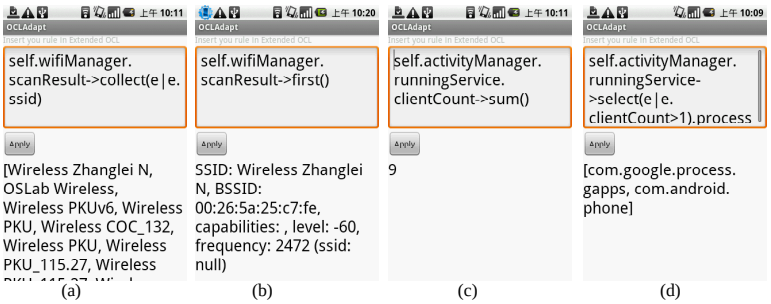


Fig. 5. Android screen shots

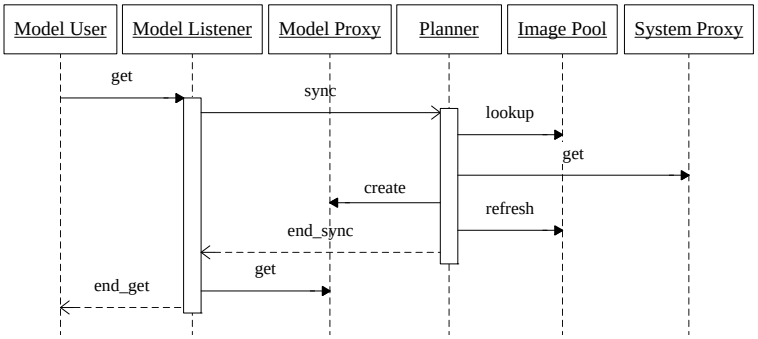


Fig. 6. A sample behavior of the synchronizers

whose client count is greater than 1, and finally retrieving the process name of the selected running services".

The above scenarios are implemented by directly using the Eclipse OCL engine. We compile the OCL engine on Android platform, and deploy it on the same device. After the user clicking the `Apply` button, the GUI retrieves the inputted OCL rule, instantiates a root element in type of `Context` from the synchronizer, and invoke the `evaluate` method of the OCL engine using this root element and the OCL rule. During the execution of the OCL engine, it will manipulate the model from this root element, by means of standard model invocation defined by EMF. And in the same time, the synchronizer breaks the invocation, and synchronizes the model with system on-demand.

Figure 6 illustrates how the synchronizer works, when we evaluate the OCL query in Section 1 on the runtime model as shown in Figure 1. Each life-line in this sequence diagram represents a component that constituting the synchronizer. At first, the initial model only contains one root model element, in type of `Context`. Following the query, the interpreter first retrieves the root's child named `wifiManager`, by invoking `get` on the model. The *model listener* interrupts this invocation, and asks the planner to perform synchronization. According to the synchronization strategy for "getting single-valued aggregation" [5], the *planner* first looks up the *image pool* and finds that this root element

corresponds to a context object provided by the Android API. Then the *planner* performs `get` on this context object. The logic of this system `get` operation is just the one defined in the description item shown in Figure 1. This `get` operation returns a system object which points to the Wi-Fi manager, and the *planner* `create`s a model element in type of `WifiManager` as an image for this object, `refresh`es the image pool, and notifies the *model listener* about the `end` of this synchronization. The *model listener* then invokes `get` on the model again, and returns the newly created model element as a result to the interpreter. Following the remaining parts of the OCL query, the interpreter performs `get` operations successively to obtain the `WifiManager`'s `scanResult`, and to obtain these results' `ssid`s. The behavior of the synchronizers is similar as shown before.

4.2 The Eclipse-SWT Case

In this case, our target systems are SWT-based Eclipse UI parts, which could be "views", "editors" or "dialogs" running on an Eclipse platform. Such UI parts, also known as `Shell`s according to the SWT terminology, are constituted of a set of `Control`s, like the `Label`s for presenting information, the `Text` fields for inputting texts, the `Button`s for triggering commands, etc. Each `Control` has its own configurations which can be retrieved and updated at runtime, such as the presented text, the background color, etc. The `Control`s and their configurations form the runtime data of such `Shell`s. The main idea of this case study is to provide runtime models for Eclipse windows, so that developers could reconfigure the windows intuitively *at runtime*. That means developers do not need to completely decide the appearance of the windows, and reconfigure the window at runtime. Moreover, this configuration is simply editing models, through visualized model editors. This is a prototype for "design at runtime", and may be useful in customizable GUIs or WYSIWYG GUI development.

The foreground image of Figure 7 presents a simplified version of the system meta-model. We defined three common types of controls, and defined some typical attributes for them. The background snapshot illustrates how to use the runtime model. The snapshot is an Eclipse platform, with the target system (the bottom part, an Eclipse "view") and the runtime model (the top part, a model opened in a tree-based visual model editor), together. The model elements reflect the controls in the window, and their attributes reflect the controls' configurations. We change the system by typing "Hi" on the text field, the model element's `text` attribute changes instantly. We can edit the model to manipulate the system: We change the `background` of the first `Label` into "red", and then the color of the system label changes automatically. Finally, we add a new model element in the type of `Button`, and a new button appears in the window .

4.3 The Java Class Structure Case

This case is a reproduction of the Jar2UML tool[3], which reflects the class structure in a Jar file as a UML model. We utilized the UML meta-model (defined

[3] `http://ssel.vub.ac.be/ssel/research/mdd/jar2uml`, a use case of MoDisco

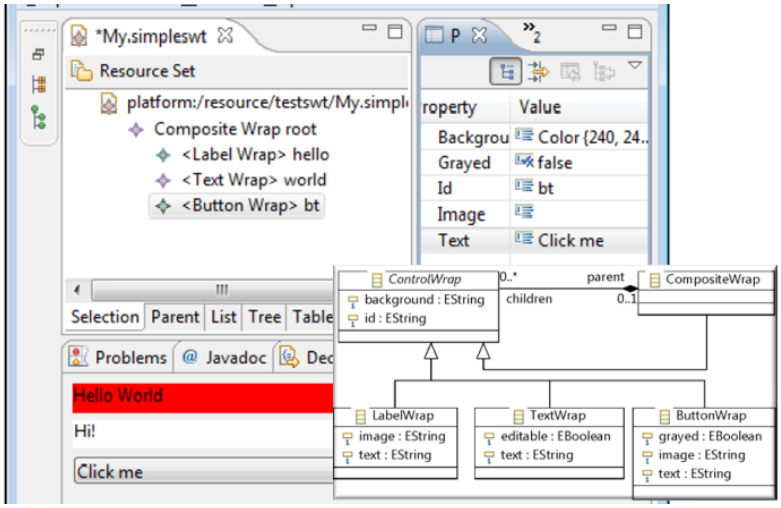


Fig. 7. Visual management of an SWT window

by Eclipse UML2[4]) as our system meta-model, and define the API provided by the BCEL library[5] for analyzing Java binary code. We used Eclipse UML2 tools to visualize the reflected UML model as a class diagram.

4.4 The JOnAS Case

Our last case study is to equip a JOnAS JEE application server[6] with runtime model. This model reflects the inner structure (e.g. what applications and EJBs are deployed), the configuration (e.g. the size of data source's connection pool), and the state (e.g. the number of EJB instances) of a running JOnAS server.

The system meta-model defines all the 21 types of MBeans supported by JOnAS, including EJBs, applications, middleware services, etc. The API description specifies how to manipulate these elements and their properties through the JMX API provided by JOnAS. We deploy the generated synchronizer on a JOnAS server with a Java Pet Store application deployed on it, and utilize Eclipse GMF[7] to visualize the runtime model maintained by the synchronizer. The graphical model editor based on GMF can be used as a graphical JOnAS management tool: We can see the inner structure of the current JOnAS server, deploy new applications or EJBs by adding model elements, and check the system elements' current states and modify their configurations. We also use the QVT transformation to synchronize this runtime model with a software architecture model in C2 style, reproducing the architecture-based runtime evolution

[4] http://www.eclipse.org/uml2
[5] Byte Code Engineering Library, http://jakarta.apache.org/bcel/
[6] http://jonas.ow2.org
[7] http://www.eclipse.org/modeling/gmf

Table 1. Summary of case studies

system	API	meta-model	access model	generated	compared	tools
		elems	*items/LOC*	*LOC*	*LOC*	
Android	Android	87	95/431	21732	-	OCL
Eclipse	SWT	43	36/220	11290	-	EMF
Java class	BCEL	29	13/109	10518	3108	UML2
JOnAS	JMX	305	47/270	37263	5294	GMF, QVT

proposed by Oreizy et al. [6]. The details of this case study could be found in our earlier work [7]

4.5 Summary and discussion

Table 1 summarizes the case studies. For each case, we list the target system and its management API, the number of elements in the system meta-model, and the number of items in the API description. After that, we list the size of the generated synchronizer. For the last two cases, we also list the sizes of the hand-written programs with the equivalent capabilities, which are developed by ourselves or other developers [4]. Finally, we list the model-driven techniques we applied upon the runtime model to implement runtime data manipulation.

These case studies illustrate the following aspects of *SM@RT*.

- *Feasibility.* The case studies covers a wide class of systems, from enterprise systems to mobile devices.
- *Efficiency for development.* It is not a hard task to define the system meta-model and API description, comparing with the multi-time-larger generated code (which approximately reflects the work required to support runtime model) and the actual manual effort to realize runtime models.
- *Effectiveness.* The generated synchronizers enable the existing MDE tools to be directly used for runtime management. In particular, we use OCL for runtime data querying, and use GMF/EMF, and UML2 for different purpose of visualization and manipulation of runtime data. All the MDE tools are directly used upon the synchronizers.

5 Related Work

There are many research approaches towards runtime models, according to a recent survey [3] and the annual workshops [2]. As an emerging topic, many of these approaches focus on how to utilize the runtime models, but not how to implement runtime models on existing systems, which is exactly the target of *SM@RT*. We share the similar idea with Sicard et al. [1] and the MoDisco project [4], i.e. wrapping the systems' APIs to reflect runtime data as standard models. Their *wrappers* or *discoverers* play the similar role as our *synchronizers*. The difference is that they require runtime model providers to manually develop the wrappers or discoverers, while our *SM@RT* tool automatically generates the

synchronizers. Our API description language shares the similar idea as feature-based code composition [8]. The common synchronization mechanism roots in the earlier research on reflective middleware [9], and the model synchronization approach towards runtime management [10].

6 Conclusion

In this paper, we focus on a specific usage of runtime models, i.e., facilitating the runtime management by enabling the use of existing MDE techniques and tools at runtime. We discuss the requirements of such runtime models, and present our *SM@RT* tool to help on providing them. We evaluate our idea and the tool through a set of case studies on a wide range of target systems.

ACKNOWLEDGEMENT This work is supported by the National Basic Research Program of China under Grant No. 2011CB302604, the National Natural Science Foundation of China under Grant No. 60933003, 60873060; the High-Tech Research & Development Program of China under Grant No. 2009AA01Z116, and the National S&T Major Project under Grant No. 2009ZX01043-002-002, the EU Seventh Framework Programme under Grant No. 231167.

References

1. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: ICSE. (2008) 101–110
2. Bencomo, N., Blair, G., France, R.: Summary of the workshop Models@run.time at MoDELS 2006. In: Satellite Events at the MoDELS 2006 Conference, LNCS,. (2006) 226–230
3. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering, in ICSE. (2007) 37–54
4. Bruneliere, H.: The MoDisco Project, `http://www.eclipse.org/gmt/modisco/`
5. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating synchronization engines between running systems and their model-based views. In: MoDELS Workshops 2009, LNCS 6002. (2009) 140–154
6. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE. (1998) 177–186
7. Huang, G., Song, H., Mei, H.: Sm@rt: Applying architecture-based runtime management into internetware systems. Int. J. Software and Informatics **3**(4) (2009) 439–464
8. Antkiewicz, M., Czarnecki, K.: Framework-specific modeling languages with round-trip engineering. In: MoDELS. (2006) 692–706
9. Huang, G., Mei, H., Yang, F.: Runtime recovery and manipulation of software architecture of component-based systems. Auto. Soft. Eng. **13**(2) (2006) 257–281
10. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental model synchronization for efficient run-time monitoring. In: MoDELS Workshops. (2009) 124–139

Run-Time Evolution through Explicit Meta-Objects

Jorge Ressia, Lukas Renggli, Tudor Gîrba and Oscar Nierstrasz

Software Composition Group
University of Bern
Switzerland
http://scg.unibe.ch

Abstract. Software must be constantly adapted due to evolving domain knowledge and unanticipated requirements changes. To adapt a system at run-time we need to reflect on its structure and its behavior. Object-oriented languages introduced reflection to deal with this issue, however, no reflective approach up to now has tried to provide a unified solution to both structural and behavioral reflection. This paper describes Albedo[1], a unified approach to structural and behavioral reflection. Albedo is a model of fined-grained unanticipated dynamic structural and behavioral adaptation. Instead of providing reflective capabilities as an external mechanism we integrate them deeply in the environment. We show how explicit meta-objects allow us to provide a range of reflective features and thereby evolve both application models and environments at run-time.

1 Introduction

Classical software development plays out like a finite game with fixed rules and boundaries. However, evolving software systems are rather an infinite game without fixed rules or boundaries [2]. Large systems not only evolve in the development phase, but also at run-time. Development itself is part of the infinite game of the system. The system may continue to evolve while running.

To enable change at run-time, a system must be self-aware and be able to fully reflect on itself [14]. A reflective system provides a description of itself that can be queried (introspection) and changed (intercession) from within. Consequently the system can reflect on itself and can change its structure and behavior. However, this is not enough to support full run-time evolution of models. To evolve a model at run-time it is necessary to access the dynamic representation of a program, that is, the operational execution of the program. This is called behavioral reflection, pioneered by Smith [18, 19] in the context of Lisp. A reification not related to the structure of the language might be required, for example, a message send.

[1] In astronomy, albedo is the proportion of the incident light or radiation that is reflected by a surface, typically that of a planet or moon [The Oxford Dictionary of English]

As an example, let us look at a financial model and how to make it available to an external audit system. Such a model typically contains sensitive information and certain properties should not be accessible and modifiable by everybody. We therefore need a representation of the model that has the same behavior except for certain restricted accessors. To achieve this, we need to change the model and integrate contextual security checks. However, such a transformation is not straightforward and might add unnecessary complexity to the application code. What we need is to modify the system so that security concerns can be quickly changed to adapt to new requirements. A reflective system where we can express from the inside these structural constraints to access or modify certain state will solve our problem.

It can be argued that this problem can be solved by using Aspect-Oriented Programming (AOP) [10]. AOP provides a general model for modularizing cross cutting concerns. Join points define points in the execution of a program that trigger the execution of additional cross-cutting code called advice. Join points can be defined on the run-time model (i.e., dependent on control flow). Although AOP is used to introduce changes into software systems, the focus is on cross-cutting concerns, rather than on reflecting on the system. Modeling the changes in the financial problem through AOP will deliver a set of join points which have a larger semantic gap to the actual requirement than using the reifications of the system. The reflective approach is better suited for continuous evolution since join points are harder to reflect upon.

The first object-oriented language to propose a clean reflection approach was Smalltalk-80 [7] by introducing meta-classes. ObjVlisp [3] and Classtalk [1] followed this line of research. Meta-classes define the internal structure and behavior of a class. There can be only one meta-class per class. There is a second approach to reflection in object-oriented languages introduced by Pattie Maes in 3-KRS [11, 12]. This approach states that each object is related to one meta-object. A meta-object specifies the structure of the object and how this object handles messages. Maes also benefited from the work of Kiczales *et al.* [9] on meta-object protocols (MOPs) over the CLOS language, an object-oriented extension of Lisp. The protocol of a meta-object encodes the behavior of the language. If the MOP is changed the language is changed as well.

To solve the financial model security issue we can either remove methods from classes or from meta-objects. Depending on the reflection approach we will be affecting one object or all instances of a class. But the key point to highlight is that we are dealing with structural elements only.

Regardless of the differences between the reflection approaches, they are built on structural reifications of the concepts that form the language (classes, objects, methods, instance variables, etc.), however, this structural approach does not solve all reflection problems. Assume there is a new requirement which says that every time the interest rate of a financial instrument is changed a message should be sent to the central audit system. Thinking in terms of classes, objects and methods is not enough to solve this requirement.

Following Smith's work on behavioral reflection McAffer [13] developed CodA, a system that approaches reflection from the point of view of operational decomposition. He proposed to separate the description of the computational behavior of an object from that of its base language structure. This approach divides the execution of an object into basic operations (*e.g.,* message send and receive, state access, object creation and deletion, etc.).

Iguana [8] further improved this approach by providing a mechanism, known as fine-grained MOP, to allow multiple reflective object models to coexist. For example, one object can have a distributed object model while other objects in the system have a centralized object model. Although structural approaches have the advantage of organizing the meta-level in terms of concepts known to the user like classes and methods, it is hard to integrate new concepts that are not represented in the language. This is important since it allows the language to evolve beyond its predefined structure. This approach provides a way of integrating reified concepts that are not originally part of the language through its operations. There are seven reification categories: object creation and deletion; message sends, receive and dispatch; and state read and write. Iguana was later ported to Java [15, 16]

The audit requirement can be solved by specifying that a message is sent to the central audit system every time a state write event related to the financial instrument interest rate is issued.

Up until now, no approach has tried to provide a unified solution to these two domains of reflection: structural and behavioral. In this paper we present Albedo, a model of fine-grained unanticipated dynamic structural and behavioral adaptation. Instead of providing reflective capabilities as an external mechanism we integrate them deeply in the environment. We show, by modeling meta-objects explicitly, how we can extend the environment from both a structural and behavioral standpoint.

The most relevant characteristics of Albedo are:

- It provides a meta-object approach to reflection.
- The meta level can be organized with language concepts and operational decomposition.
- The fine-grained MOP allows us to control the scope of the change.

Outline. Section 2 explains how the model behind the Albedo environment solves both structural and behavioral requirements. In Section 3 we present the internal implementation of our solution in the context of Smalltalk. Section 4 presents a couple of structural and behavioral reifications and how they are modeled with meta-objects. In Section 5 we summarize the paper and discuss future work.

2 Albedo Approach

The Albedo mechanism is placed at the center of the system where every language construct is expressed in terms of an Albedo meta-object. In this section

we show the two building blocks of Albedo and how they can solve the previously presented problems.

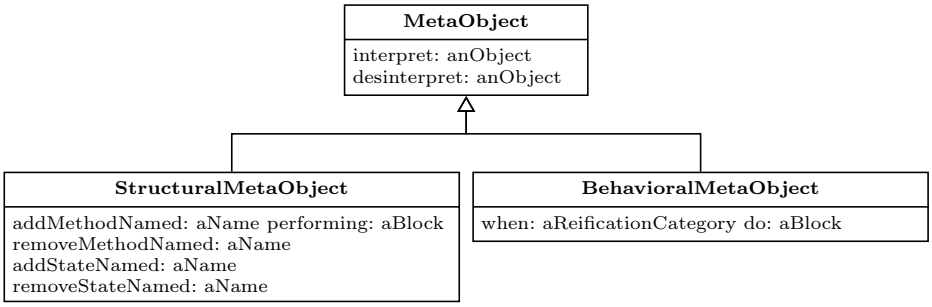


Fig. 1. Meta-Objects class diagram.

Figure 1 shows a simplified meta-object class hierarchy diagram. The meta-object abstraction is reified and it has the responsibility to adapt objects. The idea is that meta-objects specify how an object meta-level should work.

2.1 Structural Meta-object

The StructuralMetaObject abstraction reifies the meta-object responsible for modeling the structures of a program. An object-oriented program's canonical structures are objects and messages. In class-based languages, classes have too many responsibilities. They are used to model an abstraction, provide an instance creation mechanism, define the messages the instances know how to answer, and provide a template that subclasses can extend. In prototype-based languages the idea of generalization does not exist and new abstractions are built by delegating to other abstractions or objects. Traits are an example of a structural meta-object, created for sharing behavior. A trait [17] is a composable unit of behavior that can be shared among objects. If several objects share a trait then they all will be able to understand the messages defined in the trait. The StructuralMetaObject abstraction provides the means to define meta-objects like classes, prototypes and traits. New structural abstractions can be defined to fulfill some specific requirement. For example we can define the concept of traits as we discuss in Section 4.

A StructuralMetaObject acts on the basic structural units of an object-oriented language which are messages, objects and the object state. The responsibilities of a StructuralMetaObject are:

– *Adding a method.* A new method is added to the object. A name and the source code is provided. When the object receives the message it executes the compiled source code. The source code compilation is performed when the object is associated with a meta-object. If there is a compilation error the meta-object association is rolled back.

– *Removing a method.* The adapted object will not understand a particular message any more.
– *Replacing a method.* The method will have another behavior. Source code or a closure can be provided.
– *Adding state.* The addition of new state to an object allows the user to add methods that use that state.
– *Removing state.* Specific state is removed.

These responsibilities are modeled as meta-actions. A StructuralMetaObject can be defined as a particular set of these meta-actions that structurally adapt a particular object. When an object is associated to a meta-object these actions are executed and the object is adapted accordingly.

Let us consider the financial domain and pick a financial instrument which has to be audited by an external company. The auditors do not want to see a report, they want to see the real system and interact with it. This financial instrument is still active and should become due in some months. Financial instruments can be renegotiated, changing the due date and recalculating the taxes under a new contract. We want to hide this kind of behavior from the audit committee to assure the integrity of the instrument.

The following code fragment shows how this scenario can be achieved with Albedo:

```
1 anImmutableBehavior := StructuralMetaObject default.
2 anImmutableBehavior removedMethodNamed: #renegotiate.
3 anImmutableBehavior boundTo: aFinancialInstrument.
```

Listing 1. Making a financial instrument immutable regarding the renegotiation behavior.

First, we get the default structural meta-object (line 1) whose responsibility is to define the allowed behavior for immutable financial instruments. Then we remove the renegotiate method from the meta object (line 2). Finally, the meta-object is associated with the financial instrument (line 3). This association triggers the adaptation of the objects thus removing renegotiate from the set of messages understood by the financial instrument. If this message is sent to the financial instrument an error is thrown, however, if this message is sent to any other financial instrument, even to another instance of the same class, the original behavior is preserved.

Structural meta-objects deal with the definition of meta-level structural reifications. How and when they are introduced at run-time is the job of the behavioral meta-object.

2.2 Behavioral Meta-object

The BehavioralMetaObject abstraction reifies the meta-object responsible for modeling the dynamic representation of a program. By dynamic representation we refer to the language's dynamic reifications. This abstraction coresponds to the work done in Iguana and later used by McAffer in Coda. As McAffer

pointed out, the system is modeled as a set of operations whose occurrences *"can be thought of as events which are required for object execution"* [13].

To dynamically adapt the behavior of an object we need to describe what we would like to do and when. To specify what we would like to apply we delegate to a specific meta-object with the responsibility of managing an event. We specify when it should be adapted by using a computational event in the execution of a program, for example, the creation of an object, sending a message, *etc*. A set of canonical events models the basic operations known as *dynamic reification categories*. These are not the only reifications possible. New dynamic reifications can be defined, the only requirement being to specify when they should be triggered.

The dynamic reification categories are:

- Object creation
- Object deletion
- Message send
- Message receive
- Message dispatch
- State read
- State write

We selected these categories following the Iguana approach. With these basic categories we are capable of adapting an object's behavior regarding operations that are executed over the language building blocks.

For example, let us consider a new requirement which specifies that every time the interest rate of a financial instrument is changed, a message should be sent to the central audit system.

```
1 aMethodLookupMetaObject := BehavioralMetaObject default.
2 aMethodLookupMetaObject
3          when: (StateRead new)
4          do: [:owner :state |
5                CentralAuditSystem
6                   interestRateOf: owner
7                   changedTo: state].
8 aMethodLookupMetaObject boundTo: aFinancialInstrument.
```

Listing 2. Dynamically reifying method lookup for a financial instrument.

In line 1 the default behavioral meta-object is obtained. Lines 2–7 show the usage of reification categories to define when the meta-behavior should to happen. In this case the central audit system is informed of a change in the interest rate of an object. In line 8 the object is associated to the meta-object, and from this point on the new meta-behavior is obtained every time the interest is changed.

3 Implementation

Albedo is the prototype of our meta-object model approach built in Pharo Smalltalk[2]. Reflectivity [4] was used for dynamic adaptation. Reflectivity provides *unanticipated partial behavioral reflection* at the sub-method level,

[2] `http://pharo-project.org/`

using ASTs rather than source code or bytecode as underlying model of the software. We decided to use Reflectivity since *partial behavioral reflection* as pioneered by Reflex [20] is particularly well-suited for dynamic reifications because it supports a highly selective means to specify where and when to reify an abstraction in the system. *Partial behavioral reflection* offers an even more flexible approach than pure behavioral reflection. The key advantage is that it provides a means to selectively trigger reflection, only when specific, predefined events of interest occur.

The core concept of the Reflex model of partial behavioral reflection is the *link*. A link invokes messages on a meta-object at occurrences of marked operations. The attributes of a link enable further control of the exact message to be sent to the meta-object. Additionally, an activation condition can be defined for a link which determines whether or not the link is actually triggered.

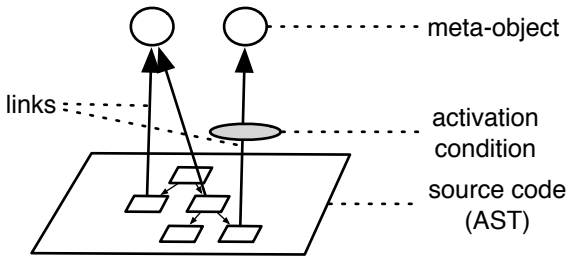


Fig. 2. The reflex model.

Links are associated with AST nodes. Subsequently, the system automatically generates new bytecodes that take the link into consideration the next time the method is executed.

Reflectivity integrates itself into Pharo by using the Reflective Methods abstraction. A Reflective Method knows the AST of the method it represents. In Pharo classes are first class objects that are available to any program. They have an instance variable named methodDict which holds an instance of MethodDictionary – a special sub-class of Dictionary. All methods of a class are stored in its method dictionary. The VM directly uses the class objects and their method dictionary when performing message sends. Normally, only instances of CompiledMethod are stored in the method dictionary of a class but Pharo allow us to store any kind of object there. The VM recognizes objects that are not instances of CompiledMethod and instead of executing bytecode the VM sends run:with:in: to the object stored in the method dictionary. When a reflective method receives this message it processes the adaptations specified in the meta-object on the AST and generates a new compiled method which is eventually executed. If no adaptation is present the reflective method caches the compiled

method for performance savings. Figure 3 visualizes this relationship between a class, the method dictionary and the methods.

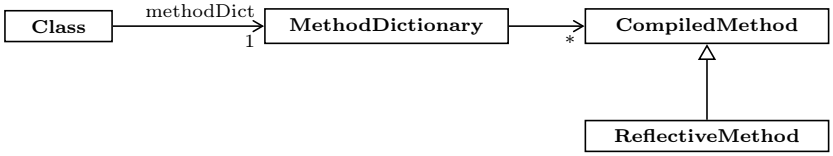


Fig. 3. Reflective Methods in Method Dictionaries

Reflectivity was conceived as an extension of the Reflex model of *Partial Behavioral Reflection* [20]. Reflex was originally realized with Java. Therefore, our approach can in be implemented in a more static mainstream language like Java. The reason for choosing Smalltalk and Reflectivity for this work is that it supports *unanticipated* use of reflection at runtime [4] and is integrated with an AST based reflective code model [5]. A Java solution would likely be more static in nature: links cannot be removed completely (as code cannot be changed at runtime) and the code model would not be as closely integrated with the runtime of the language.

4 Discussion

Albedo is not only useful for evolving systems at run-time but it is also useful for developing new language features. By modeling meta-objects explicitly we can extend the environment from both a structural and behavioral standpoint. In this section we discuss the language impact of having a system that can be extended from a structural and behavioral point of view. Next, we introduce two examples of reifying language features to show how the language itself can be modified.

4.1 Traits Example

A trait [17] is a composable unit of behavior that can be shared among objects. If several objects share a trait then they all will be able to understand the messages defined in the trait.

Let assume that we want all financial instruments to share the same behavior. For example, suppose we want to provide a common implementation for the renegotiation feature. Furthermore we do not want to impose a class hierarchy structure on all financial instruments to introduce this feature, but instead keep the possibility to assign the feature dynamically to an instrument. We can easily fulfill these needs by defining the feature as a trait, however if the host language does not implement traits we cannot introduce this feature as we would like.

Albedo provides a mechanism to define dynamically the trait abstraction thus modifying the language model.

```
1 aTrait := StructuralMetaObject default.
2 aTrait addMethodNamed: #regenerate performing: 'self recalculateTaxes.
3                                 self recalculateDates'.
4 aFinancialInstrument := aFinancialInstrumentClass new.
5 anotherFinancialInstrument := aFinancialInstrumentClass new.
6 aTrait boundTo: aFinancialInstrument.
7 aTrait boundTo: anotherFinancialInstrument
```

Listing 3. Building the trait abstraction with structural meta-objects.

First, we introduce the trait abstraction itself as a structural meta-object in line 1. The message renegotiate is defined in line 2 for this trait, its behavior is to recalculate taxes and dates. By using the existing Class abstraction defined with meta-objects we create two financial instruments in lines 4-5. The class abstraction is built using a structural meta-object and adding a message new that creates an instance of a class. The class is bound to the instance as a meta-object. Finally, we associate the trait as the meta-object to both objects thus making them capable of answering the message renegotiate.

A trait is defined as a StructuralMetaObject. However, by definition, traits should not have state. To achieve this we need to remove the possibility of adding state in the trait structural meta-object.

```
1 aTraitBehavior := StructuralMetaObject default.
2 aTraitBehavior removedMethodNamed: #addStateNamed: .
3 aTraitBehavior removedMethodNamed: #removeStateNamed: .
4 aTraitBehavior boundTo: aTrait.
```

Listing 4. Making traits stateless.

We first define another structural meta-object called TraitBehavior (line 1). This abstraction has the responsibility of defining which are the messages a trait meta-object is capable of answering. In lines 2–3 both state-related messages are removed from the trait behavior definition. Finally, in line 4 the TraitBehavior is set as the meta-object of the trait meta-object defining its responsibilities.

4.2 Method Lookup Example

The method-lookup reification defines the process that specifies which method should be executed when an object receives a message. To reify method-lookup in some languages it is necessary to perform complex computations, apply method wrappers, manipulate method dictionaries [6] or adapt byte-code.

Most languages, including Java and Smalltalk, do not reify method lookup. Being able to change the way a message is mapped to a method has many applications, for example test coverage, benchmarking and logging. Moreover, class-based languages impose a method lookup strategy that follows the class hierarchy. In some cases we need to have a strategy different from the traditional method lookup. We might not be so sure about how the class hierarchy should be organized. Here we have to provide a different strategy for matching a message to a method. For example, a financial instrument needs to reuse the behavior

defined in another financial instrument which is not its super-class. We are not yet sure how the hierarchy should be reshaped, or if we should use composition instead of inheritance. So we decide to provide a new method lookup strategy to the financial instrument for checking first if the method is defined in the second financial.

To implement method lookup a two-level meta-object structure is required as shown below.

```
1  aMethodLookupBehavior := StructuralMetaObject default.
2  aMethodLookupBehavior addStateNamed: #strategy.
3  aMethodLookupMetaObject := BehavioralMetaObject default.
4  aMethodLookupMetaObject
5          when: (MessageReceived new)
6          do: [:receiver :message :args |
7                strategy
8                   resolve: message
9                   for: receiver
10                  with: args ].
11 aMethodLookupBehavior boundTo: aMethodLookupMetaObject
12 aMethodLookupMetaObject boundTo: aFinancialInstrument.
```

Listing 5. Reifying method lookup with behavioral meta-objects.

The method lookup meta-object itself is a behavioral meta-object that reifies a message received by delegating to a predefined strategy. In lines 1–2 we define the structural meta-object and introduce a strategy instance variable. This is a meta-meta-object named aMethodLookupBehavior that specifies that the interpreted object will have a strategy instance variable. In lines 4–10 the method lookup meta-object is defined and the MessageReceived behavioral reification category is used to adapt how the method lookup should be resolved. The resolution of the lookup is delegated to the strategy active at run-time. The strategy can be changed at any point in time, thus delivering different behavior for the same object receiving the same message. Finally, aMethodLookupBehavior is associated to the meta-object of aMethodLookupMetaObject. Any object that is associated with aMethodLookupMetaObject will change the way the method lookup is resolved for that particular object.

5 Conclusion

In this paper we have introduced Albedo, a unified approach to behavioral and structural reflection. Albedo is a reflective model and environment for dynamically defined unforeseen language feature reifications. Even the most basic constructs of a language are expressed in terms of it. We have shown how different structural and behavioral meta-objects like traits and method lookup can be modeled with this environment.

What is more, we provide the means to dynamically define or modify abstractions thus eliminating the limitations of reflective models. This allows us to redefine static or run-time abstractions and manipulate their structure and behavior thus helping in the evolution of run-time models.

As future work we would like to analyze the performance impact of Albedo. We also would like to analyze various use cases to further validate the model. We would like to further research the composition of different meta-objects.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010) and CHOOSE, the Swiss Group for Object-Oriented Systems and Environments. We also thank Mircea Lungu and Fabrizio Perin for their feedback on earlier drafts of this paper.

References

1. Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 419–432, October 1989.
2. James P. Carse. *Finite and Infinite Games — A Vision of Life as Play and Possibility*. Ballantine Books, 1987.
3. Pierre Cointe. Metaclasses are first class: the ObjVlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 156–167, December 1987.
4. Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.
5. Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.
6. Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.
7. Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.
8. Brendan Gowing and Vinny Cahill. Meta-object protocols for C++: The Iguana approach. Technical report, AAA, 1996.
9. Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
10. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.
11. Pattie Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, January 1987.
12. Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.
13. Jeff McAffer. Meta-level programming with coda. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag.
14. Oscar Nierstrasz, Marcus Denker, Tudor Gîrba, Adrian Lienhard, and David Röthlisberger. Change-enabled software systems. In Martin Wirsing, Jean-Pierre Banâtre, and Matthias Hölzl, editors, *Challenges for Software-Intensive Systems and New Computing Paradigms*, volume 5380 of *LNCS*, pages 64–79. Springer-Verlag, 2008.
15. Barry Redmond and Vinny Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for java. In *Proceedings of European Conference on Object-Oriented Programming, workshop on Reflection and Meta-Level Architectures*, 2000.

16. Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002.
17. Nathanael Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Bern, February 2005.
18. Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. thesis, MIT, Cambridge, MA, 1982.
19. Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of POPL '84*, pages 23–3, 1984.
20. Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

A Model-Driven Approach to Graphical User Interface Runtime Adaptation

Javier Criado[1], Cristina Vicente-Chicote[2], Nicolás Padilla[1], and Luis Iribarne[1]

[1] Applied Computing Group, University of Almeria, Spain
{javi.criado,npadilla,luis.iribarne}@ual.es
http://www.ual.es/acg
[2] Department of Information Technology and Communications, Technical University of Cartagena, Spain
cristina.vicente@upct.es
http://www.dsie.upct.es/personal/cristinav/

Abstract. Graphical user interfaces play a key role in human-computer interaction, as they link the system with its end-users, allowing information exchange and improving communication. Nowadays, users increasingly demand applications with adaptive interfaces that dynamically evolve in response to their specific needs. Thus, providing graphical user interfaces with runtime adaptation capabilities is becoming more and more an important issue. To address this problem, this paper proposes a component-based and model-driven engineering approach, illustrated by means of a detailed example.

Keywords: runtime model adaptation, model transformation, graphical user interface

1 Introduction

Graphical User Interfaces (GUIs) play a key role in Human-Computer Interaction (HCI), as they link the system with its end-users, allowing information exchange and improving communication. Nowadays, users increasingly demand "smart" interfaces, capable of (semi-)automatically detecting their specific profile and needs, and dynamically adapting their structure, appearance, or behaviour accordingly.

GUIs are increasingly built from components, sometimes independently developed by third parties. This allows end-users to configure their applications by selecting the components that provide them with the services that better fit their current needs. A good example of this is iGoogle, as it provides end-users with many gadgets, allowing them to create personal configurations by adding or removing components on demand.

Following this trend, the proposal presented in this paper considers GUIs as component-based applications. Furthermore, it considers the components integrating GUIs software architectures at two different abstraction levels: (1) at design time, components are defined in terms of their external interfaces, their

internal components (if any), and their visual and interaction behaviours, while (2) at runtime, the former abstract components are instantiated by selecting the most appropriate Commercial-Off-The-Shelf (COTS) components (i.e., those that better fit the requirements imposed both by the abstract component and by the global GUI configuration parameters) from those available in the existing repositories.

Our proposal does not only rely on a component-based approach but also, and primarily, on a Model-Driven Engineering (MDE) approach. As detailed in the following sections, we propose a GUI architecture description meta-model that enables (1) the definition of component-based abstract GUI models at design-time, and (2) the runtime evolution (be means of automatic model-to-model transformations) of these architectural models according to the events detected by the system. The instantiation of these abstract models at each evolution step is out of the scope of this paper.

The remainder of the article is organized as follows. Section 2 reviews related works. Section 3 describes the proposed approach and its constituting elements, namely: the proposed GUI architecture meta-model, and a set of model transformations enabling runtime GUI adaptation. In order to illustrate the proposal, a GUI model evolution example is also described in detail in this section. Finally, Section 4 draws the conclusions and outlines future works.

2 Related Work

There are many model-driven approaches in the literature for modelling user interfaces, as detailed in [1]. Some of them use a MDE perspective for web-based user interfaces [2]. However, in most cases, models are considered static entities and no MDE technique is applied to add dynamism, for instance, using model transformations.

Model transformations enable model refinement, evolution or even, automatic code generation. In [3], the authors investigate the development of plastic user interfaces (which have the context adaptation ability), making use of model transformations to enable their adaptation. However, these transformations are used at design-time and not at runtime, as we propose here. In [4], the authors propose an approach that makes use of model representations for developing GUIs, and of model transformations for adapting them. This work, in which the research described in this paper is based on, also considers these GUI models as a composition of COTS components.

The adoption of Component-Based Software Development (CBSD) proposals for software applications design and implementation is increasingly growing. An example can be found in [5], where the authors identify the multiple GUI evolution possibilities that come from working with component-based software architectures (e.g., addition of new components, interface reconfiguration, adaptation to user actions or task, etc.). However, this proposal implements GUI evolution by programming GUI aspects, instead of using model transformation techniques, as we propose in this work. Another example is shown in [6], which

presents a combined MDE and CBSD approach to enable the modelling of structural and behavioural aspects of component-based software architectures. However, this proposal is aimed at general-purpose software architectures, and not particularly suited for GUI development. In [7], the authors focus their research on component retrieval, composition and re-usability in *standalone* DSLs (Domain Specific Languages). This is useful in web applications, especially in those making use of the semantic web. However, as before, this work does not apply these ideas directly to compose GUI applications.

On the other hand, recent software engineering proposals advocate for the use of models at runtime (models@runtime) [8]. Existing research in this field focuses on software structures and their representations. Thus, significant bodies of work look at software architecture as an appropriate basis for runtime model adaptation [9]. Our vision of models@runtime is completely aligned with this idea as our GUIs are, in fact, architecture models. In [10], the authors study the role of models@runtime to manage model variability dynamically. Their research focuses on reducing the number of configurations and reconfigurations that need to be considered when planning model adaptations. However, this work is not focused on GUIs, but in Custom Relationship Management (CRM) applications.

Next section presents the proposed GUI modelling and runtime adaptation approach, in which GUIs will be modelled as component-based architectures. These architecture models will be capable of evolving through model transformations in order to self-adapt according to the events detected by the system.

3 Runtime GUI Adaptation

This paper focuses on applications with *Graphical User Interfaces* (GUI). In fact, our application models may contain any number of GUIs (e.g., one for each type of user). Each GUI, in turn, is built by assembling components, in particular COTS, which are well known in the CBSD literature. We call these components *cotsgets* for their similarity to the gadgets, widgets and other components frequently used in GUI development.

All the *cotsgets* included in each GUI, together with their behaviour and the composition and dependency relations that may exist among them, conform the GUI architecture. As we have opted for a MDE approach, we model GUI architectures using a meta-model. This architecture meta-model can be seen as an aggregation of three parts or subsets, namely: (1) an *structural meta-model*, (2) an *interaction meta-model*, and (3) a *visual meta-model*.

Firstly, the *structural meta-model* allows designers to model composition and dependency relationships among components. Dependencies are modelled by connecting component ports, which may provide or require any number of interfaces (sets of services). Secondly, the *interaction meta-model* is used for modelling the behaviour associated with user-level interactions, defined in terms of the performed tasks. This meta-model includes concepts such as roles, subtasks, choreography, etc. Finally, the *visual meta-model* aims to describe the

component behaviour from the point of view of its graphical representation on the user interface.

As a solution to the interface adaptation process, this work proposes a MDE approach to GUI model evolution [11], where interface architectures are considered as models capable of evolving at runtime. To achieve this, we have implemented a two-stage process, where: (1) the interface architecture models, defined in terms of abstract components, are evolved by means of a model-to-model transformation according to the (user or application) events detected by the system, and (2) the resulting abstract models are then instantiated by means of a regeneration process, where a *trader* selects (from the existing repositories) the cotsgets that better fulfill the requirements imposed by the abstract architecture model, and then regenerates the application in terms of the selected executable components. Thus, the first stage of the process (*transformation phase*) deals with the runtime adaptation of the abstract interface architecture models, while the second one (*regeneration phase*) deals with their instantiation (see Figure 1). It is worth noting that this article is focused only on the transformation phase.

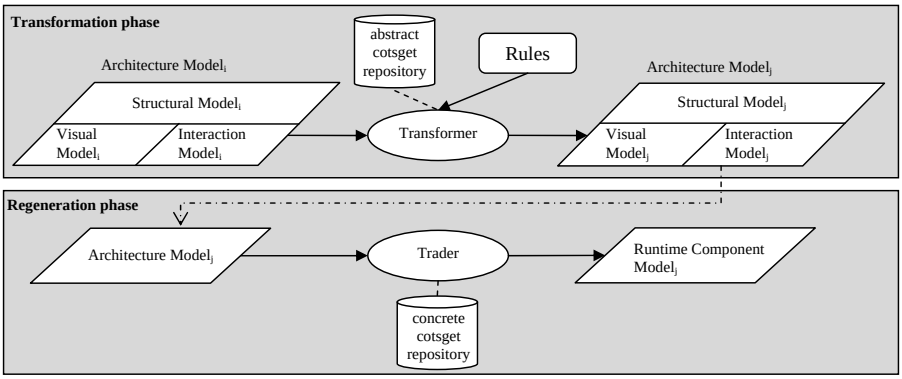


Fig. 1. Schema of Model Adaptation

The model-to-model transformation, implementing the first stage of the process, comprises a set of rules that define how to evolve the current abstract interface architecture model depending on the events detected by the system. As an output, the transformation generates a new abstract interface architecture model, defined in terms of the same meta-model as the input one (i.e., the transformation evolves the input model rather than translating it from one modelling language into another). For the sake of clarity, we have implemented this transformation in two parts: (1) the first one, takes the input interface model and evolves the state machines associated to its components according to the detected event, and (2) the second one executes the actions associated to the new current states of the evolved state machines. Further details about this transformation will be given next in section 3.2.

3.1 Architecture Meta-model

In this paper, we focus on the structural and the visual subsets of the architecture meta-model. The former enables the description of the software architecture in terms of its internal components and the connections existing among them. Similarly, the later enables the specification of the system visual behavior according to the expected runtime adaptation to certain user or application events. An excerpt of the architecture meta-model, showing the main concepts included in these two subsets, is shown in Figure 2.

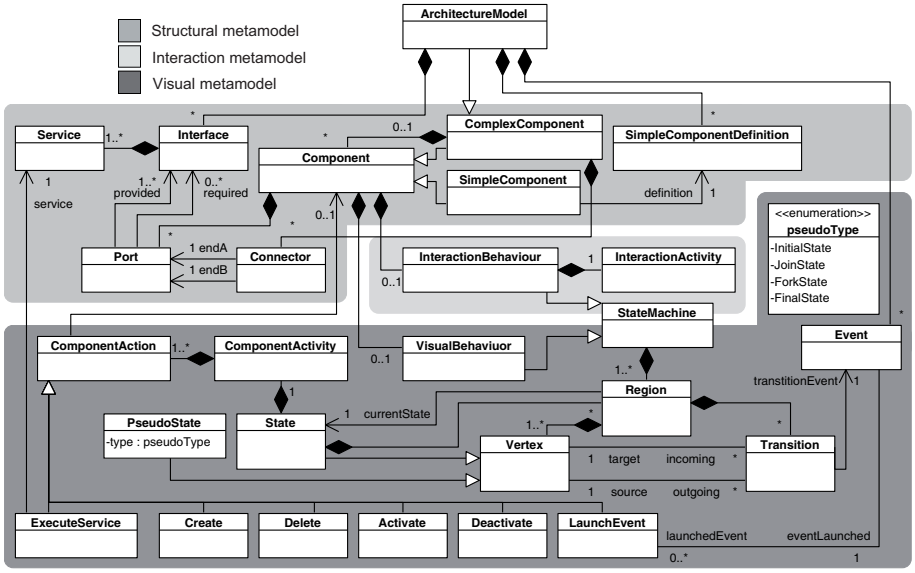


Fig. 2. Architecture Meta-model

The concept *ArchitectureModel* is the meta-model root, and it contains component interfaces (*Interface*), simple component definitions (*SimpleComponentDefinition*), and all the events considered relevant for the system evolution (*Event*). Being defined in the root of the model, these three kinds of elements can be reused by all the other elements in the model. Both *ComplexComponent*s and *SimpleComponent*s are subtypes of the abstract meta-class *Component*. *SimpleComponent*s have a reference to their corresponding *SimpleComponentDefinition*, while *ComplexComponent*s are defined in terms of their internal *Component*s, which can be, in turn, either simple or complex. Each *Component* contains two behavioural descriptions: (1) a *VisualBehavior* which, using a state machine model, defines how each component visually evolves depending on certain *Event*s, and (2) an *InteractionBehavior*, which enables designers to model user's interaction and cooperation (this is out of the scope of this paper).

The *StateMachine*s used to model the component visual behaviour may contain any number of orthogonal (i.e., concurrent) *Region*s which, in turn, may contain any number of *State*s. Each *State* contains a *ComponentActivity* that models the workflow of *ComponentAction*s that need to be executed when the component reaches that state. On the other hand, the *Transition*s between states are associated to one of the *Event*s defined in the ArchitectureModel.

It is worth noting that this work is not intended not prescribe how to construct or deduce the state machine model that better describes each component visual evolution. Conversely, this work is focused on the model transformation supporting that evolution, which implementation is detailed next.

3.2 Runtime Model Adaptation Process

As previously stated, the runtime adaptation of the abstract inteface architecture model has been implemented by means of a model-to-model transformation (Figure 3). This transformation, defined as a set of rules, takes the current interface model (AM_A) and a detected event as its inputs, and generates an evolved interface model (AM_B) as its output. Although the process seems quite straight forward, implementing it in one step is not that easy. Thus, for the sake of simplicity, we have splitted the transformation in two.

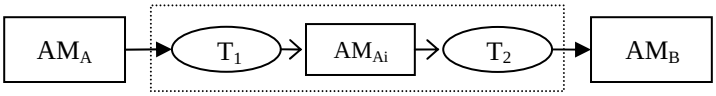


Fig. 3. Adaptation process

The first part of the model transformation (T_1) takes the interface architecture model and the event collected by the system as an input (AM_A), and produces an intermediate interface architecture model (AM_{Ai}) where all the state machines being affected by the collected event are appropriately updated. To achieve this, the transformation finds, for all the *currentState*s (one for each region in every state machines in every component), all the outgoing transitions being fired by the collected event, and updates the *currentState* to the target of the fired transition. Once the state machine models have been updated, the second model transformation (T_2) is exectuted, taking the resulting AM_{Ai} model as an input. The role of this second transformation is to execute the *ComponentAction*s contained in all the updated *currentState*s. As a result, a new interface architecture model (AM_B) is generated. In this first approach to model GUI evolution, we have defined six types of actions that might be executed on a component (as a result of an event launched either by the user or by other component): `Create`, `Delete`, `Activate`, `Deactivate`, `Execute Service` and `Launch Event` (see Figure 2). Table 1 shows two example rules, each one belonging to one of the transformations.

Table 1. Example of ATL rules

T	ATL rule

T_1	`rule RegionEvolution`
	`{ from f: INMM!Region`
	`(f.smParent.parent.parent.currentEvent.eventTransition->exists(t |`
	`t.source = f.currentState) )`
	`to o: OUTMM!Region`
	`(name<-f.name,transitions<-f.transitions,vertex<-f.vertex,`
	`currentState<-f.smParent.parent.parent.currentEvent.eventTransition->`
	`select(t |t.source = f.currentState)->collect(t | t.target),`
	`update<-true) }`

T_2	`rule CreateActionExecutable(f: INMM!Create)`
	`{ to t: OUTMM!Create(parameters<-f.parameters,`
	`sourceComponent<-f.sourceComponent, parent<-f.parent),`
	`c: OUTMM!SimpleComponent`
	`(name<-f.parent.parent.parent.smParent.parent.name +`
	`f.sourceComponent.name,`
	`parent<-f.parent.parent.parent.smParent.parent,`
	`definition<-f.sourceComponent) }`

The first of these example rules (*RegionEvolution*) belongs to T_1 and is related to *Region* elements. This rule only affects those *Region*s containing a transition that (1) has the *Region*'s current state as its source, and (2) is fired by the current event. As a result of apply this rule, the value of the current state will be changed (to the state being the target of the fired transition) and the 'update' attribute will be set to 'true' (to inform the second transformation that the actions associated to that state need to be executed). The second example rule (*CreateActionExecutable*) belongs to T_2 and it is called when the transformation finds a `Create` type action that needs to be executed. Its purpose is to copy the *CreateAction* element to the output model and also to add the *Component* associated to this action (*sourceComponent*) to the interface model.

We have implemented our two model transformations using ATL (ATLAS Transformation Language) [14]. The ATL language is a Domain Specific Language (DSL) aimed at describing model-to-model transformations. ATL is inspired on QVT and is a hybrid language that allows both declarative and imperative constructs. We decided to use ATL as its implementation is quite robust, and it is widely spread in use by the MDE community.

In this first approach, random events are simulted and both transformations are manually launched one after the other. However, we are working on an improved implementation that automatically invokes both transformations every time an event is detected, making use of the ATL facilities for programatically executing transformations.

3.3 A GUI Runtime Adaptation Example

In order to illustrate the proposed approach, this section presents a case study on an example GUI runtime adaptation. It describes in depth a few steps of the adaptation process.

The example shows an interface architecture model composed by two graphical user interfaces (GUI_1 and GUI_2). Each of these GUIs has two simple components (C_1 and C_2). We will simulate an event that adds a *Chat* component to GUI_1. This event will also produce the addition of a *Chat* component to GUI_2. Finally, we will also simulate the generation of a new event that deletes GUI_2 (and all its subcomponents) from the architecture model. Figure 4 shows a snapshot of the interface architecture model at the initial stage ($\texttt{ArchitectureModel}_0$).

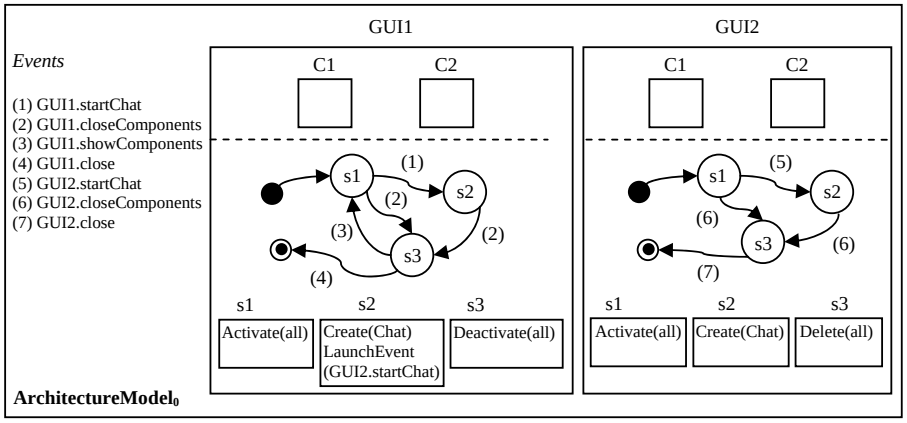


Fig. 4. Initial Architecture Model

Given the initial interface architecture model, when an event occurs the adaptation process starts. An example of this process is shown in Figure 5, which illustrates the model transformation steps executed after the `GUI1.startChat` event happens. As result, the first transformation (T_1) evolves the state machine associated to `GUI1`, changing its current state to `s2`, as indicated by the model. Then, when the T_2 transformation is launched, it executes the `Create(Chat)` and `LaunchEvent(GUI2.startChat)` actions. The first action implies the addition of a `Chat` component within `GUI1`, while the second action causes the launching of a `GUI2.startChat` event. We obtain *Model B* as result.

The adaptation process concludes when all the events haven processed. However, the `GUI2.startChat` event still needs to be attended. Thus, T_1 is launched again and the `GUI2` component changes its current state to `s2`. Finally, T_2 executes the `Create(Chat)` action, resulting in the addition of a new `Chat` component within `GUI2`. In this case, we obtain *Model C* as a result.

Next, figure 6 shows another adaptation example starting from *Model D* (obtained by setting GUI2.closeComponents as the system *currentEvent* in Model C). In this case, we show the models involved in the adaptation process using the reflective model editor provided by the Eclipse Modeling Framework (EMF). In the first step, T_1 changes the current state of `GUI2` to `s3` and sets the 'update'

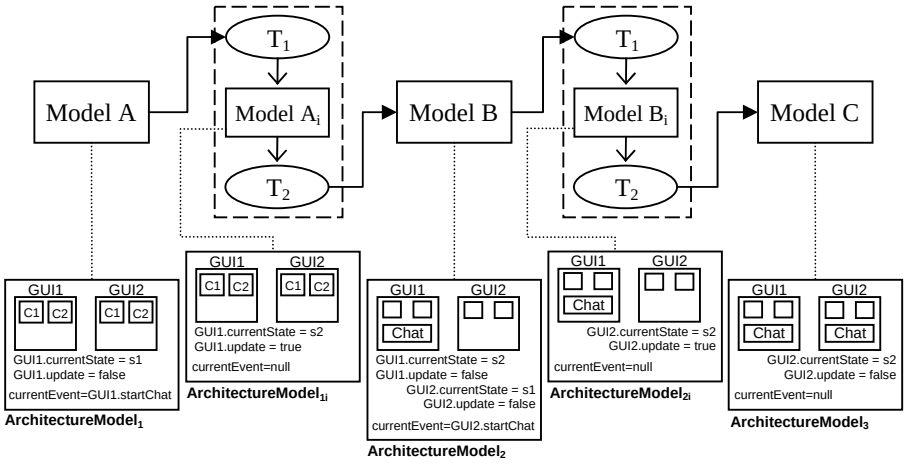


Fig. 5. Transformation Steps

attribute to 'true', as shown in *Model* D_i (central column in figure 6). Then, T_2 executes the `Delete(all)` action, producing the deletion of all the components in `GUI2`, including itself, as shown in *Model E* (right column in figure 6).

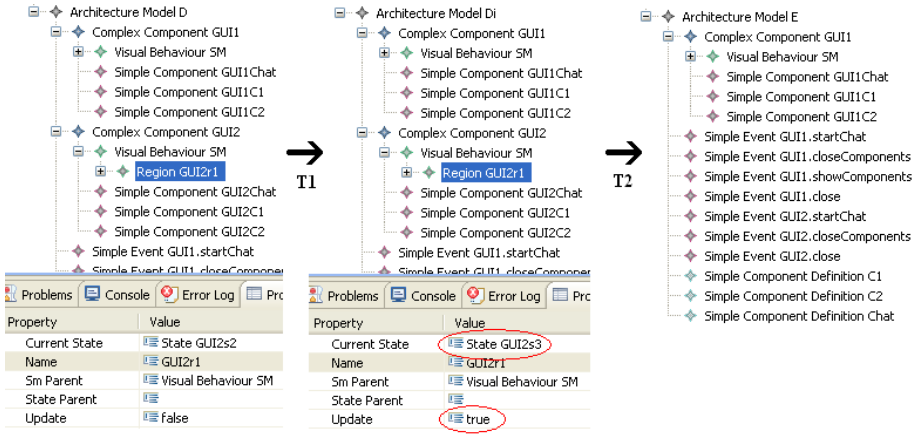


Fig. 6. From *Model D* to *Model E*

4 Conclusions and Future Work

Nowadays, the increasingly growing number and complexity of Information Systems force developers to make them more flexible, easy to adapt and evolve,

and accessible for being manipulated at runtime. Graphical User Interfaces play a key role in this kind of systems, as they ease communication between applications and their end-users. Thus, it is necessary to obtain GUI dynamism and adaptability to user profiles and context. It is also very important to get this adaptation while the system is running, without stopping its execution and without re-modelling its components; in other words, to support a runtime adaptation process. However, most GUIs are still built based on traditional software development paradigms, which do not take into account that they have to be distributed, open, changeable and adaptable. In contrast, GUIs should be able to regenerate themselves at runtime depending on the context, the user interactions, and the changing application requirements.

In this paper, we have presented a MDE approach for the development of adaptable graphical user interfaces. The proposed approach aims to build these applications as assemblies of GUI components. These GUIs will be architecture models that can evolve over time through model transformations with the purpose of changing and adapting on system events. We describe a combined MDE and CBSD proposal to GUI architecture modelling and runtime adaptation. This approach revolves around (1) a meta-model for formally specifying the component-based structure and the visual and interaction behaviour of GUI architectures, and (2) a model-to-model transformation that enables GUI model evolution according to the behavioural rules and the actions performed by the user at runtime. Finally, a runtime adaptation example has been presented that illustrates the proposed approach.

As future work, we would like to develop a graphical tool using the *Eclipse Graphical Modeling Framework* (GMF) [15] in order to easily draw new GUI models conforming to the proposed architecture meta-model. We also plan to evaluate how the use of other MDE tools, in particular those enabling the inclusion of action semantics in meta-models (e.g., Kermeta [16] or AMMA [17, 18]) can help us improving our runtime adaptation process. Besides, we plan to enable the inclusion of alternative behaviours that can be appropriately activated depending, e.g., on previous GUI evolution records or on any relevant contextual information (i.e., enabling the adaptation not only of the interface models but also of the model-to-model transformation itself, also at runtime). Furthermore, we also plan to automate the adaptation process by making use of the ATL facilities for programatically executing the transformations.

Finally, we are interested in studying possible change detection in the interaction meta-model (not covered in this article) by means of automated co-evolution mechanisms and meta-model adaptations [19, 20].

Acknowledgments. This work has been supported by the EU (FEDER) and the Spanish MICINN under grant TIN2007-61497 and TIN2010-15588 project.

References

1. Pérez-Medina, J.L., Dupuy-Chessa, S., Front, A.: A Survey of Model Driven Engineering Tools for User Interface Design. Task Models and Diagrams for User

Interface Design, LNCS 4849, pp. 84–97 (2010)
2. Chavarriaga, E., Macia, J.A.: A model-driven approach to building modern Semantic Web-Based User Interfaces. Advan. in Eng. Soft. 40, 1329–1334 (2009)
3. Sottet, J.S., Calvary, G., Favre, J.M., Coutaz, J., Demeure, A.: Towards Mapping and Model Transformation for Consistency of Plastic User Interfaces. CHI2006, Workshop on The Many Faces of Consistency in Cross-platform Design, (2006)
4. Iribarne, L., Padilla, N., Criado, J., Asensio, J.A., Ayala, R.: A Model Transformation Approach for Automatic Composition of COTS User Interfaces in Web-Based Information Systems. Information Systems Management, 27, 207–216 (2010)
5. Grundy, J., Hosking, J.: Developing adaptable user interfaces for component-based systems. Interacting with Computers, 14, 3, 175–194 (2002)
6. Alonso, D., Vicente-Chicote, C., Barais, O.: V3Studio: A Component-Based Architecture Modeling Language. In 15th IEEE Conf. ECBS, pp. 346–355, (2008)
7. Henriksson, J., Aßmann, U.: Component models for semantic web languages. Semantic techniques for the web, 233–275 (2009)
8. Blair, G., Bencomo, N., and France, R.B (eds.): Models@Run.Time. Special Issue, Computer, IEEE Computer Society (2009)
9. Garlan, D., Schmerl, B.: Using architectural models at runtime: Research challenges. Software Architecture, pp. 200–205 (2004)
10. Morin, B. and Barais, O. and Jézéquel, J.M. and Fleurey, F. and Solberg, A.: Models at Runtime to Support Dynamic Adaptation. IEEE Computer, 42(10), pp. 44–51 (2009)
11. Mens, T.: Introduction and Roadmap: History and Challenges of Software Evolution. Software Evolution, pp. 1–11 (2008)
12. ISO, *Information Technology — Open Distributed Processing — Trading Function: Specification*. ISO/IEC 13235-1, ITU-T X.950
13. Iribarne L, Troya JM, and Vallecillo A: A Trading Service for COTS Components. The Computer Journal. 4, 3, pp. 342–357 (2004)
14. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming. 72 (1–2), 31–39 (2008)
15. Eclipse Fundation. Eclipse Graphical Modeling Framework (GMF), `http://www.eclipse.org/modeling/gmf`
16. Muller, P.A. and Fleurey, F. and Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. Model Driven Engineering Languages and Systems, pp. 264–278 (2005)
17. Del Fabro, M.D. and Jouault, F.: Model transformation and weaving in the AMMA platform. In Workshop on Generative and Transformational Techniques in Software Engineering (GTTSE), Braga, Portugal, pp. 71–79 (2005)
18. Di Ruscio, D. and Jouault, F. and Kurtev, I. and Bézivin, J. and Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Laboratoire D'Informatique de Nantes-Atlantique. Research Report (2006)
19. Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. 12th Int. IEEE EDOC, pp. 222-231 (2008)
20. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. ECOOP 2007, LNCS 4609, pp. 600–624 (2007)

Monitoring Model Specifications in Program Code Patterns

Moritz Balz, Michael Striewe, and Michael Goedicke

Paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen, Essen, Germany
{moritz.balz,michael.striewe,michael.goedicke}@s3.uni-due.de

Abstract. Numerous approaches exist that derive executable systems from well-defined specifications. However, model specifications are not available in program code of such derived systems, which impedes continuous validation and verification at run time. We earlier proposed to embed model specifications into well-defined program code patterns to bridge this semantic gap. We now present an elaboration of our approach to monitor such systems at run time with respect to the underlying abstract models. For this purpose, different techniques are considered that allow to access the modeling information without relying on additional metadata. Based on this, we present a tool that monitors the execution of state machines.

1 Introduction

The creation of software based on formal models is supported by means of various modeling, simulation and verification tools. However, current technologies for model-driven software development (MDSD) cause a loss of semantic information when such models are transformed into source code by manual or automated code generation [1]: The inherent loss of semantic information entails that models are related to derived systems only implicitly [2], thus preventing us from being able to monitor the execution with respect to the model semantics.

To bridge this semantic gap, we proposed to embed model specifications in object-oriented program code [3], for example for state machines [4]. Such *embedded models* introduce program code patterns representing the abstract syntax of models. This single-source approach allows not only to verify programs at development time with respect to the related models, but also to execute embedded models at run time by frameworks relying on structural reflection. In this contribution we consider this an opportunity to monitor the execution: Since these program code patterns represent model specifications completely, different degrees of abstraction are available in the code at the same time. Hence we can monitor model execution at run time without using other representations than the program code. At the same time we can observe how models behave with application data.

This paper is structured as follows: Section 2 describes the monitoring approach by introducing concepts for embedding and identifying modeling information in program code. Then we describe different possible techniques to access

the related program code fragments at run time in section 3. Based on these, a tool for monitoring state machines is introduced in section 4. Afterwards we give an overview of related work in section 5 and draw conclusions in section 6.

2 Approach

The objective of this contribution is to monitor executed software with respect to high-level specifications, but without using additional meta information, so that no inconsistencies can occur and the tool chain is as small as possible. While monitoring as a way of verifying the execution of software systems at run time is well-established, few approaches realize verification with respect to formal models the software is based on. The reason, as mentioned in the introduction, is that the related specifications are not naturally available in the program code that constitutes programs at run time: The code usually describes execution logic only and not its abstract semantics. When it is monitored or verified, the resulting information is generic, focuses on technical details of program code, or must rely on tracing metadata to relate the code to formal models. Considering these problems, we introduce in this section our general approach of coupling model specifications and program code.

2.1 Embedded Models

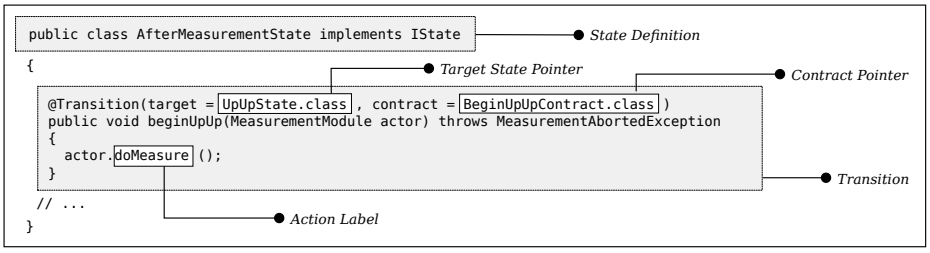
A monitoring as described above means that the program code must contain the specification information. Considering object-oriented programming languages like Java, we can observe a trend to increase the expressiveness of program code fragments. For example, *embedded DSLs* [5] are domain-specific languages that are embedded into other languages, so that semantics of DSLs are used inside a general-purpose language. In addition, some general-purpose languages are able to carry type-safe metadata, e.g., Java Annotations. This enables *attribute-enabled programming* [6] making program code interpretable even at run time.

Embedded models build upon these concepts to relate program code to abstract specifications systematically. Each embedded model provides a program code pattern representing the abstract syntax of a formal model so that a bijective projection between both exists. The pattern elements rely on the semantics of the underlying programming language and its expressiveness regarding single fragments and their interconnections. The statical elements of the programming language and their relations are considered building blocks constituting the pattern. They are of interest in our context since expressiveness of the monitoring depends on their accessibility by appropriate mechanisms at run time.

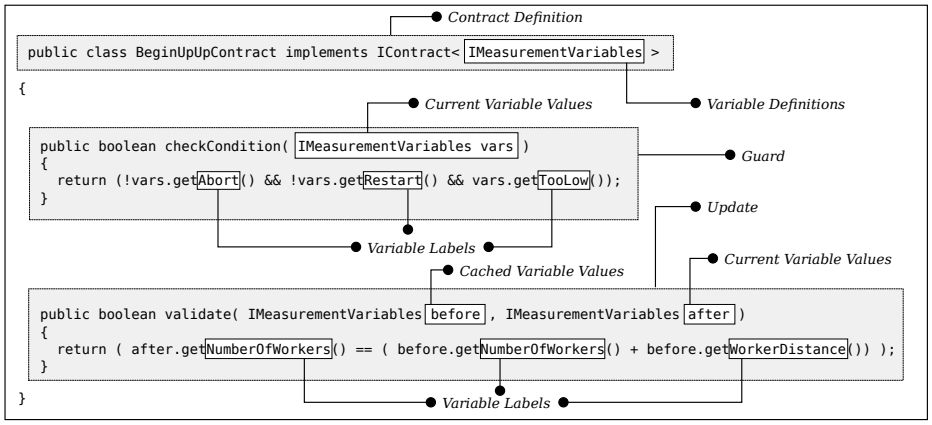
The pattern code is interpreted by means of structural reflection at run time to execute the model specifications. Each embedded model provides an execution framework that accesses and invokes the language elements and thus creates a sequence of actions matching the related model semantics. Considering the monitoring, it is essential that the program code pattern elements and their expressiveness regarding relations to the abstract specifications are by this means accessible at run time.

2.2 Implementation for State Machines

An instance for embedded models exists for the domain of state machines. Since meaningful monitoring in our context depends on the availability of model elements in the program code at run time, we will introduce the program code pattern here and refer to it later. Figure 1 shows an example containing all program code structures of interest. The class at the top represents a state; the class name equals the name of the state. The method in the state class represents a transition. It is decorated with metadata (the annotation `@Transition`) referring to the target state class and a "contract" class containing guards and updates. An interface type referred to as "actor" is passed to transition methods. Its methods are interpreted as action labels which can be called when a transition fires. Thus, a sequence of actor method invocations inside a transition method is interpreted as a sequence of action labels for this transition.



State and Transition Definition in Source Code



Contract Definition in Source Code

Fig. 1. A state definition with an outgoing transitions and its contract. The first method of the contract evaluates a pre-condition with respect to the current variable values, while the second method evaluates a post-condition by comparing the current values to the previous values.

Guards and updates are implemented as two methods in a "contract" class which is shown at the bottom of figure 1. Both evaluate boolean expressions

and return a single boolean value. The guards use the current variable values of the state machine to determine if a transition is able to fire, the updates compare the current values with the values from the point in time before the transition fired to determine the changes to the state space. For this purpose both methods access a "variables" type which is a facade type representing the variables constituting the state space of the state machine. This type contains "get" methods for each variable. The name and return type of each method are interpreted as name and data type of the corresponding variable.

The execution framework interprets and invokes these fragments at run time. The surrounding program code accesses for this purpose the execution framework and passes the class definition of the initial state as well as the variables and actor facade types as parameters. The state machine is then executed as follows:

1. The initial state's class and variables interface are passed to the execution framework. All states reachable from the initial state are instantiated.
2. The current state is set to the initial state.
3. All transition methods of the current state are visited and the variables type instance is passed to the related guard method to determine if the transition is able to fire.
4. The current variable values are cached.
5. The method representing the transition that is able to fire is invoked.
6. The current variable values and the cached variable values are passed to the update method for the validation of variable updates.
7. The current state is set to the target state of the executed transition. The process is continued until the current state is a final state or the state machine runs into a deadlock.

2.3 Monitoring at Run Time

As can be seen in the state machine example, embedded models introduce program code patterns whose elements are related to model specifications. The models are thus views on the program code and need not to be stored in separate notations, so that no inconsistencies between model and implementation can occur. Consistency is not only maintained at development time, but also at run time: Since the related code fragments are not supplementary or optional, but instead used by the execution framework, executed systems with embedded models carry complete information about related specifications naturally.

This availability of models at run time is important for our objective to monitor programs with respect to models, since the model views can be extracted from the code during execution. For this purpose the well-defined elements of the program code patterns serve as entry points for interpreting and monitoring the program code. This enables a validation of programs with two purposes: First, the model view itself is of interest for monitoring the model execution by the framework, so that inferences can be made on correctness of the model from this information. Second, embedded models are tightly integrated with arbitrary program code. This allows for high flexibility during implementation, but causes

the need to validate correctness of the surrounding code with respect to the model. This is supported with appropriate monitoring since the behaviour of the model with application data can be observed.

Monitoring of embedded models thus considers program code pattern instances, for example of state machines, as well as program code of the execution framework: Since it controls the execution, it is an entry point for actions to be monitored. Inside the execution framework for state machines, the following steps can be considered:

- The execution framework iterates on the state machine flow until a final state is reached. The current active state is denoted by a variable inside this iteration pointing to the state class definition. Changes to this variable must be monitored in order to determine state activation.
- Once a state is activated, the execution framework iterates the contained transition methods. The transition under examination is also denoted by a variable that must therefore be observed.
- For each transition the execution framework invokes the guard and update methods and passes the variables facade instance as a parameter. Of interest are all operations inside this methods, especially those that comprise state machine variable values. To interpret the guards and updates thoroughly, the composition of the overall result of these methods from single operation results is also important to monitor.

We will now introduce appropriate monitoring techniques and afterwards a tool that implements the approach.

3 Monitoring Techniques

Our objective is to use this approach for monitoring program execution with respect to models at run time, but without artificial tracing information. Thus it is important to consider the accessibility of the program code patterns and their elements during execution. We will introduce the basic technological approaches for this purpose here. While all of them have already been used by other approaches for monitoring, our contribution here is the application to program code patterns carrying the abstract syntax of formal models. We will therefore not focus on the general technologies, but on their adequacy for monitoring the references to model specifications at run time, in which we encounter important differences.

3.1 Listener Approach

Since all information about the running system and the embedded state machine semantics is available inside the state machine execution framework, the easiest way for monitoring is to extend this framework in order to emit information of interest for monitoring. The execution framework is based on structural reflection

and accesses and interprets a considerable part of the program code structures constituting the pattern. Besides setting listeners programmatically, module-based platforms (like OSGi [7] in the context of Java) allow for a loose coupling of execution framework and components receiving information about the execution. In the case of state machines, listeners can be notified about events for every operation performed on the embedded model:

- Initialization and start of a state machine. This includes information about all states, transitions and variables as extracted from the Java code via reflection. States are uniquely identified by their fully qualified class names.
- Activation of states. This indicates that guard evaluation and transition selection in this state will happen subsequently.
- Selection of transitions. This indicates that program control will be handed over to the business logic in this transition.
- Validation of updates after a transition. The variable values are updated in this event. Additionally, the cached variable values are supplied to allow for comparisons. Additional information is supplied if the validation failed. When this event is fired, program control has been taken over by the state machine execution framework again.

The advantage of listeners is their easy integration into tools based on the Java platform, especially in module-based environments. Since the listeners are accessible from inside the same Java Virtual Machine (provided appropriate programming interfaces or module lookup services exist), even self-monitoring of applications is possible. Thus an application can gain information about its own execution inside the state machine. This is possible without concurrency problems since the framework passes control of the program flow to the listeners during notifications, so that all actions are handled sequentially.

While the approach is working at this level, the degree of detail is limited: Method contents in Java are not accessible by means of reflection and thus black boxes. For this reason operations inside guards and updates are not visible, but only their results after the related method was invoked by the framework.

3.2 Aspect-Oriented Approach

Aspect-oriented programming (AOP) aims to separate cross-cutting concerns from business logic. Monitoring and tracing are often-mentioned examples for AOP usage: Emission of monitoring information is formulated as aspects that are woven into program code. To monitor state machine execution, the code structures of interest are accessed by pointcuts. Appropriate advice written in AspectJ [8] are shown in listing 1.1. The first and the third pointcut wrap around guard and update methods, invoke them and read the result. Afterwards the monitor is notified about the contract class and the current result. The second pointcut is invoked before a transition method is executed, i.e., any method in a class implementing the `IState` interface. It notifies the monitor about the related state class and transition method name.

```
// Wrap guard method invocation and notify about the result
boolean around(Object vars) : execution(* IContract.checkCondition(..)) && args(vars) {
  boolean result = proceed(vars);
  monitor.notifyGuard(thisJoinPointStaticPart.getSignature().getDeclaringType(), result);
  return result;
}

// Notify about forthcoming transition method invocation
before() : execution(* *.*(..)) && target(IState) {
  monitor.notifyTransition(thisJoinPointStaticPart.getSignature().getDeclaringType(),
    thisJoinPointStaticPart.getSignature().getName());
}

// Wrap update method invocation and notify about the result
boolean around(Object before, Object after) :
       execution(* IContract.validate(..)) && args(before, after) {
  boolean result = proceed(vars);
  monitor.notifyUpdate(thisJoinPointStaticPart.getSignature().getDeclaringType(), result);
  return result;
}
```

Listing 1.1. The AspectJ monitoring aspect. All points of interest in the program code pattern are clearly identifiable by simple rules regarding their classes and method names, so that pointcuts can be defined unambiguously.

The main advantage of AOP in this context is that monitoring can be applied without the need to modify the execution framework. With load-time weaving, monitoring capabilities can even be supplemented in systems after the program code has been compiled. This allows for flexible mechanisms that can be applied depending on the context. This is enabled by the fact that the pattern elements of embedded models are well-known and obligatory: Aspects can identify them so that advice and pointcuts can address program code elements related to model elements. Similar as with listeners, this also allows for self-monitoring.

However, while this exterior view on the pattern allows for dynamic extension of such software, it prevents full access to information of interest: Pointcuts can handle information regarding the location of program code in which they are executed (keyword `thisJoinPointStaticPart`). But, they do not gain access to information in terms of sequences of pointcuts: In each state, a certain number of guards is evaluated. Afterwards, one transition method is invoked. While pointcuts are informed about the single actions, they cannot determine which guard belongs to the transition being executed; this information has to be guessed or supplemented by interpreting the program code afterwards. To solve this problem, the execution framework could be changed to make pointers to the objects of interest available as fields.

3.3 Debugging Approach

The debugging approach delegates low-level observation of the program state to the executing platform. The related Java Platform Debugger Architecture (JPDA) [9] provides well-defined programming interfaces to access related events so that those of interest for our monitoring approach can be filtered from the

event queue. In the case of embedded state machines, state activation and transition selection are monitored by observing fields containing the related references in the execution framework with `ModificationWatchpointEvent`s. For guards and updates, `MethodExitEvent`s are of interest that are triggered after all code of a method has been executed, but before the method is left. We use them to access return values of variable interface methods when they are invoked. Together with information about local variable values we can monitor evaluation of guards and updates with such events, too: Since only expressions are used inside these methods, the evaluation is fully comprehensible afterwards by inspection of the values of local variables. The return value of the method and thus the result of the evaluation is also available in this event.

A debugger can hence access all elements of the program code pattern in model implementations as well as all local variables in the execution framework. Different to the listener and AOP approaches, this allows for monitoring guard and update method contents. Since all details of expressions are available, the evaluation of guards and updates can be recorded and presented to the developer for each step. The debugging approach is therefore the only one able to access all elements of the program code pattern. Access to variables and method invocation results is possible without additional effort when they are accessed by the application itself. For the state machine model this is sufficient since the variables are of interest only when they are evaluated in guards. A debugger would also allow to invoke methods at any time. This could be of interest for variable methods to determine their current value. This is, however, intrusive to the program flow, since variable methods may contain arbitrary business logic, which would be executed at times not expected by the developer.

The main influence of debuggers, however, is the need for two running instances: The application being debugged and the debugger itself that controls execution. All information that can be gained is accessible only by the latter, so that a self-monitoring of applications is not possible. In addition, debuggers in general have a strong impact on performance, so that a monitoring of production systems is currently not desirable with this technology. We thus expect that this approach can be used as debuggers are used in general – when the applications are validated during development or maintenance. In this case the relation to abstract models is more meaningful than debugging at the source code level only.

4 Monitoring Tool

These approaches enable monitoring of program code based on embedded models without using tracing information or other metadata, but by considering well-defined code structures only. We will now introduce a tool that is based on such approaches and monitors the related information. Its user interface shown in figure 2 reflects our requirements for the practical use of the monitoring.

The graphical view allows to watch activated state classes and transition methods. Current and cached variable values are shown to exhibit the current state space and to enable monitoring of changes during transitions. Updates that

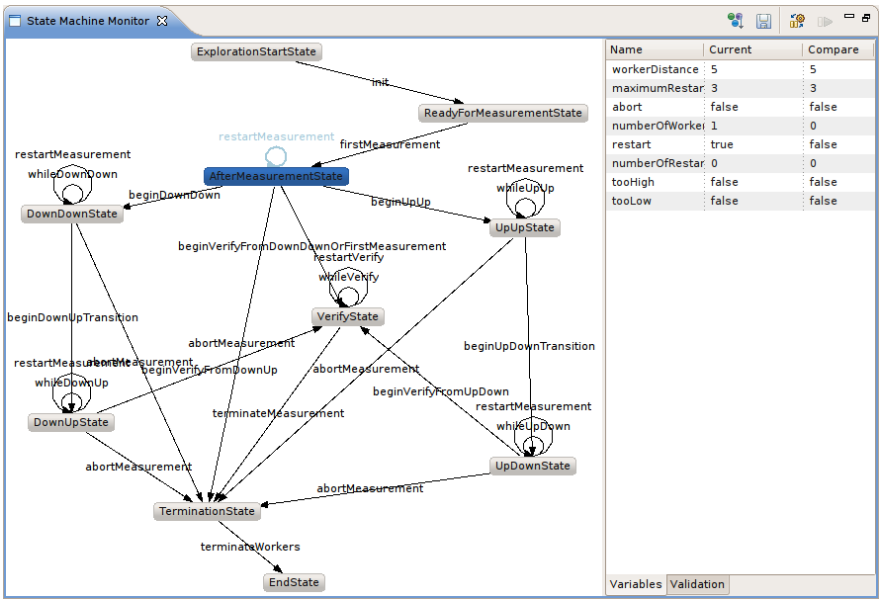


Fig. 2. A state machine model being monitored. Left hand we see the state machine with the active state and transition highlighted, right hand the variable values constituting the state space.

could not be validated successfully are listed separately; since updates do not have impact on the program flow, this information allows developers to look for the causes of such inconsistencies later on. The state machine flow altogether can be paused and resumed by the user. This is possible since business logic is invoked during transitions, and execution control will afterwards return to the state machine. The third button visible on top of the screenshot notifies the execution framework that the state machine flow should pause after the current transition; the button to the right allows then for stepwise execution.

The tool is realized on the Eclipse platform, making it easy to be integrated in Eclipse-based development tools. It uses listeners that are loosely coupled over the OSGi service registry that is provided by the Eclipse platform: Listeners like our tool are hence OSGi bundles being deployed alongside, but independent from business logic. The listener is registered as a named OSGi service that is detected by the execution framework. The resulting architecture as sketched in figure 3 allows to use almost arbitrary tools to be notified about events for every operation performed on the embedded state machine.

5 Related Work

Following our objective to monitor the execution of program code that is related to model specifications, we must consider related work with respect to two

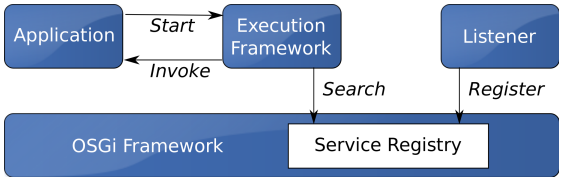


Fig. 3. Component architecture with the monitoring listener. Applications are composed of components using the execution framework based on the OSGi platform. The listener component is optional and hence only coupled via the service registry.

topics: First, general approaches that relate program code to high-level specifications which are in theory appropriate for monitoring; second, the application of monitoring in specific technological environments.

Round-trip engineering [10] relates generated program code to models but targets the development time instead of the run time and cannot be fully automated [11]. Informal specifications can be inferred from program code by detecting patterns [12], and similar, specifications can be extracted from program code based on design patterns [13]. However, this requires manual effort or is based on heuristics and not appropriate for a precise monitoring. Executable models [14] are accessible at run time, too. However, they are only appropriate for applications completely expressed as models, while we consider cases where models are connected to program code and thus monitor the related data exchange.

Monitoring for compliance with so-called *design models* [15] or design pattern contracts [16] is based on low-level semantics of detailed patterns. Similarly, model checkers for program code work with low-level semantics of the programming language and thus consider whole applications as models [17]. In contrast, monitoring with embedded models is related to abstract specifications. For this reason it can also clearly be distinguished from general debugging approaches.

We do not aim to present a notation for the specification of all possible system models like the Java Modeling Language (JML) [18] or the approach to use Smalltalk with its introspection capabilities as a meta language [19]. In contrast to static analysis tools like DiscoTect [20] we do not target detection of unknown structures and models, but focus on well-known models that can thus be examined more thoroughly and with respect to a formally-founded background. We also do not require changes in the program code to introduce references to specifications as is necessary for PathFinder's verification statements [21] or the approaches to monitor OCL constraints with aspect orientation [22, 23], which rely on metadata in source code comments. Instead, we can infer all model specifications directly from the program code pattern.

6 Conclusion

We presented our approach to monitor model specifications that are embedded in object-oriented program code. We were acting on the assumption that the re-

lated program code pattern structures are precise enough to allow for inference to model specifications even at run time. To show this, different approaches for information retrieval have been evaluated as possible alternatives. Our conclusion is that all are appropriate to monitor the state machine semantics, although in different degree of detail and with different impact on the necessary changes to the program code. All are non-intrusive regarding the source code of the monitored system and two of them are even non-intrusive to the source code of the execution framework. However, the degree of detail varies since only debugging approaches allow to monitor guards and updates in detail. On the other hand, listeners and AOP require less overhead at run time. With AOP, monitoring aspects can even be attached dynamically to the programs since they can work on the pattern specifications after compilation.

For the current implementation of a monitoring tool, the listener approach was chosen since it allows to access the most important information with little effort and provides the ability for self-monitoring. However, if the required environment is available, the debugging approach is more thorough and allows to monitor every detail of the state machine execution. Future work will thus include the development of an appropriate monitoring tool. Due to the maturity of the JPDA and related user interfaces in integrated development environments, we will then be able to integrate the monitoring with the debugging user interface of development environments. With this integration, the monitoring of abstract model specifications can be seamlessly integrated with debugging of arbitrary Java applications, thus making model validation at run time an integral part of the development process.

References

1. Brown, A.W., Iyengar, S., Johnston, S.: A Rational approach to model-driven development. IBM Systems Journal **45**(3) (2006) 463–480
2. Tichy, M., Giese, H.: Seamless UML Support for Service-based Software Architectures. In Guefi, N., Artesiano, E., Reggio, G., eds.: Proceedings of the International Workshop on scientiFic engIneering of Distributed Java applIcations (FIDJI) 2003, Luxembourg. Volume 2952 of Lecture Notes in Computer Science., Springer-Verlag (November 2003) 128–138
3. Balz, M., Striewe, M., Goedicke, M.: Continuous Maintenance of Multiple Abstraction Levels in Program Code. In: Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development - FTMDD 2010, Funchal, Portugal. (2010) 68–79
4. Balz, M., Striewe, M., Goedicke, M.: Embedding State Machine Models in Object-Oriented Source Code. In: Proceedings of the 3rd Workshop on Models@run.time at MODELS 2008. (2008) 6–15
5. Kabanov, J., Raudjärv, R.: Embedded Typesafe Domain Specific Languages for Java. In: PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, New York, NY, USA, ACM (2008) 189–197
6. Schwarz, D.: Peeking Inside the Box: Attribute-Oriented Programming with Java 1.5. ONJava.com (June 2004) `http://www.onjava.com/pub/a/onjava/2004/06/30/insidebox1.html`.

7. OSGi Alliance: OSGi Service Platform, Core Specification, Release 4, Version 4.1. IOS Press, Inc. (2005)
8. Colyer, A., Clement, A., Harley, G.: Eclipse AspectJ. Addison-Wesley (2004)
9. Sun Microsystems, Inc.: Java^{TM} Platform Debugging Architecture API `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.
10. Sendall, S., Küster, J.: Taming Model Round-Trip Engineering. In: Proceedings of Workshop on Best Practices for Model-Driven Software Development. (2004)
11. Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In Briand, L., Williams, C., eds.: Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings. Volume 3713 of LNCS., Springer (2005) 476–491
12. Philippow, I., Streitferdt, D., Riebisch, M., Naumann, S.: An approach for reverse engineering of design patterns. Software and Systems Modeling **4**(1) (February 2005) 55–70
13. Mili, H., El-Boussaidi, G.: Representing and Applying Design Patterns: What Is the Problem? In Briand, L.C., Williams, C., eds.: MoDELS. Volume 3713 of Lecture Notes in Computer Science., Springer (2005) 186–200
14. Hen-Tov, A., Lorenz, D.H., Schachter, L.: ModelTalk: A Framework for Developing Domain Specific Executable Models. In: Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling. (2008)
15. Sefika, M., Sane, A., Campbell, R.H.: Monitoring Compliance of a Software System With Its High-Level Design Models. In: ICSE '96: Proceedings of the 18th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (1996) 387–396
16. Soundarajan, N., Hallstrom, J.O., Tyler, B.: Monitoring Design Pattern Contracts. In: Proceedings of the the 3rd FSE Workshop on the Specification and Verification of Component-Based Systems. (2004) 87–93
17. Holzmann, G.J., Joshi, R., Groce, A.: Model driven code checking. Automated Software Engineering **15**(3-4) (2008) 283–297
18. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design. In Kilov, H., Rumpe, B., Simmonds, I., eds.: Behavioral Specifications of Businesses and Systems, Kluwer (1999) 175–188
19. Ducasse, S., Gîrba, T.: Using Smalltalk as a Reflective Executable Meta-language. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings. Volume 4199 of Lecture Notes in Computer Science., Springer (2006) 604–618
20. Yan, H., Garlan, D., Schmerl, B., Aldrich, J., Kazman, R.: DiscoTect: A System for Discovering Architectures from Running Systems. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2004) 470–479
21. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model Checking Programs. Automated Software Engineering Journal **10**(2) (2003)
22. Richters, M., Gogolla, M.: Aspect-Oriented Monitoring of UML and OCL Constraints. In: Proceedings of Workshop Aspect-Oriented Software Development with UML. (2003)
23. Chen, F., D'Amorim, M., Roşu, G.: A formal monitoring-based framework for software development and analysis. In: Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04). Volume 3308 of LNCS., Springer-Verlag (2004) 357–373

Separating Local and Global Aspects of Runtime Model Reconfiguration

Frank Trollmann, Grzegorz Lehmann, Sahin Albayrak

DAI-Labor, TU Berlin
Faculty of Electrical Engineering
and Computer Science
Frank.Trollmann@dai-labor.de, Grzegorz.Lehmann@dai-labor.de,
Sahin.Albayrak@dai-labor.de

Abstract. There is a growing need for applications that are able to adapt themselves to the context of use. One promising approach for the adaptation of an application during its execution is the use of models at runtime. In this approach models of the application and its context of use are kept alive during the execution. The application can be adapted by reconfiguring the structure of these models.
Model Reconfiguration has local aspects as it handles the structure of a model and has to deal with its specific properties. It also possesses global aspects, as the joint reconfiguration of several models is required due to consistency considerations. This paper aims at solving possible conflicts between the global and the local aspects of Model Reconfiguration by introducing a distinction between two Levels of abstraction that enables the designer to separate and interrelate global and local aspects of Model Reconfiguration.

1 Introduction

There is a growing need for applications that are able to adapt themselves to the current context of use. Especially in dynamic and personalized areas like smart homes such adaptive applications are important as they can adjust to the user's specific needs as well as her environment. Adaptations trigger changes in the applications user interface or behavior. In [1] the need for adaptations as well as special challenges in this field of research is stressed.

Adaptability results in a growing complexity of software applications and their development. According to [2], one promising approach for dealing with this complexity is the models at runtime approach (also called models@run.time approach). This approach is similar to Model Driven Engineering (MDE) [3], where the development of a software application is accompanied by the creation and transformation of a set of models. The difference between these approaches is in their goal. MDE aims to generate the final application code from the intermediate models. The goal of the models@run.time approach is to keep the models alive at runtime. In this approach the running application results from an interpretation of these models.

In the models@run.time approach a running application consists of a set of models. These models represent different aspects of the application and its context of use. The applications user interface and behavior result from the interpretation of these models within a certain framework. The advantages of this approach for adaptations are twofold. First, the current state of the application and its context of use can easily be retrieved by querying the representing models. This is essential because the application is supposed to react to these states by adapting itself. Second, a change in the structure of one or more of these models automatically affects the execution of the application. Thus, the adaptation of an application can be achieved by changing the structure of its models. We call such a structural change a Model Reconfiguration.

Some adaptations of an application require the joint reconfiguration of several models. This can be necessary if an intended adaptation is within the scope of several models at once. Another reason for a joint reconfiguration of models is to maintain the consistency in models that partially overlap in the aspects they represent. According to this, Model Reconfiguration needs to have global aspects that enable the designer to express joint reconfigurations of several models.

A models@run.time framework may contain different models expressed in different modeling languages. Reconfiguration techniques are often specific to one modeling language. This allows them to optimally deal with the specific structural and behavioral properties of this modeling language. This means Model Reconfiguration also needs to have certain local aspects.

When implementing a framework for Model Reconfiguration, there can be a conflict of interest between these local and global aspects. In this publication we propose a distinction between a Reconfigurable Model and a Reconfiguration Model. This distinction serves to establish a separation of concern between the local and global aspects of Model Reconfiguration, which allows the designer to model and interrelate both.

The paper is structured as follows. First, in Section 2 the problems arising from the global and local aspects of Model Reconfiguration are subsumed. Reconfigurable Models are then discussed in Section 3. Afterwards, our description of the Reconfiguration Model is given in Section 4. In Section 5 existing approaches to Model Reconfiguration are introduced and related to the concepts introduced in this paper. Section 6 gives a conclusion and hints to future work.

2 Problem Statement

While authors are clear about the overall goals of Model Driven Engineering, the actual set of models required to build an application is far from fixed. One possible set of models is described in the CAMELEON Reference Framework [4]. Current approaches differ in the set of used models, as well as the modeling languages these models are described in.

The same goes for models@run.time approaches. Several possible runtime ensembles of models do exist and several modeling languages are used for describing

these models. Techniques for Model Reconfiguration are often local to a certain modeling language. This determines the local aspects of Model Reconfiguration.

These local aspects are important because Model Reconfiguration is tightly interwoven with the structure that is reconfigured. The advantage of a reconfiguration technique, specifically tailored to a modeling language, is that it can handle or preserve certain properties, specific to this modeling language. One example for such a specific approach is the Graph Transformation technique, analyzed in [5]. This technique can only be applied to P/T nets. Due to this exclusiveness it is able to preserve the firing behavior of these net.

It is also important to change the structure of several models at the same time. This enables the designer to treat the set of runtime models as a consistent whole. For example, the CAMELEON Reference Framework contains three different models for user interfaces: the Abstract, Concrete and Final User Interface Model. When these models are used at runtime they have to be kept consistent with each other. This can be best done by reconfiguring them jointly and thus ensuring consistency after each reconfiguration.

The set of runtime models within an application is not restricted to a fixed set of models. In addition, the models, used within one runtime ensemble are likely to be modeled in different modeling languages. This constitutes a potential conflict between global and local aspects of Model Reconfiguration. According to the local aspects it is possible that the designer chooses a different reconfiguration technique for each model in the runtime ensemble. These techniques are local to their modeling language and cannot be applied the other models. This is a problem for the global aspects of Model Reconfiguration which require a joint reconfiguration of the set of runtime models.

Our approach towards these potential problems is to separate the local and global aspects within two different components. The idea is to unite each model and its local reconfiguration as a so-called Reconfigurable Model. A component, called a Reconfiguration Model, steers the global reconfigurations. This model uses the Reconfigurable Models in order to accomplish this goal.

For this separation to work, the Reconfiguration Model should be able to abstract from the following properties of the Reconfigurable Models:

- *Model Kinds:* The Reconfiguration Model should be independent of the kinds of used models as well as their purpose in order to not restrict the designer to a fixed set of models.
- *Modeling Language:* The Reconfiguration Model should not be limited in the set of modeling languages it is able to reconfigure. This way the designer is free in her choice of modeling languages.
- *Reconfiguration Technique:* The Reconfiguration Model should be able to abstract from the used reconfiguration technique. This way the designer is free in her choice of reconfiguration technique.

In Sections 3 and 4 the concepts of Reconfigurable Model and Reconfiguration Model are introduced and discussed in detail.

Figure 1 shows an example for excerpts of a Task and User Interface model which are part of a runtime ensemble. The Task Model represents the applica-

tions structure of task and subtasks. The User Interface Model represents its user interface. These two models represent the user interface of a login window and its part of the task tree. The user is able to input username and password in parallel and then finish the task "Login" by clicking on the login button.

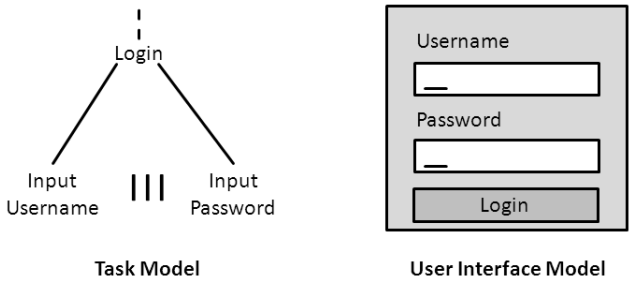


Fig. 1. Running Example: Two Models of a Login Mask

One possible adaptation of the user interface is to show the input of username and password in consecutive windows. A reason for this adaptation is the availability of different authentication protocols. In this case it is not certain that the user needs a password to log into the system. For example, he could also be identified by the MAC Address of his devices. In this case the input of a password becomes obsolete. Such an adaptation requires changes in the User Interface Model and the Task Model. The user interface has to be adapted to showing two windows. One to input the username and one to input the password. In addition the task model has to reflect the fact that "Input Username" and "Input Password" are now executed consecutive.

This example is oversimplified as the reconfiguration is really more complex than indicated here. The changes in the models are more complex due to the requirement to reflect more than one authentication method. In addition, other models have to be reconfigured to connect the new user interface and its execution logic. Nevertheless, this toy example already requires the joint reconfiguration of two models and will serve as a running example throughout the rest of the paper.

3 Reconfigurable Model

The notion of a Reconfigurable Model is introduced in order to encapsulate the local aspects of Model Reconfiguration. In each Reconfigurable Model the designer is able to concentrate on the structural changes of one model. In this scope she is able to choose a reconfiguration technique that suits her preferences. In this Section the notion of a Reconfigurable Model is defined and discussed.

A Reconfigurable Model is a model that can be reconfigured. In addition to the models structure it contains means for changing this structure. A scheme for a

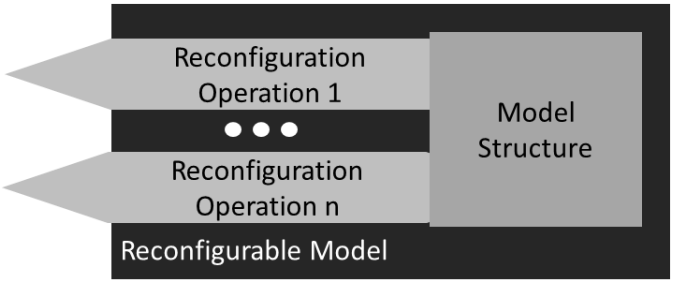


Fig. 2. Scheme of a Reconfigurable Model

Reconfigurable Model can be seen in Figure 2. A Reconfigurable Model consists of a Structure and a set of Reconfiguration Operations. The Reconfiguration Operations can be executed, resulting in a change of the structure. A more formal definition of a Reconfigurable Model can be seen in Definition 1.

Definition 1 (Reconfigurable Model). *A Reconfigurable Model r=(S,OPs) consists of a model S, determining the Structure of the Reconfigurable Model, and a set of Reconfiguration Operations OPs, that can be applied to change this Structure.*

The Structure S of a Reconfigurable Model is not restricted. An arbitrary model conforming to an arbitrary meta model may constitute this structure. Reconfiguration Operations *OPs* represent ways to change S in a way that it still conforms to its meta model. A more formal definition of a Reconfiguration Operation is given in Definition 2.

Definition 2 (Reconfiguration Operation). *A Reconfiguration Operation for a meta model MM is a function* $OP : M \rightarrow M$ *mapping one model, that conforms to MM to another one. M is the set of all models that conform to MM.*

A Reconfiguration Operation is a function that can be applied to the current Structure S to generate a new Structure. This operation strictly acts within the set of models conforming to the meta model *MM* of S. Thus, the Reconfiguration Operations cannot violate the conformity to the meta model. A Reconfiguration Operation can be executed from outside of the Reconfigurable Model without any knowledge of S. For each Reconfiguration Operation *OP*, a Reconfiguration Endpoint *OP'* is available which automatically applies *OP* to S.

While modeling a Reconfigurable Model the designer has to provide the current Structure S as well as the set of Reconfiguration Operations *OPs*. In principal, any model can be used for defining S, regardless of its modeling language. Based on this model and its modeling language, the designer then chooses the most suitable reconfiguration techniques to model *OPs*.

This process represents the standard case of producing a Reconfigurable Model. Other variations are also imaginable. For example the Reconfiguration

Operations could be generated automatically from another description of variability, like an enumeration of all possible structures.

At runtime, S is used as the initial structure of the Reconfigurable Model. The Reconfiguration Operations can be executed in order to change this structure. These operations and their Reconfiguration Endpoints enable external models, like the Reconfiguration Model, to trigger changes in S.

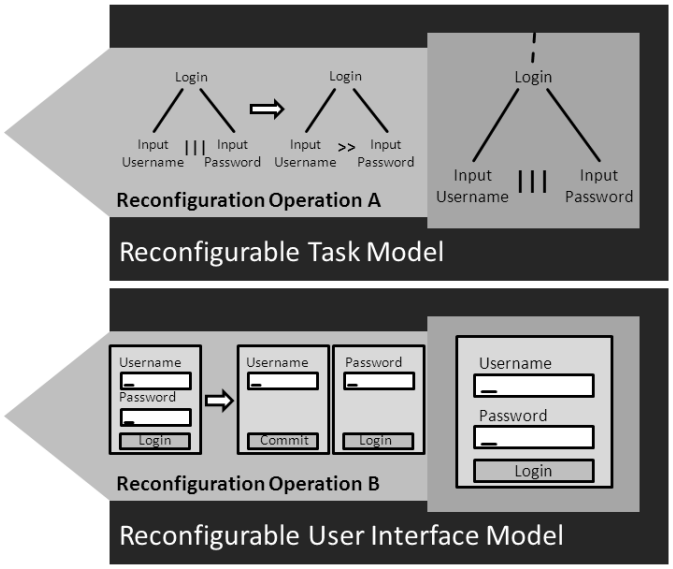


Fig. 3. Running Example: Reconfigurable User Interface and Task Model

In our Running example there are two models that need to be made reconfigurable. These are the Task and User Interface Model, introduced in Figure 1. In Figure 3 their reconfigurable versions are depicted. In both cases the structure S consists of the models introduced in Figure 1. Each model contains one Reconfiguration Operation. In the Reconfigurable Task Model the two parallel tasks "Input Username" and "Input Password" can be made consecutive. The Reconfiguration Operation of the Reconfigurable User Interface Model splits the login window and distributes the input elements for username and password.

In the Figure it is not mentioned how these Reconfigurations are implemented. The techniques, used in both models are independent from each other and can be chosen by the designer. Due to the graphical representation of a user interface she might decide to use a form of Graph Transformation in the Reconfigurable User Interface Model. For changing one temporal operator in the Reconfigurable Task Model she might choose an action as provided by the KerMeta environment. These two Reconfiguration Operations describe the changes, required for our Running Example. However, on the level of Reconfigurable Mod-

els it is not possible to interrelate these two model changes. This is the purpose of the Reconfiguration Model, introduced in the next Section.

4 Reconfiguration Model

In the previous section Reconfigurable Models as an encapsulation of the local aspects of Model Reconfiguration, are described. This section introduces the notion of a Reconfiguration Model. This model builds upon the definition of Reconfigurable Models and reflects the global aspects of Model Reconfiguration.

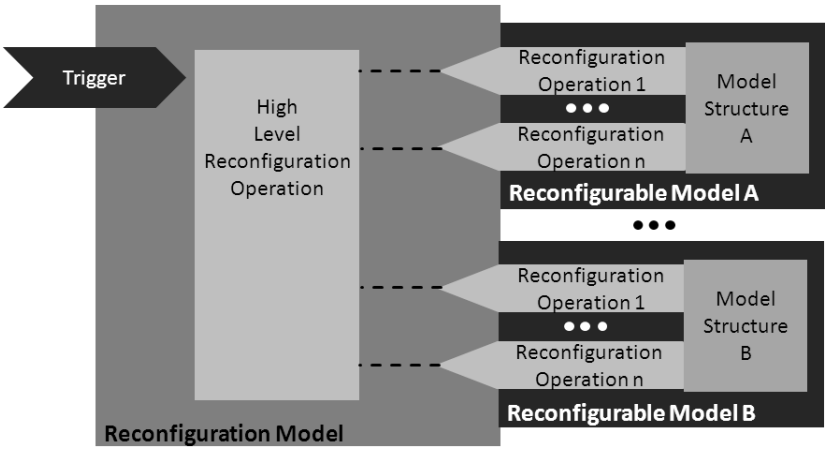


Fig. 4. Scheme of a Reconfiguration Model

A scheme for a Reconfiguration Model can be seen in Figure 4. A Reconfiguration Model contains High Level Reconfiguration Operations. Each of them is connected to a trigger, which is responsible for determining when to execute this Operation. The Reconfiguration Model can adapt a set of Reconfigurable Models. This is done by executing the Reconfiguration Endpoints of their Reconfiguration Operations.

The triggers serve as Guards for executing the High Level Reconfiguration Operations. Whenever a trigger fires the High Level Reconfiguration Operation is executed and calls the Reconfiguration Operations it requires.

The designer models the High Level Reconfiguration Operations to describe complex joint reconfigurations of several models. The structural changes in each model are accomplished by calling the Reconfiguration Operations, provided by their Reconfigurable Model. This represents the global aspects of Model Reconfiguration. Inside this global description of reconfiguration logics she is able to abstract from the properties mentioned in Section 2:

– *Model Kinds:* The Reconfiguration Model can work with any Reconfigurable Model regardless of its inner implementation or purpose. Therefore, it is independent of the actual set of models used at runtime.
– *Modeling Language:* The structure that is reconfigured is hidden within the Structure S of a Reconfigurable Model. The Reconfiguration Model only triggers the Reconfiguration Operations for changing this structure and does not touch the structure directly. Thus, the Reconfiguration Model can work independent of the Modeling Language.
– *Reconfiguration Technique:* The Reconfiguration Model only needs a reference to the Reconfiguration Operations in order to execute them. It does not have to know how they are implemented. Thus, it can work independent of the reconfiguration technique used to describe the Reconfiguration Operations.

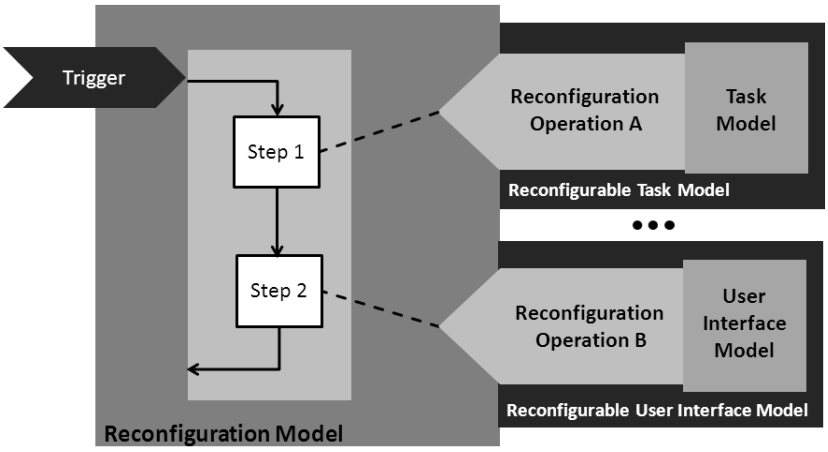


Fig. 5. Running Example: The Reconfiguration Model

A Reconfiguration Model for our running example is depicted in Figure 5. This Reconfiguration Model uses the Reconfigurable Task Model and Reconfigurable User Interface Model depicted in Figure 3. It contains one High Level Reconfiguration Operation, which consists of two steps. In the first step the Task Model is reconfigured, using Reconfiguration Operation A. In the second step Reconfiguration Operation B of the Reconfigurable User Interface Model is triggered. This High Level Reconfiguration Operation executes all changes discussed in our running example.

The purpose of this publication is to propose the differentiation between local and global aspects of Model Reconfiguration and their separate handling by introducing two levels of abstraction. The complete definition of both components is still work in progress. Although a High Level Reconfiguration Operation is depicted as a series of consecutive steps in the running example, we do not believe

this to be the final or best solution. Some hints on possible implementations are given in the remainder of this section.

There are several possible ways to implement a High Level Reconfiguration Operation. For example, a visual language could be used for describing the workflow of application of the Reconfiguration Operations. Languages like UML - Statecharts, Flowcharts or Petri Nets could be utilized for this task. Several lessons can also be learned from the field of Graph Transformation, where several means for high level transformation logic are proposed. Imperative programming languages, like Java or C++, could also be utilized.

Before deciding on one of these alternatives a detailed analysis of the properties and control structures, required for a comfortable modeling of high level reconfigurations has to be carried out. In the next section, related work in the field of Model Reconfiguration is discussed. This work is then related to the local and global aspects of model reconfiguration and the concepts introduced in this paper.

5 Related Work

Several approaches towards Model Reconfiguration have been implemented. This section serves to introduce some of these approaches and interrelate them to the local and global aspects identified in this paper and our notion of Reconfigurable Model and Reconfiguration Model.

One of the most recognized techniques for reconfiguring models is Graph Transformation. A Graph Transformation rule searches and substitutes one occurrence of a pattern within a graph with another one. The concrete syntax of several modeling languages can be described as a graph. For this reason a variation of a Graph Transformation technique is an obvious choice for structural adaptations in these languages. A variety of applications for Graph Transformations to dynamic systems can be found in [6]. Although neither of these applications is specially applied to the reconfiguration of models at runtime, they all contain an initial structure that is reconfigured using graph transformation rules. This is very similar to our notion of a Reconfigurable Model.

Several specific Graph Transformation languages do exist. These languages are dedicated to a certain modeling language and can only be applied to models within this language. For example, in [5] a Graph Transformation approach for rewriting P/T nets is introduced. This technique preserves the firing behavior of P/T nets. This shows the capability of specific Graph Transformation languages to preserve properties of the transformed models and their structure. A specific Graph Transformation Language can be a good choice of a reconfiguration technique to describe Reconfiguration Operations.

In [7] this specific Graph Transformation language for P/T nets is applied to model a flexible emergency scenario. In this publication, the initial scenario is modeled using a P/T net and the possible changes to this scenario are modeled as a set of Graph Transformation rules, specific for P/T nets. This setup is very similar to our notion of a Reconfigurable Model. The P/T net can serve as the

current Structure S and the set of Graph Transformation rules are similar to our Reconfiguration Operations OPs.

In [8], Schürr studies approaches towards building programmed graph replacement systems from Graph Transformation rules. He also proposes his own approach towards unifying these approaches. The purpose of programmed graph replacement systems is to provide means for defining complex schemes of reconfiguration out of Graph Transformation rules and thus enable the designer to take a global view on Graph Transformation. However, an abstraction from the concrete reconfiguration technique or modeling language was out of scope for Schürr. For this reason programmed graph replacement systems do not make these abstractions. Nevertheless, we consider this publication to be a valuable source of inspiration for the design and implementation of High Level Reconfiguration Operations.

USIXML [9] uses Graph Transformation techniques in a more general scope. In this framework all models are described in XML. This format is used as the basis for Graph Transformation. This enables a transformation between different models, used for backward and forward engineering. The approach towards Graph Transformation taken in USIXML is also an interesting one for Model Reconfiguration as it enables the designer to describe several models as one joint XML file and then reconfigure them jointly. This approach captures certain global aspects. However, it can only be applied to models that are described within an XML structure. Thus, it is not general enough to satisfy our requirements from Section 2.

Graph Transformation is not the only concept that has been tested within the scope of Model Reconfiguration. In [10] Morin et Al. describe their approach towards modeling adaptive systems using models and aspects. The system is specified as a set of aspect models. They are weaved into one runtime model, which represents the whole running application. The system is adapted by reconfiguring the aspect models and weaving a new runtime model. This newly woven runtime model is then compared to the old one and a script for transforming the old into the new one is generated. In this publication model reconfiguration also clearly has local and global aspects. Reconfigurations are specified for each aspect model but are then woven into one runtime model. Global consistency can be checked by specifying a set of consistency constraints. However, this can only serve to check consistency after the reconfiguration. In our opinion a way for specifying how two aspects are reconfigured jointly in order to preserve their consistency is still required.

The meta modeling language KerMeta [11] can also proof as a useful tool for Model Reconfiguration. The purpose of this language is to provide a language that is able to model the structure and behavior of a modeling language. The behavior is modeled by an action language. This way the designer of a modeling language can model the structure and behavior of this language jointly. This approach can also proof interesting for model reconfiguration as structural changes of such models can also be described by this action language. The KerMeta

action language can be used as a technique for implementing Reconfiguration Operations.

In this section we introduced a selection of approaches towards runtime reconfiguration of models. None of these approaches were explicitly able to model all local and global aspects of Model Reconfiguration. However, several similarities between these approaches and the components and separation introduced in this paper have been found. This leads us to expect that the separation and components we introduced, even given their current level of abstraction, capture many of the aspects, also addressed by these publications and are a good starting point for further research towards a universal Reconfiguration Model.

6 Conclusion and Future Work

This paper proposes a separation between a Reconfigurable Model, which is a model that offers certain Reconfiguration Operations that can be executed at runtime, and a Reconfiguration Model, which is responsible for triggering and steering the reconfigurations in all models used at runtime. The aim of this separation is to provide an approach to Model Reconfiguration that captures local and global aspects. Local aspects are strongly interwoven with the used models and modeling languages. Global aspects concern the interrelation of several models and their joint reconfigurations.

The Reconfigurable Model reflects local aspects of Model Reconfigurations and enables the designer to model Reconfiguration Operations that are close to the used modeling languages. In the Reconfiguration Model the designer can take a global view on Model Reconfiguration and interrelate the reconfigurations of different models.

Several existing approaches have been analyzed regarding their capability to capture the global and local aspects of Modeling Reconfiguration. Although none of the analyzed approaches had the flexibility to deal with all our requirements, they had several similarities to our approach.

In the near future the notions of Reconfigurable Model and Reconfiguration Model have to be further detailed. For example, the current definition provides no means for expressing additional application conditions for Reconfiguration Operations. For this step several sources of inspiration have been identified within the related work.

In Future Work we also plan to define a specific language for describing Reconfiguration Models. In Section 4 some ideas on how the components within this Reconfiguration Model can be implemented are given. These sources of inspiration have to be analyzed for their actual usability before a decision towards the final implementation can be made. In addition to a language for expressing such high level reconfiguration operations, a set of control structures, like conditional or repeated application of rules has to be defined and formalized.

Additionally, we plan on defining and executing a case study with the defined reconfiguration language as a proof of concept.

References

1. Coutaz, J.: User interface plasticity: Model driven engineering to the limit! In: ACM, Engineering Interactive Computing Systems (EICS 2010) International Conference. Keynote paper., ACM publ. (2010) 1–8 Keynote paper.
2. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer **42**(10) (2009) 22–27
3. Schmidt, D.C.: Model-driven engineering. IEEE Computer **39**(2) (2006)
4. Calvary, G., Coutaz, J., Thevenin, D.: A unifying reference framework for the development of plastic user interfaces, Springer-Verlag (2001) 173–192
5. Ehrig, H., Habel, A., Kreowski, H.J., Parisi-Presicce, F.: Parallelism and concurrency in high-level replacement systems. Math. Struct. in Comp. Science **1** (1991) 361–404
6. Blostein, D., Schürr, A.: Computing with graphs and graph rewriting. Technical report, FACHGRUPPE INFORMATIK, RWTH (1997)
7. Hoffmann, K., Ehrig, H., Padberg, J.: Flexible modeling of emergency scenarios using reconfigurable systems. In: Proc. of the 10th World Conference on Integrated Design & Process Technology. (2007) 15 CDROM.
8. lim: Programmed graph replacement systems. In: In Rozenberg, G. (Ed.), Handbook on Graph Grammars: Foundations, World Scientific (1997) 479–546
9. Limbourg, Q.: Multi-Path Development of User Interfaces. PhD thesis, Université Catholique de Louvain, Institut d'Administration et de Gestion (IAG), Louvain-la-Neuve, Belgium (2004)
10. Morin, B., Barais, O., Nain, G., Jezequel, J.M.: Taming dynamically adaptive systems using models and aspects. In: ICSE '09: Proceedings of the 31st International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2009) 122–132
11. alain Muller, P., Fleurey, F., marc Jézéquel, J.: Weaving executability into object-oriented meta-languages. In: in: International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713 (2005, Springer (2005) 264–278

Using Models at Runtime For Monitoring and Adaptation of Networked Physical Devices: Example of a Flexible Manufacturing System

Mathieu Vallée[1], Munir Merdan[2], and Thomas Moser[3]

[1] Institute of Computer Technologies
[2] Automation and Control Institute
[3] Institute for Software Technology and Interactive Systems
Vienna University of Technology, Austria
{firstname.lastname}@tuwien.ac.at

Abstract. The emergence of software-intensive systems connecting physical devices to network-based applications involves new design challenges. As an example, flexible manufacturing systems composed of multiple networked devices in interaction with the physical world, are subject to imprecision and to unpredictable breakdowns. Applications and control software are therefore highly complex, and must operate in heterogeneous and rapidly changing environments.
To address these issues, we describe an approach using models at runtime for efficiently monitoring and adapting the software controlling mechatronic devices. We consider a decentralized system, in which each device is represented as an agent. Each agent maintains a model integrating a representation of itself, of its environment and of the agent society, and uses this model to detect inconsistencies, to envision possible future states and to create explanations based on past states. In this paper, we focus on presenting our model and highlighting the results, benefits and challenges arising from using models at run-time with networked physical devices.

1 Introduction

The emergence of software-intensive systems connecting physical devices to network-based applications offers new exciting possibilities. Among them, flexible manufacturing systems providing a faster, more efficient response to market changes are envisioned [10]. However, engineering such complex systems operating in heterogeneous and rapidly changing environments poses numerous challenges. Traditional control approaches cannot cope with new requirements, due to their rigidity and limited capability for agile adaptation to unexpected internal and external disturbances [8]. The application of decentralized control architectures, based on autonomous and co-operative agents, is considered as a promising approach. Intelligent agents offer a convenient way of modeling processes that are distributed over space and time, making the control of the system decentralized [7], increasing flexibility and enhances fault tolerance. Using agent-based

software for controlling a flexible manufacturing system has been largely investigated in the recent years. However, most work focus on planning and scheduling issues [8] (disconnected from the actual control of physical devices [3]) or use simple, reactive agents for producing specialized adaptation behaviors. Currently, concerns about robustness and stability prevent the wide industrial adoption of these initial solutions [15]. Advances on self-awareness, self-adaptation and self-healing are needed [4].

To address these issues, we proposed an approach in which agents use models for efficiently monitoring and adapting the software controlling mechatronic devices. Each agent manages a model integrating a representation of itself, of its environment and of the agent society, and enabling it to detect inconsistencies, to envision possible future states and to create explanations based on past states. In recent works [18, 11, 9], we showed how this approach supports complex tasks such as detection of anomalies of sensor reading, failure recovery and runtime reconfiguration. In this paper, we focus on presenting our use of models at runtime with networked physical devices and discussing the challenges arising from this approach.

This paper is structured as follows. Section 2 introduces some background and example about distributed intelligent control of a flexible manufacturing system. Section 3 details the world model of an automation agent, forming the central piece of our approach. Section 4 discusses our results and lessons learned. Section 5 discussed related work and section 6 concludes with a summary.

2 Background: Distributed Intelligent Control of a Flexible Pallet Transfer System

As an example for monitoring and adapting software in interaction with networked physical devices, our current studies focuses on flexible manufacturing systems. In this paper, we use the example of a pallet transport system, located in the Odo-Struder-Laboratory[4]. Due to their role to connect different parts of the system and to carry and route materials between them, transportation systems are in most cases seen as a key element, but the increasing need for more flexibility significantly complicates the control of these systems.

Overview of the Pallet Transfer System The pallet transfer system (Fig. 1) consists of software-controlled manufacturing components: transport components such as conveyor belts (dark green lines) and diverters (yellow circles); and assembly machines (colored rectangles with round corners). Product parts are transported on pallets (colored rectangles; colors represent the target machines). Each pallet carries an RFID tag providing information on its destinations, which diverters can access through RFID readers (rectangles on conveyors). Figure 2

[4] Industrial automation systems laboratory of the Automation and Control Institute, Vienna University of Technology

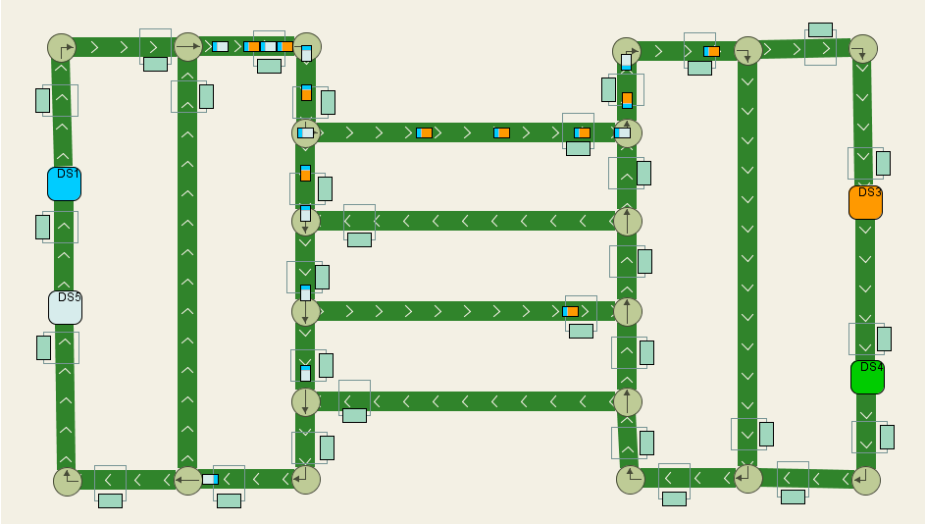
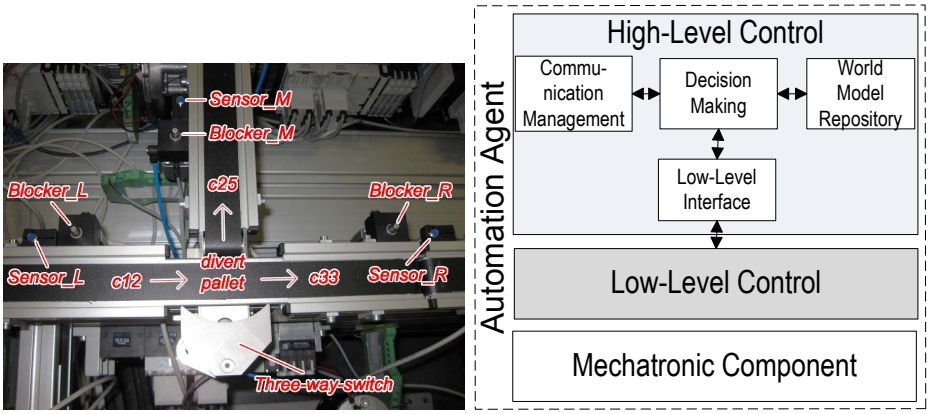


Fig. 1. Overview of the pallet transfer system



(a) Mechatronic component (diverter) (b) Architecture of the automation agent

Fig. 2. Automation agent controlling a diverter

(a) depicts the mechatronic component realizing a diverter. It is mainly composed of a switch (directing a pallet), sensors (detecting the presence of a pallet) and blockers (preventing a pallet from moving).

Architecture of an Automation Agent In order to support the design of distributed intelligent control software for manufacturing systems, we introduced a generic architecture for automation agents in [17]. The architecture is depicted on Fig. 2 (b), and consists of two software layers, in addition to the mechatronic component. The low-level control (LLC) layer is in charge of controlling the hardware. The high-level control (HLC) layer is in charge of diagnostics, of coordination with other agents and of self-adaption based on the representation of the world. In the case of the diverter, a "diverter agent" contains a LLC layer responsible for moving the switch depending on the destination of on incoming pallets, and a HLC layer responsible for , e.g, redefining routes in response to disturbances at other components or validating sensor readings with information from other agents. Distinguishing LLC and HLC within each automation agent is fundamental for devices in interaction with the physical world. In our architecture, the LLC is responsible for performing all necessary operations in real-time, while the HLC is only responsible for non-functional monitoring and adaptation, which might require longer computation time and interactions with others agents or even with human operators. To enforce this layering while enabling efficient adaptation, we base our LLC on the IEC 61499 standard, enhanced with programmable reconfigurations capabilities [22, 9].

Figure 2 (b) also depicts more precisely the four main modules composing the inner architecture of the HLC. The world model repository contains a world model, i.e., a symbolic representation of the world of the agent. The low-level interface enables the HLC to monitor and to adapt the LLC. It especially provides facilities for receiving event notifications about the current operations of the LLC and for requesting reconfiguration in the LLC. The communication manager provides facilities for managing the communication with other agents. The decision-making component is in charge of coordinating the reasoning about states of the world and deciding what to do (e.g., communicate with other machines, request an operation from the LLC, issue notifications to an operator). Event notifications generated by the LLC, by communication with other agents or by the world model trigger the decision-making procedures.

3 World Model of an Automation Agent

The world model plays a central role in the architecture of an automation agent. In this section, we describe its content and illustrate it using the example of the diverter agent.

3.1 Properties

The world model has to provide two key properties. Firstly it should *integrate information about different views* of the world, which are sometimes overlapping.

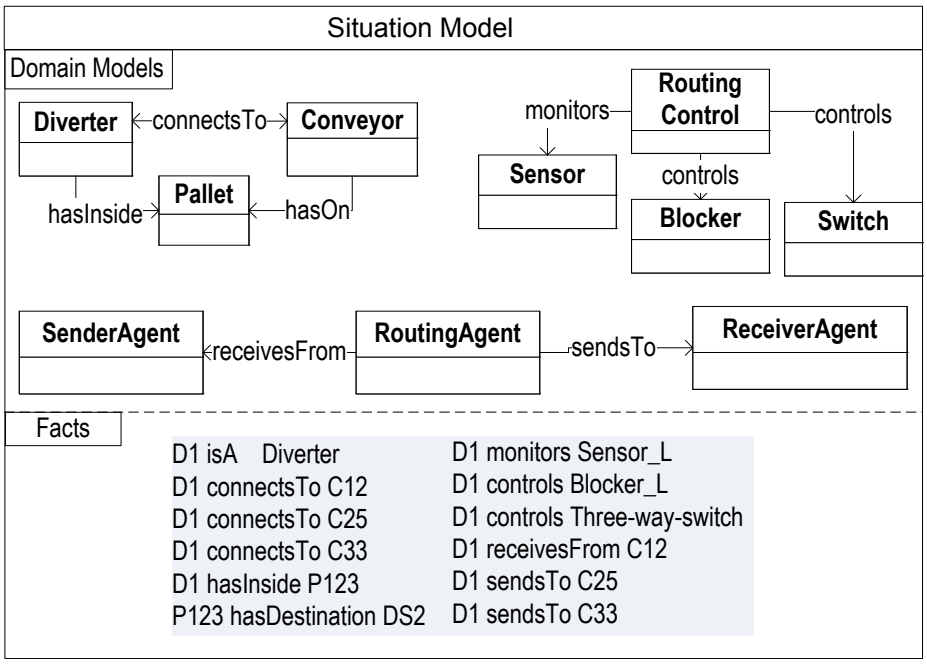


Fig. 3. Situation Model for the diverter agent

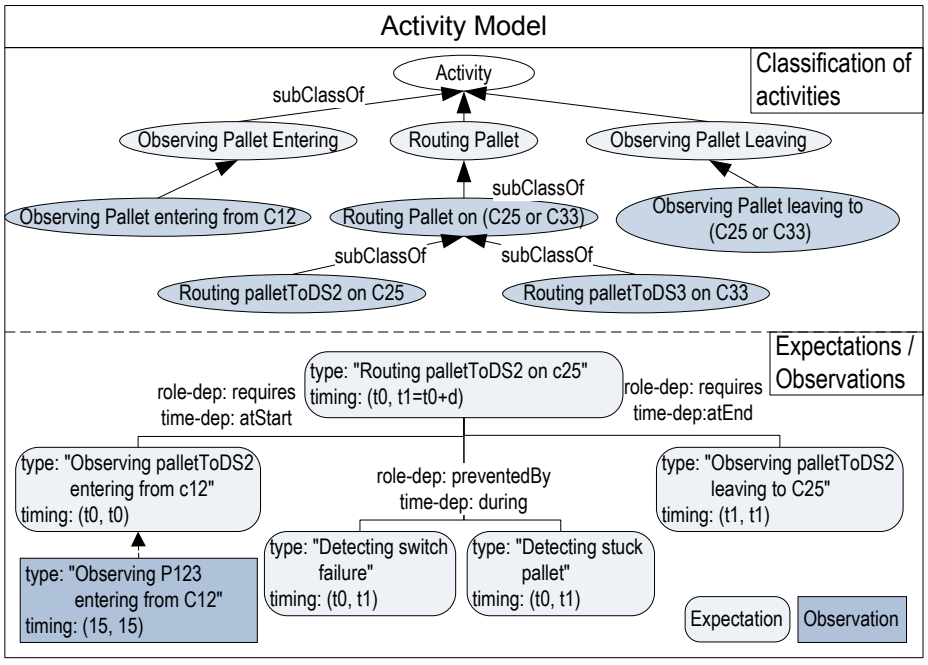


Fig. 4. Activity Model for the diverter agent

It must integrate information about the state of its environment (the physical world in which it is evolving), about the agent society (the other agents with which it is interacting, their roles and tasks), and about its own internal structure and processes. Clearly, this is related to reflection, so the representation of the agent itself is a key element of the world model [17].

Secondly and most importantly, the world model must support a *flexible synchronization* with the real world. In particular, it is in general not possible to assume that the model provides a complete and up-to-date view on the real world. We must cope with partial and scarce observations, and we use models to compensate for the lack of direct information with assumptions about what the state of the world should be. As a consequence, the model must be validated and revised anytime new information is received. More precisely, it should provide three key features:

- The *detection of inconsistencies* between the current world model and new information (received from LLC or other agents). In a real world setting, inconsistencies may arise both from inaccuracy of the model on the one side, and from imprecision of the information sources on the other side.
- The *derivation of possible future states* of the world and their relevant characteristics (answering "what-if" questions). It should be possible to define expectations about the future state of the world, and to plan meaningful observations accordingly. It should also be possible to envision multiple possibilities about the future in order to be prepared for adaptation.
- The *derivation of explanations* from past states of the world (answering "why" questions). Although all information about the world may not be accessible, models can be used to explain current observations with assumption about past states of the world which could not be observed directly. In some casesw, this can trigger additional observation to confirm assumptions. This is particularly useful for diagnosis, when root causes for failures can be identified from reasoning on the world model.

3.2 Elements of the World Model

The world model consists of two parts. Figures 3 and 4 illustrate the world model for the diverter agent example.

The *situation model* (Fig. 3) holds knowledge about the agent situation. The situation of an agent consists both of its own characteristics and its relations to other entities in the world. The *domain models* (top) are models of the type of entities in the domain of the agent. They defines relevant classes of entities as well as relations between entities. For our example, we define that a diverter is connected to conveyors and can have a pallet located inside. Such concepts and relations can be extracted from existing models, such as the one presented in [12]. The *facts* (bottom) express the current knowledge about the world. Facts are expressed using the vocabulary defined by the models. They represent an abstraction of some meaningful aspects of the world, which can be used for

realizing high-level control tasks. For our example, facts express that D1 is a diverter, which is connected to conveyors C12, C25 and C33.

The *activity model* (Fig. 4) contains knowledge about the activities of the agent, i.e., the events and processes occurring in the world in which the agent is participating (as actor or observer). The *classification of activities*(top) models the types of activities in which the agent can be involved. Types are defined formally using description logic formulas and are organized hierarchically based on the subsumption relationship [2], noted $subClassOf$. Primitive types are defined as direct subclasses of Activity. Derived types are defined by restricting the primitive types to take into account the actual world of the agent. For instance, the generic type "Routing Pallet" is refined to more specific types like "Routing palletToDS2 on C25" corresponding to the case of the diverter agent.

The *expectations and observations* (bottom) model the activities that are expected and observed by the agent. Expectations and observations are defined by the specification of a type (based on the classification of activity types) and timing, expressed using time intervals [1]. Expectations are linked by dependencies, indicating how observations on one expectation can have consequences on other expectations. For instance, it is expected that "Routing palletToDS2 on C25", taking place between $t0$ and $t1$, requires both that "Observing Pallet entering from C12" takes place at $t0$ and "Observing Pallet leaving to C25" takes place at $t1$ (with a given tolerance). Additionally, it is expected that this activity would be prevented by "Detecting Switch Failure" during the same interval of time. Assuming that a pallet P123, with destination DS2, enters the diverter, an observation is added to the model, indicating that the activity "Observing P123 entering from C12" takes place at time 15.

3.3 Runtime Synchronization

The world model, and in particular the model of expectations and observations about activities, is synchronized incrementally, whenever new information is received from the LLC or from other agents. It is thus constantly evolving at runtime to reflect the current knowledge about the world as well as the current expectations that could be derived from this knowledge. Conversely, the changes in the model can be reflected in the underlying software, especially thanks to the LLC reconfiguration abilities.

Figure 5 depicts the general workflow for updating the model:

1. *Integration.* Whenever new information about the world is available, it is integrated in the world model by expressing the related type of activity and timing.
2. *Identification.* Forming a new observation requires identifying how it relates to existing expectations in the model. A matching is performed using the type and timing information. In case no expectation can be identified, an anomaly is reported.
3. *Propagation.* The addition of a new observation may trigger the creation of new expectations, Propagation occurs by considering dependencies be-

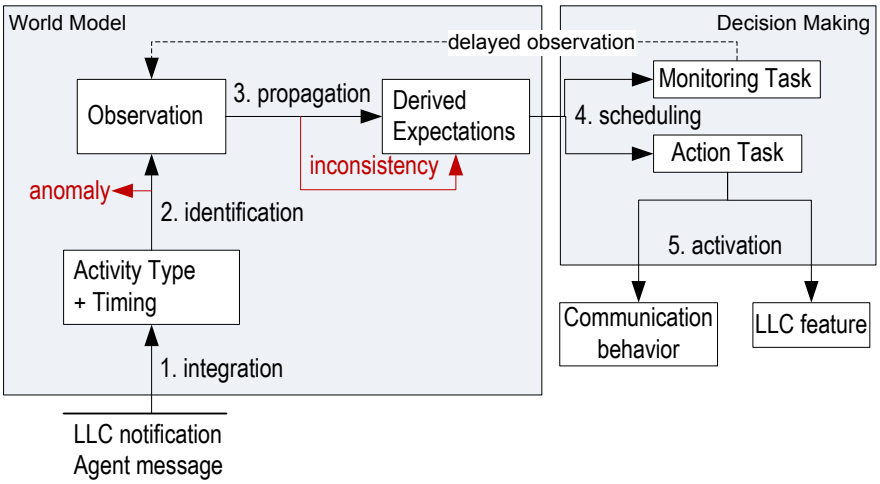


Fig. 5. Runtime synchronization of the world model

tween expectations and creating new expectations if needed. At this step, inconsistencies may be detected.

4. *Scheduling.* When new expectations are added to the world model, new decision-making tasks may be added to reflect them. We use monitoring tasks to trigger observations based on timing constraints (typically, these are observations about something that did not happen). Action tasks trigger external actions, either from the LLC or from other agents.
5. *Activation.* Relevant changes and anomalies in the activity model are notified to the decision making component, which is in charge of initiating the appropriate actions. Typical actions are setting up a communication behavior (i.e., initiating/terminating an interaction or cooperation protocol with other agents) or a LLC feature (i.e., adding/removing components in the LLC).

4 Results and Lessons Learned

We used the automation agent architecture and the world model as a basis for designing flexible manufacturing systems. In this section, we summarize our results by giving an overview of tasks involving the world model at runtime. We then discuss some important points and lessons learned.

4.1 Benefits of using a Model at Runtime

We designed the automation agent architecture and its world model to be generic and to apply to different classes of problems. Indeed, we could address several issues in a flexible manufacturing system using the model described in the previous section.

Detection of anomalies When interacting with the physical world, we are constantly faced with anomalies, disturbances and failures. Detecting anomalies before they cause more critical disturbances and failures is a definite advantage. Using its world model, an automation agent is able to detect anomalies which would otherwise be unnoticed by classical control software. For instance, we showed in [18] how an automation agent models its expectation about the completion of a transport task, and monitors relevant sensors to verify it. In case the sensor reading does not occur as expected, an anomaly is raised, indicating that a pallet is possibly stuck, or that the sensor in not working properly. Such a mechanism also benefits from the decentralized approach, enabling scalability and direct detection close to the relevant hardware.

Online diagnostics To enable the robust operation of a flexible manufacturing system, diagnostics and fault-recovery mechanisms are needed. The presented world model is especially helpful for the identifications of causes for a failures, and supports searching for explanations when an anomaly is observed [11]. This mechanism relies on defining expectations that could lead to the observation, and trying to verify them. Several directions can be exploited for verifying an expectation, for instance self-testing (e.g., trying to detect if a pallet was stuck by moving the switch to release it) and cooperation with other agents (e.g., asking a neighboring agent whether it detected a pallet which seems lost).

Runtime reconfiguration One of the most advanced features of a flexible manufacturing system is runtime reconfiguration, which is especially helpful to address challenges posed by an heterogeneous and continuously changing environment. As an example, in a pallet transport system, a destination may become unreachable due to the unexpected breakdown of a component. In order to keep the system running, we have studied solutions based on local reconfiguration, enabling conveyor belts to run in the opposite direction and intersections to modify their routing behavior. This requires a profound reconfiguration of the low-level control software, which our architecture allows. As presented in [9], the world model is directly involved in this task, both for identifying the need for reconfiguration and for preparing reconfiguration operations.

4.2 Lessons Learned

Besides the presented benefits, we can point out some lessons learned. They underline some important issues about the usage of models at runtime with networked physical devices, such as the ones encountered in a flexible manufacturing system. We identify three main challenges:

Challenge 1: Modeling dynamic aspects of the world. For dealing with a physical system, modeling dynamics is very important. A static view, even if regularly updated, is insufficient, as relevant information may not be accessible or integrated it in a timely manner. Modeling dynamic aspects enable

to obtain information from reasoning rather than from direct observation. Models and formal methods for managing time, time dependencies [21] as well as time imprecision are required.

Challenge 2: Synchronizing models incrementally. Physical devices operate under time constraints, and real-time execution is often incompatible with expensive model-based representation and reasoning. In order to cope with time constraints, we adopted an approach in which a fast LLC is fully responsible for performing all the functional operations of the system, while the slower HLC only performs complementary tasks to adapt and improve the behavior of the system. We found this approach suitable, but it also brings new challenges in terms of how the world model can reflect the reality while having only intermittent access to information from the world, and how it can synchronize to the real-world. Incremental synchronization of a model at runtime is essential [19].

Challenge 3: Integrating models in evolving systems. Working with networked physical devices requires the management of fragmented models over distributed agents. Moreover, large-scale systems require components to evolve independently. Ontologies are a general solution for interoperability [13], but are often unsuitable at runtime, since processing is overly complex. Considering that the system is rarely open, we consider just-in-time model-based generation of adapters and ad-hoc classifiers as more efficient, while ontologies provide a suitable abstraction for designers at design time.

5 Related Work

Model-driven engineering in gradually taking up in manufacturing systems [16]. However, these efforts mostly focus on models at design time, and do not seek to address issues regarding flexibility and robustness at runtime.

Previous works on using models at runtime are therefore highly relevant to our work. The general approach of using reasoning on a model to reconfigure a component-based system was already described by Oreizy *et al.* [14]. More recent effort have been focusing on modeling variability and adaptation in this approach in a generic way [5], providing a basis for the specification and validation of dynamic adaptive systems. One of the shortcomings for a direct application of such solutions in our domain is the lack of modeling of the dynamic behavior of the system, which we require for anticipating future states, detecting anomalies, as well as diagnosing past states in the presence of limited observations. Some works address more directly the behavioral modeling of some aspects of a dynamic adaptive system [20], as well as model-based runtime detection of errors [6]. However, we are not aware of a general solution for modeling activities among a distributed system of networked devices, which is required in our case.

6 Summary

In this paper, we presented an approach in which automation agents use models for efficiently monitoring and adapting the software controlling mechatronic

devices. Each agent manages a model integrating a representation of itself, of its environment and of the agent society, and enabling it to detect inconsistencies, to envision possible future states and to create explanations based on past states. We detailed the generic architecture and the model we use for representing the world of an agent. This model features a static part, called the situation model, and a dynamic part, called the activity model. One essential feature of our approach lies in the incremental synchronization of the activity model using information from low-level control software and from other agents.

As a further contribution of this paper, we presented results and lessons learned in this work. We have showed that the proposed approach using a model at runtime is the basis for monitoring and adapting control software in a flexible manufacturing systems. It provides significant improvement in terms of flexibility, robustness and performance. However, we point out that this approach raises new challenges regarding modeling dynamic aspects of the world, synchronizing models incrementally, and integrating models in evolving systems. Although our work partially addresses this challenges, further research in these directions is needed.

References

1. Allen, J.: Maintaining knowledge about temporal intervals. Communications of the ACM 26(11), 832–843 (1983)
2. Baader, F., Horrock, I., Sattler, U.: Description logics. In: Handbook on ontologies, pp. 3–28. Springer (2004)
3. Brennan, R.: Toward Real-Time distributed intelligent control: A survey of research themes and applications. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 37(5), 744–765 (2007)
4. Chituc, C.M., Restivo, F.J.: Challenges and trends in distributed manufacturing systems: Are wise engineering systems the ultimate answer? In: Second International Symposium on Engineering Systems MIT, Cambridge, Massachusetts (2009)
5. Fleurey, F., Dehlen, V., Bencomo, N., Morin, B., Jzquel, J.M.: Modeling and validating dynamic adaptation. In: Models in Software Engineering, pp. 97–108. Springer (2009), http://dx.doi.org/10.1007/978-3-642-01648-6_11
6. Hooman, J., Hendriks, T.: Model-based run-time error detection. In: Second International Workshop on Models@run.time (2007)
7. Jennings, N., Bussmann, S.: Agent-based control systems: Why are they suited to engineering complex systems? IEEE Control Systems Magazine 23(3), 61–73 (2003), http://dx.doi.org/10.1109/MCS.2003.1200249
8. Leitão, P.: Agent-based distributed manufacturing control: A state-of-the-art survey. Engineering Applications of Artificial Intelligence 22(7), 979–991 (2009), http://dx.doi.org/10.1016/j.engappai.2008.09.005
9. Lepuschitz, W., Zoitl, A., Vallée, M., Merdan, M.: Towards self-reconfiguration of manufacturing systems using automation agents. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews IN PRESS, 1–18 (2010), http://dx.doi.org/10.1109/TSMCC.2010.2059012
10. McFarlane, D., Marik, V., Valckenaers, P.: Guest editors' introduction: Intelligent control in the manufacturing supply chain. IEEE Intelligent Systems 20(1), 24–26 (2005), http://dx.doi.org/10.1109/MIS.2005.8

11. Merdan, M., Vallée, M., Lepuschitz, W., Zoitl, A.: Monitoring and diagnostics of industrial systems using automation agents. International Journal of Production Research Special Issue on Multi-agent and Holonic Techniques for Manufacturing Systems: Technologies and Applications, IN PRESS (2011)
12. Merdan, M., Koppensteiner, G., Hegny, I., Favre-Bulle, B.: Application of an ontology in a transport domain. In: IEEE International Conference on Industrial Technology (IEEE-ICIT 2008). Sichuan University, Chengdu, China (2008), http://dx.doi.org/10.1109/ICIT.2008.4608572
13. Obitko, M., Vrba, P., Mark, V., Radakovic, M.: Semantics in industrial distributed systems. In: IFAC (2006)
14. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems and Their Applications 14(3), 054–62 (1999), http://dx.doi.org/10.1109/5254.769885
15. Pěchouček, M., Mařík, V.: Industrial deployment of multi-agent technologies: review and selected case studies. Autonomous Agents and Multi-Agent Systems 17(3), 397–431 (Dec 2008), http://dx.doi.org/10.1007/s10458-008-9050-0
16. Strasser, T., Rooker, M., Hegny, I., Wenger, M., Zoitl, A., Ferrarini, L., Dede, A., Colla, M.: A research roadmap for model-driven design of embedded systems for automation components. In: 7th IEEE International Conference onIndustrial Informatics (INDIN 2009). pp. 564 –569 (jun 2009)
17. Vallée, M., Kaindl, H., Merdan, M., Lepuschitz, W., Arnautovic, E., Vrba, P.: An automation agent architecture with a reflective world model in manufacturing systems. In: IEEE International Conference on Systems, Man, and Cybernetics (SMC09). San Antonio, Texas, USA. (2009), http://dx.doi.org/10.1109/ICSMC.2009.5346161
18. Vallée, M., Merdan, M., Vrba, P.: Detection of anomalies in a transport system using automation agents with a reflective world model. In: Proceedings of the IEEE International Conference on Industrial Technologies (IEEE-ICIT 2010). pp. 489 – 494. Viña del Mar-Valparaso, Chile (2010), http://dx.doi.org/10.1109/ICIT.2010.5472751
19. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental model synchronization for efficient run-time monitoring. In: Fourth International Workshop on Models@run.time (2009), http://ceur-ws.org/Vol-509/paper_8.pdf
20. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: International Conference on Software Engineering (ICSE'06). China (2006), http://dx.doi.org/10.1145/1134285.1134337
21. Zhang, J., Cheng, B., , Goldsby, H.: Amoeba-rt: Run-time verification of adaptive software. In: Second International Workshop on Models@run.time (2007)
22. Zoitl, A.: Real-Time Execution for IEC 61499. No. ISBN: 978193439-4274, ISA-o3neidaA, USA (2009)

Monitoring Executions on Reconfigurable Hardware at Model Level

Tobias Schwalb[1], Philipp Graf[2], and Klaus D. Müller-Glaser[1]

[1] Karlsruhe Institute of Technology, Institute for Information Processing Technology, Germany, {tobias.schwalb,klaus.mueller-glaser}@kit.edu
[2] FZI Forschungszentrum Informatik, Germany, graf@fzi.de

Abstract. Development, debugging and test of embedded systems get more and more complex due to increasing size and complexity of implementations. To dominate this complexity, nowadays designs are often based on models. However, while the design is on model level, the monitoring and debugging are still either on signal or on code level. This paper presents a continuous concept that allows monitoring and real-time recording of executions on reconfigurable hardware at model level. Besides developed hardware debugger modules, a development environment has been integrated. It allows on model level generation of the implementation, control of the recording and monitoring at runtime and visualization of the execution. An algorithm, running in the background, maps acquired data from the hardware to the model and commands from the model-based development environment to the hardware. The method is demonstrated using the example of statechart diagram monitoring.

Keywords: debugging, model-based control, monitoring, back annotation, real-time recording

1 Introduction

In software and hardware development costs, time-to-market and quality are often contradicting to each other, which increase with the complexity of the system. The complexity arises mainly by the increasing demands in terms of functionality, energy efficiency and the ongoing integration on hardware. This affects especially the possibilities to monitor embedded systems at runtime. To dominate the increasing complexity, more and more abstract approaches for the development of embedded systems come up. One option is model-based development using graphical languages, for example Unified Modeling Language (UML)[1], statecharts[2] or signal flow graphs[3]. These models can be used to automatically generate source code for programming embedded systems.

To preserve the benefits of flexible development and simultaneously achieve high performance, reconfigurable hardware devices (e.g. Field Programmable Gate Arrays - FPGAs) are deployed. These offer the possibilities to implement algorithms in hardware and parallel execution, to gain more computing power. However, developing reconfigurable systems is more complex, especially in the area of debugging and testing with regard to real-time conditions.

The challenge is to combine these domains into a continuous model-based development process that allows debugging and monitoring on model level [4]. In a previous paper [5] an initial concept for model-based debugging of reconfigurable hardware has been presented. The paper focused mainly on the overall concept and the on-chip hardware architecture for real-time recording. This paper presents an advancement of the concept and focuses on the control of the functionality at runtime and the mapping between the different abstraction levels. In this context, the next chapter describes the state of the art in terms of monitoring and debugging reconfigurable hardware. In Section 3 the concept allowing model-based debugging on reconfigurable hardware is described, while Section 4 gives a brief overview of the on-chip architecture. The following section focuses on the software used for instantiation and runtime control. Section 6 introduces the method to achieve a mapping between hardware implementation and model to allow back annotation to model level and control of the debugging. The next section shows carried out tests and their results. We close with conclusions and outlook on future work in Section 8.

2 State of the Art

2.1 Simulation vs. Debugging

In the FPGA development process simulation[6] and debugging[7] are used for the identification of errors. Simulation, compared to debugging, has the advantage that a hardware system is not necessary, therefore it can be performed earlier in the development process. Using simulation, all signals of the design are directly accessible and their behavior can be displayed. Using debugging, the access to internal signals is limited, because the signals need to be either recorded on-chip (limited memory) or forwarded to output pins (limited number) for external processing. In general, an embedded system exists in context of its peripherals and surroundings. In a simulation all these need to be additionally integrated and it is very complex to consider all parameters. Therefore, there is always an uncertainty if the simulation represents the real system. With debugging, peripherals and surroundings are present in the real system and do not need to be simulated. In addition, using simulations it is difficult to determine non-functional parameters, for example real-time conditions or performance.

2.2 Debugging Reconfigurable Hardware

For debugging FPGAs [8] it has to be mainly distinguished between real-time and non real-time debugging. In the latter, breakpoints[3] stop the clock of the design under test or it is executed step by step. When the design stops, the status of the on-chip registers is read back and interpreted to determine the system state. Disadvantages are that performance and timing cannot be analyzed and it is not possible to obtain the status of signals before stopping the system. For real-time

[3] Configurable event triggered by a set of conditions that halt the system

debugging essentially two different methods exist: the first forwards the signals of interest to output pins of the FPGA. Recording and processing is performed by external hardware (e.g. digital logic analyzer). The number of signals is limited, because every signal requires an extra output pin. The advantage is that on-chip logic or memory is not needed. The second option integrates additional on-chip modules to record the signals and transfer the data via an interface to a PC. An example is ChipScope[9] by Xilinx, it stores the signal flow in on-chip Block-RAM and transfers data via the JTAG interface. This method can record many signals, but recording time is limited by on-chip memory.

All discussed debuggers work and get controlled during runtime on signal or code level, i.e. breakpoints or trigger conditions for recording are set with respect to the signals in the design. This it is not suitable for a model-based design process, because the developer designs the system on model level and does not know about signals or source code, as this is mostly automatically generated according to the developed model.

2.3 Model-based Debugging

Debugging on model level for embedded systems is possible, but not widely used. The commercial software Matlab[10] offers model-based debugging in their Stateflow part, it allows the implementation and debugging of statechart diagrams, but supports debugging only on special microprocessor platforms.

A general approach for model-based debugging on embedded systems has been presented in [11], [12], [13]. These papers describe a concept which refers to different abstraction layers in a model-based design process and a framework for a modular system architecture. Also a prototype implementation and a connection to a real-time in-circuit emulator are shown. In this paper, these principles are extended with real-time aspects within reconfigurable hardware and the automatic generation of the hardware debugging platform. In this context, a mapping between model and hardware platform is developed, that additionally allows the control of the debugging during runtime from model level. The concepts of the presented debugging environment [13] are integrated in a new developing environment that is based on model-based developing frameworks and extended with interactive control modules.

3 Model-based Design Flow for Debugging

The development of reconfigurable systems is heading towards a model-based design flow. Hence also monitoring, debugging and control of the debugging needs to take place on model level. A continuous concept for debugging on model level is depicted in Figure 1.

The flow shows on the left side the design and transformation, in the middle bottom the execution and on the right side the mapping and debugging. In the middle, a direct connection for model-based control is added. In the beginning, the user designs his system using different models. In this context, the models

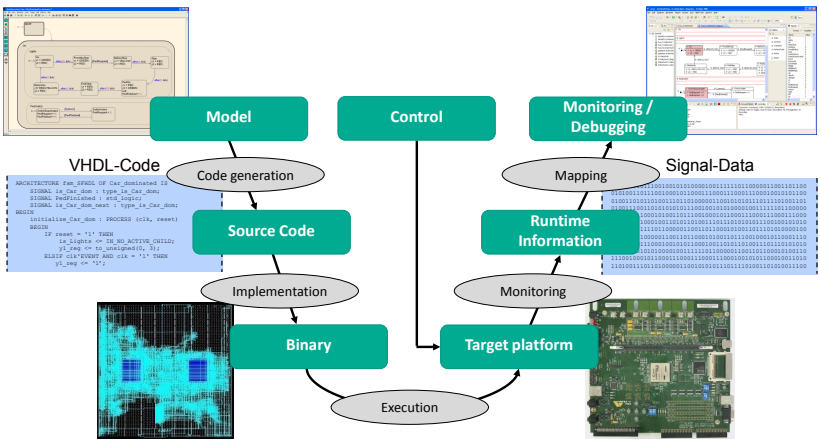


Fig. 1. Design flow for model-based monitoring and control [10],[14],[16]

have to be sufficiently detailed to generate executable source code. This automatic code generation is currently supported by various toolsets and mainly allows the generation of C code and partly HDL code. As we examine reconfigurable hardware, we concentrate on VHDL code. To enable debugging, the generation process needs to implement an interface for debugging into the hardware design. After generation, the design is synthesized by the FPGA-specific tools using mapping and routing algorithms to generate a bitstream, which is used to program the FPGA. The design, which includes the user implementation and additional debugger modules, is executed on a FPGA. During execution in the right branch of the process the signals of interest are captured by the debugger modules. The data is transferred to a PC during monitoring or after recording. Since the data obtained on-chip relates to signal level, it is mapped to the model. In visualization the user can monitor the execution.

During the whole process, the user is working on model level, the intermediate steps are performed by algorithms. Therefore, developing, debugging and control of the debugging take place on the same abstraction level. The automatic interpretation and mapping of signals to the model cannot be regarded as reverse engineering, since information from the left branch of the process is needed. It is rather a reversal of the transformation from model to hardware.

4 On-chip Architecture

The modular architecture of the debugger modules is depicted in Figure 2. First the recorded signals of the designs under test are buffered in a FIFO. This first FIFO can also run in a ring buffer mode, which allows recording signals before a trigger is released. This mode allows easier identification of the reason of an error, because the history of events can be recorded and parts of the system state reconstructed. After the FIFO, the data can be processed by different

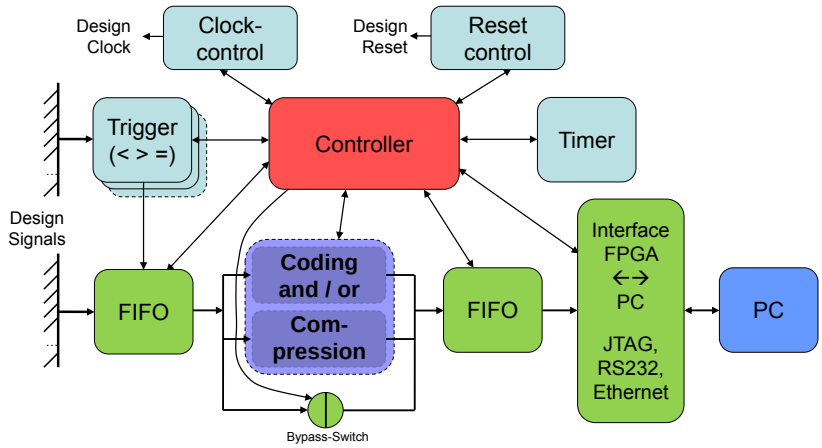


Fig. 2. Architecture of the on-chip debugger modules

compression or coding algorithms to get a reduction of the data. This processing is optional, but allows to use on-chip memory more efficiently and record a longer period of time. The second FIFO stores the data into on-chip memory, before it is transmitted to a PC using a communication interface. This architecture is described in more detail in [5]. In comparison to the original design, the Pseudo-Random-Generator is excluded and the coding and compression unit is shirked to decrease the use of logic resources. The DDR-Interface is no longer supported, because of its speed in comparison with internal memory in the actual system.

Besides recording, a direct monitoring of the design signals is also possible, since the transmission is independent of the recording. In this mode compression is bypassed and the FIFOs are directly read out. However, a restriction is the bandwidth, which depends on the speed of the interface to the PC and the processing in the model-based development environment. Therefore, real-time monitoring is only possible within small systems with few signal changes.

The debugger is managed by the controller, which is a small microprocessor. It monitors the status, controls recording and the design under test as well as communicates with the PC. To control the design under test, its *clock enable* and *reset* signal can be changed, which allows to stop and reset the design independent of the debugger modules. As inputs, trigger modules monitor signals of the design under test on the occurrence of certain conditions to start or stop the recording. The modules can trigger on edges or conditions of signals as well as compare signals to other signals or with fixed values. The trigger module can be repeatedly instantiated to allow complex chained comparisons. Additionally, the trigger conditions can be modified at runtime, switching integrated multiplexers or changing memory cells. The controller also communicates with the PC, receiving commands for control and transmitting recorded data. The design has been extended by a timer module to enable time-based recordings.

5 Software and Model-based Control

For model-based generation of the platform, visualization of the execution and control of the debugging at runtime a model-based developing environment has been implemented. The environment is build on Eclipse[14], extended with the Eclipse Modeling Framework (EMF) and Graphical Modeling Framework (GMF) for model-based design and the xPand framework for code generation.

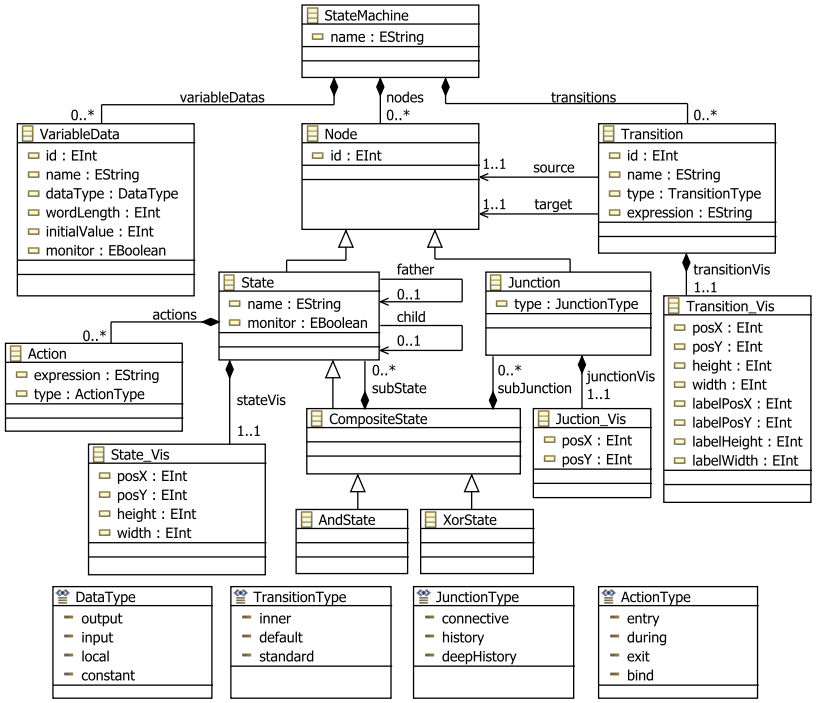


Fig. 3. Meta model for statechart diagrams with monitoring extensions

With regard to statecharts in the first step, a meta model has been developed (Figure 3). As we use Eclipse the meta model is based on the Ecore meta meta model. The meta model relies on the design of statecharts in Matlab Stateflow to get the opportunity to convert Matlab files into the developed IDE. It is similar to the UML statechart meta model [1], but simplified concerning states and transitions. The additional class *VariableData* keeps the inputs, outputs and internal variables that are used within the statechart for communication. All main classes, namely *Node*, *Transition* and *VariableData* have an attribute *id*, which allows direct identification of the individual element. The classes on the bottom show enum-classes defining different types of elements within the meta model. The additional visualization classes (*..._Vis*) store layout information, if

a Matlab Stateflow model is converted. The flag *monitor* integrated in the class *State* and *VariableData* is used during the generation of the platform to specify the monitored instances.

According to the meta model, three models are created in the GMF-framework concerning the graphical editor on model level. The first model describes the palette in the editor, i.e. the tools that are available to modify the model. In the example, there are tools to draw simple states, xor-states, and-states and junctions as well as transitions between states. The second model, the gmfgraph model, describes the graphical representation of the elements in the model, i.e. their shape, color etc. The last model layouts a mapping between the three models, it creates a connection between elements in meta model, tools and graphical representations. After creation of these models a model-based IDE can be generated by the framework. The result is depicted in Figure 4 (middle and left part). In the middle, is the modeling area with the tool palette and on the left side is the project management. The windows on the bottom and on the right are additionally implemented and explained in the next paragraph.

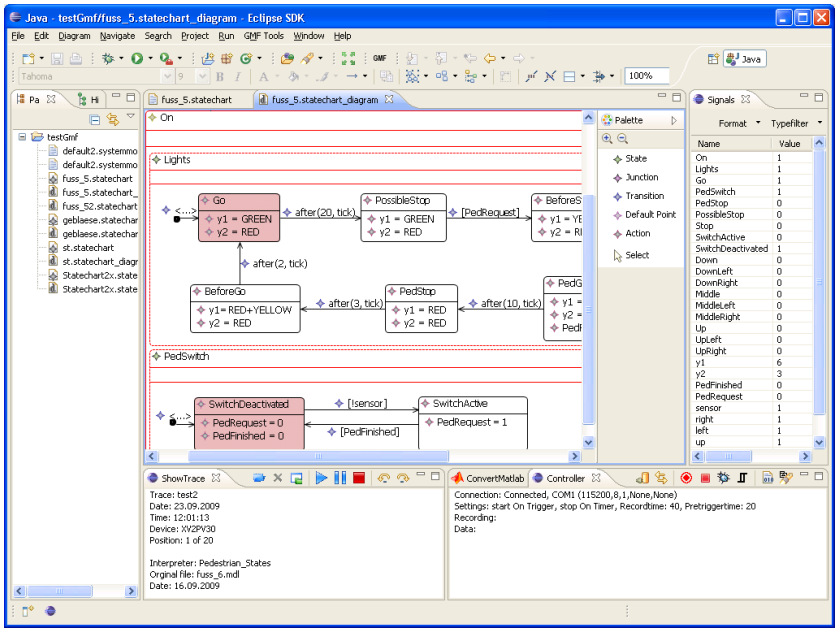


Fig. 4. Model-based development environment

The window on the right allows displaying all recorded data with regard to the model. It displays in addition to the active states, which are shown in the model, the monitored values of internal variables, inputs and outputs. The view in the middle below can be used to visualize executions that have already been recorded and saved to a file. The window on the right below is the controller

for on-chip recording and monitoring. It integrates a connection to the on-chip debugger to send commands and receive data during runtime.

The controller window allows controlling the *reset* and *clock enable* signal of the design under test. To control the recording start- and stop-conditions can be specified. The recording can start on the trigger, with the start of the design or manually. However, in this context, a manual start does not fulfill real-time conditions, because the time between the click and the actual start of the recording in the system cannot be exactly determined. The recording can stop either on a second trigger, on a timer or when the recording memory is full. Using the timer the recording time can be specified with regard to the number of clock cycles. Also the pretrigger time can be specified the same way.

Additionally, the trigger conditions are specified in the controller window. These are described with regard to the model, for example the string *in(Go) and in(SwitchActive)* would specify the trigger to release when state *Go* and *SwitchActive* are active at the same time. The trigger can be specified according to states, inputs, outputs and internal variables. The complexity have to match the implemented number of trigger modules, i.e. if the trigger condition is compound from three statements also minimal three trigger modules have to be present in the hardware implementation. In the next step, the design under test can be started, i.e. the *clock enable* signal is released and/or a *reset* performed. When recording is finished the data is transferred and stored in a XML file, which can be directly visualized to perform a postmortem analysis[4]. This enables model-based real-time debugging, but as the parameters have to be setup before recording, some knowledge according to the error needs to be present.

In another option, the controller can directly monitor the execution on-chip. The status of all recorded signals are polled every 100ms and the transmitted data is directly interpreted and visualized. The clock in the design under test can run continuously or controlled step by step to enable slow execution. No real-time debugging is possible using direct monitoring, because it only allows either slow execution (step by step) or slow monitoring (every 100ms).

6 Mapping of Model and Hardware Implementation

In a model-based development process the design of the system is on model level and the source code for programming is mostly automatically generated from the model. If there is an error in the system, the user wants to monitor and debug his system on model level - the same level it has been designed. However, the data on a FPGA relies on signal level, therefore a mapping between model and hardware is needed. In addition, in a FPGA internal signals representing model elements are not directly accessible, therefore during generation of the system an adapted debugging interface needs to be integrated.

The design flow for generation of the platform and mapping of model and hardware system is depicted in Figure 5. In the example Matlab Stateflow is

[4] Analysis that is performed after an expected event (e.g. an error / a system crash)

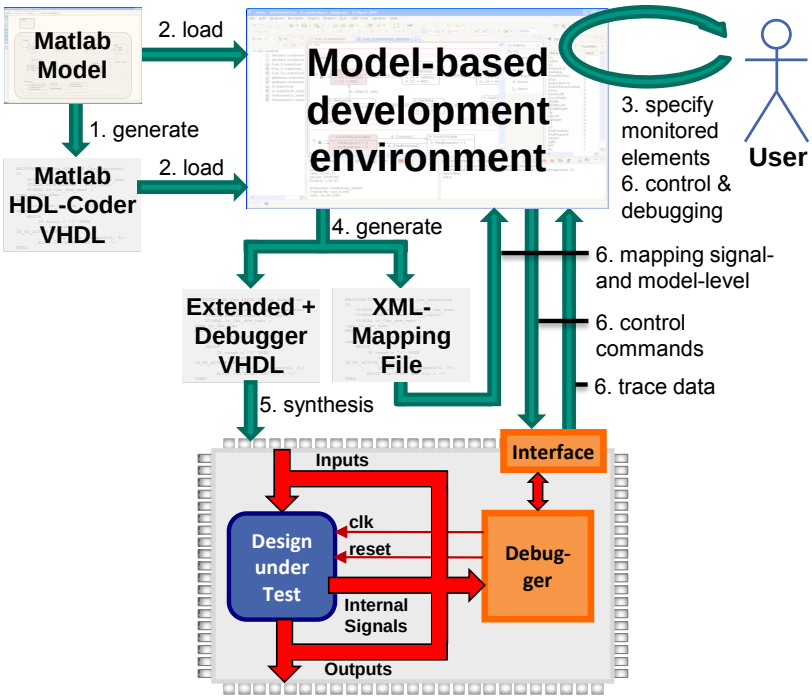


Fig. 5. Flow for platform generation and mapping of model and hardware

used to design the statechart diagram and the Matlab HDL-Coder to generate the VHDL code, which represents the functionality of the statechart. If the model is designed and the VHDL code generated, both is load into the described development environment (see Section 5). The Matlab file is converted to an EMF model file which is based on the meta model shown in Figure 3. For efficient use of the FPGA resources in the next step the user can specify in the model the monitored elements.

According to the specifications in the model, the (by Matlab) generated VHDL code is extended to enable monitoring of inner states and variables. Transitions cannot be monitored directly, but as they form connections between states, according to changes from one state to the next, the used transition can be determined. The monitored signals are grouped together into two vectors, one for recording and a second for the trigger, and integrated as outputs in the VHDL code. The debugger modules are connected to the statechart module by a generated interface file. The interface is a top level VHDL structural description and connects signals from outside to signals in the design and specifies signals between the debugger modules and the statechart module. As all names of the signals are known or read from the Matlab file, the interface file can be directly generated.

To get the mapping between signals and types in the VHDL file and elements in the model, an algorithm has been developed. This algorithm uses the fact, that in the VHDL code all signals and types have the same name as in the model and that all states in a composite state are grouped together. Therefore, as the model and VHDL always follow the same principles the mapping can be identified. This mapping is stored in a XML file, because it is needed later during debugging. The file contains the name, position, size and the function of the signals in the vectors with regard to the elements in the model. The file also contains general information concerning the model, which later allows later an easier identification.

The generated VDHL code is in the next step synthesized by FPGA specific tools to generate a bitstream, which is integrated on the hardware. When the design is executed the debugger modules record the specified signals. The debugger modules are independent of the model and all signals, as described, are forwarded to the debugger in a vector. Therefore, during debugging additional information is needed for visualization of the execution on model level and control of the debugging (e.g. setting the trigger conditions). This information is contained in the XML file generated in the previous step. Therefore, the user can debug the system on model level and control the debugging according to the model notation. The back annotation of the recorded signals to the model and generation of control commands is performed by algorithms in the background using the mapping information.

In general, the back annotation from hardware to the model follows the same principles as the back annotation in a general software debugger [15], after transmitting the data gained on low level, it is combined with the mapping information to get a representation on high level. However, with respect to reconfigurable hardware it needs to be regard that processes can run in parallel and that a single element in the model can be represented by many signals. Also the coding and interpretation of the signals (binary, high-active, low-active, ...) according to the status of the elements in the model needs to be considered.

7 Integration and Test

Different tests have been carried out to evaluate the functionality and integrity of the depicted method and platform. The tests are mainly performed using a development board, including a Xilinx Virtex II Pro FPGA [16] as well as interfaces for communication and programming. The size of the debugger modules is variable and depends on the number of signals for recording and triggering as well as the possible recording length. It also depends on the number of trigger modules (i.e. the possible complexity of the trigger condition) and the type of compression unit.

Different designs were evaluated from small models used with minimal debugger modules up to large systems recording 128 signals. A common example is described in more detail. It is based on a model, which describes the traffic light system at a crosswalk and includes 14 states, 1 input, 2 outputs and 2 internal

variables. The model is designed in Matlab Stateflow and according VHDL code generated. Both is load in the model-based development environment (see model in Figure 4). In the example every element in the model is selected for recording. After specification the according VHDL and XML files are generated. The VHDL design is integrated on the FPGA using the Xilinx ISE design suite, some additional adjustments are carried out according to the FPGA, the clock signal and external connections. The debugger modules record altogether 32 signals at 100MHz with a depth of 1024 clock cycles. Therefore, the recording data rate is 3.2Gbit/s. After recording or during monitoring the data is transferred to the PC using a RS232 interface. In the example the debugger modules use approximately 4% of the FPGA resources. The debugger does not include coding or compression, which would significantly increase FPGA resources.

During debugging the development environment connects to the debugger modules on the FPGA using the integrated RS232 interface. The XML file provides a mapping between the hardware implementation and the model and allows the user to control the debugging according to the crosswalk model. Further, the XML file is used to visualize the internal execution of the system in the model, highlighting active states and displaying the value of inputs, outputs and internal variables. Besides the real-time recording and postmortem analysis the system could be directly monitored during runtime. The transmission, interpretation and visualization of the status of 32 signals in the design every 100ms are performed without any timing problems. However, this is only a data rate of 320bit/s, therefore there is no real-time monitoring possible. In another design - with 128 signals - the transmitting interval even needed to be reduced to 250ms to process the data before the next is transmitted, the most of that time thereby is consumed by the graphical visualization.

8 Conclusion and Outlook

A method for model-based real-time recording and monitoring on reconfigurable hardware has been presented. Therefore, the possibilities of an abstract and complex functional and algorithmic inspection of reconfigurable system have been increased. In comparison with present techniques, the user does not only develop on model level, but can also debug and monitor as well as control the debugging on the same level. All intermediate steps from the model to hardware implementation and vice versa are carried out automatically by algorithms.

The underlying hardware modules are capable of monitoring and real-time recording as well as independent of the reviewed model. The debugger provides high modularity and adjustability during runtime, allowing several ways to identify causes of errors without re-synthesizing the design. The integrated development environment allows automatic integration of the debugger using a generated interface, control of the debugging during runtime and visualization of the execution - all on model level. The algorithms in the background convert the received data from hardware to model level and convert the commands from model level to hardware.

In future, there will still be many developed modules on code level, therefore the concept could be extended to allow mixed debugging on model, code and signal level. According to the software, the integration of breakpoints on model level could be added allowing more specific debugging scenarios. Also the configuration of the debugger in terms of the compression and complexity of the trigger condition, which is at the moment performed in the VHDL code, could be integrated into software. In addition, the IDE will be extended to allow complete code generation of statechart diagrams to get independent of Matlab.

References

1. Object Management Group: Unified Modeling Language (UML) Specification, Version 2.2 (2008)
2. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8 (1987, 3), pp. 231–274 (1987)
3. Barry, J. R., Lee, E. A., Messerschmitt, D. G., Lee, E. A.: Digital communication. Springer, New York (2004)
4. Schmidt, D.C.: Model-Driven Engineering. J. Computer. Vol. 39 Iss. 2, pp. 25–31 (2006)
5. Schwalb, T., Graf, P., Müller-Glaser, K.D.: Architektur für das echtzeitfähige Debugging ausführbarer Modelle auf rekonfigurierbarer Hardware. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, pp. 127–137. Berlin (2009)
6. Howe H.: Pre- and postsynthesis simulation mismatches. In: Verilog HDL Conference, pp. 24-31. IEEE International, Santa Clare (1997)
7. Lach, J., Mangione-Smith, W., Potkonjak, M.: Efficient error detection, localization, and correction for fpga-based debugging. In: 37th Design Automation Conference, pp. 207-212. Los Angeles (2000)
8. McKay, N., Singh, S.: Debugging techniques for dynamically reconfigurable hardware. In: Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp. 144–122. Napa Valley (1999)
9. Arshak, K., Jafer, E., Ibala C.: Testing fpga based digital system using xilinx chipscope logic analyzer. In: 29th International Spring Seminar on Electronics Technology, pp. 355-360. ISSE06, St. Marienthal (2006)
10. Mathworks: Matlab & Simulink (2010) `http://www.mathworks.de/`
11. Graf, P., Hübner, M., Müller-Glaser, K.D., Becker, J.: A Graphical Model-Level Debugger for Heterogenous Reconfigurable Architectures. In: 17th International Conference on Field Programmable Logic and Applications (FPL), pp. 722–725. Amsterdam (2007)
12. Graf, P., Reichmann, C., Müller-Glaser, K.D.: Towards a Platform for Debugging Executed UML-Models in Embedded Systems. In: UML Modelling Languages and applications, pp. 238–241. Springer, Heidelberg (2004)
13. Graf, P., Müller-Glaser, K.D.: ModelScope Inspecting Executable Models during Run-time. In: 30th International Conference on Software Engineering, pp. 935–936. Leipzig (2008)
14. Eclipse Foundation: Eclipse Modeling Project (2010) `http://www.eclipse.org/modeling`
15. Rosenberg, J.B.: How Debuggers Work. John Wiley & Sons Inc., New York (1996)
16. Xlilinx: Virtex-II Pro and Virtex-II Pro X - FPGA User Guide v.4.2 (2007)

Knowledge-based Runtime Failure Detection for Industrial Automation Systems

Martin Melik-Merkumians, Thomas Moser, Alexander Schatten, Alois Zoitl and Stefan Biffl

Christian Doppler Laboratory for Software Engineering Integration
for Flexible Automation Systems
Vienna University of Technology, Austria
{firstname.lastname}@tuwien.ac.at

Abstract. Engineers of complex industrial automation systems need engineering knowledge from both design-time and runtime engineering models to make the system more robust against normally hard to identify runtime failures. Design models usually do not exist in a machine-understandable format suitable for automated failure detection at runtime. Thus domain and software experts are needed to integrate the fragmented views from these models. In this paper we propose an ontology-based engineering knowledge base to provide relevant design-time and runtime engineering knowledge in machine-understandable form to be able to better identify and respond to failures. We illustrate and evaluate the approach with models and data from a real-world case study in the area of industrial automation systems. Major result was that the integrated design-time and runtime engineering knowledge enables the effective detection of runtime failures that are only detectable by combining runtime and design-time information.

1 Introduction

Complex industrial automation systems need to be flexible to adapt to changing business situations and to become more robust against relevant classes of failures. Production automation systems consist of components, for which a general design and behavior is defined during the design phase, but much of the specific design and behavior is defined during implementation, deployment, and runtime with a range of configuration options. The educational process plant is used to simulate complex industrial batch processes (like refineries, breweries, or pharmaceutical plants). It consists of two tanks holding the process fluids. The liquid level of the tanks are checked by several analog and digital sensors. The lower tanks also contains a pump which either transports the process fluid into the upper tank, or is used to mix up the process fluid in the lower tank. Also the lower tank contains a heater and a temperature sensor to heat the process fluid to specified temperatures. Figure 1 shows on the left hand side an image of the real system, while the right hand side of the figure displays a simplified version of the underlying data model of the educational process plant. Additionally, some

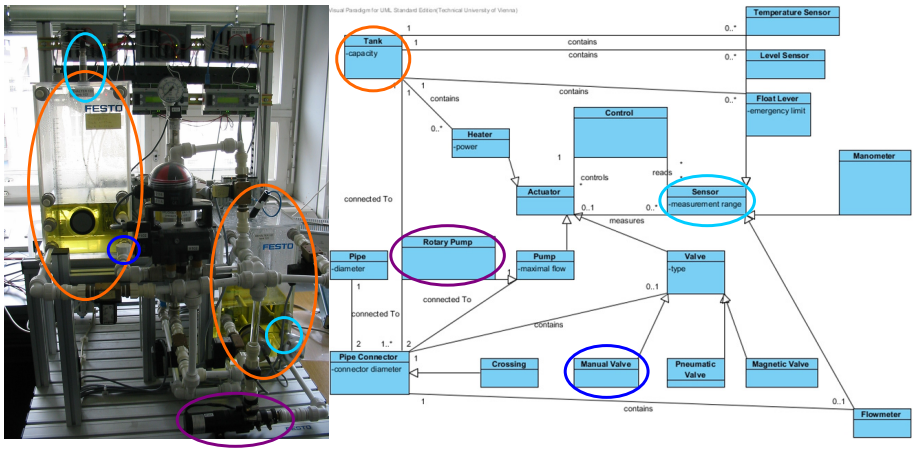


Fig. 1. Educational Process Plant - real system and underlying data model

of the data models elements have been colored in the same color as their real-world representation in the image of the educational process plant in order to show the links between real-world system and underlying data model.

Engineers, who want to detect failures at runtime which can not be compassed by analyzing singular sensor values or failures at sensor-less components (e.g. broken actuators which are not monitored by a sensor), need information from software models that reflect dependencies between components at design and runtime, e.g., the workshop layout, recipes and production procedures. During development design-time software models (representing electrical, mechanical, and software engineering models), like data-oriented models (e.g., EER models or P&ID[1] (Piping and Instrumentation) diagrams [8]) or work flow-oriented models (e.g., sequence diagrams or state charts) are the basis to derive runtime models but are often not provided in machine-understandable format to reflect on failures at runtime, i.e., the knowledge is kept in an explicit human-understandable way but cannot be accessed by components automatically. Domain and software experts are needed to integrate the fragmented views (e.g., propagating model changes into other models, cross-model consistency checks) from these models, which often is an expensive and error-prone task due to undetected model inconsistencies or lost experience from personnel turnover.

Practitioners, especially designers and quality assurance (QA) personnel, want to make complex industrial automation systems (which like the educational process plant consist of components defined by general design-time behavior, derived runtime configuration, and runtime specific behavior enactment) more robust against normally hard to identify runtime failures. QA people could

[1] Industrial standard for P&IDs: IEC 61346: Industrial systems, Installations and Equipment and Industrial Products Structuring Principles and Reference Designations

benefit from more effective and efficient tool support to check system correctness, by improving the visibility of the system defect symptoms (e.g., exceptions raised from assertions).

Challenges to detect and locate defects at runtime come from the different focus points of models: e.g., components and their behavior are defined at design time, while configurations may change at runtime and violate tacit engineering assumptions in the design-time models. Without an integrated view on relevant parts of both design-time and runtime models inconsistencies from changes and their impact are harder to evaluate and resolve between design and runtime. Better integrated engineering knowledge can improve the quality of decisions for runtime changes to the system, e.g., better handling severe failures with predictable recovery procedures, lower level of avoidable downtime, and better visibility of risks before damage occurs. As shown in [11], with the help of ontologies and reasoning most of these problems can be addressed.

In this paper we present an approach to improve support for runtime decision making with an ontology: a domain-specific engineering knowledge base (EKB) that provides a better integrated view on relevant engineering knowledge in typical design-time and runtime models, which were originally not designed for machine-understandable integration. The EKB can contain schemes on all levels and instances, data, and allows reasoning to evaluate rules that involve information from several models that would be fragmented without machine-understandable integration. The major advantage of using an ontology for representing and querying the domain-specific engineering knowledge is the fact that ontologies are well suited to model logical relationships between different variables in axioms which can be used later for the derivation of assertions based on measured runtime data. We illustrate and evaluate the ontology-based approach with two types of runtime failure (RTFs) from a real-world use case study in the area of industrial automation systems. Major result was that the integrated design-time and runtime engineering knowledge enables the effective detection of normally hard to identify runtime failures.

In the remainder of the paper we survey relevant engineering models for their contributions and limitations to support runtime decision making; we describe a real-world case on runtime failure detection for collecting evidence to which extent richer and better integrated semantic knowledge can translate into better decision making.

2 Evolution of Engineering Models towards Runtime System Analysis and Adaptation

Engineering models have evolved from means to structure complex domains and designs at design time towards model-driven approaches that bring domain information closer to implementation and runtime. However, systems that are designed for adaptation at runtime need more advanced approaches to provide relevant and accurate engineering knowledge to guide runtime system analysis and adaptation.

Structuring design complexity. Models are used on various levels in software engineering: Data models like entity relationship (ER) diagrams originate in the late 1970s [5]. With the Unified Modeling Language (UML) [3] a standardized set of diagrams and modeling techniques were introduced for object-oriented design. However, these models are mostly used initially during the design time of a project and get seldom adapted to changes during implementation or operation.

Connecting design and implementation. There are tool providers, who claim to support round trip engineering from design models to source code and back. However, a stronger and more consistent integration between the design models and the implementation phase artifacts comes from the Model-Driven Architecture (MDA), Model-Driven Development (MDD) [16], and Model-Driven Configuration management. There are several interesting aspects about MDD: The models develop from high-level abstractions to concrete code over several intermediate steps. The initial model can be a general UML model or a domain-specific language model. In MDD (opposed to earlier modeling approaches) the model is used also in the implementation phase, i.e., used to create platform-specific code, but is not used at runtime. There also exist approaches for using MDA for the engineering of automation systems. Melik-Merkumians et al. [10] present an approach that separates between logical control applications and the plant model. The logical control application models the intended behavior of the control application in a target independent way. The plant model defines the control devices, their abilities, and their interconnections (e.g., communication system). By mapping both models together the control code executed in the control devices with their hardware specific parameters can be generated automatically.

Connections between design, implementation, and runtime. Traditional software engineering approaches mostly focus on the development phase and see configuration management (CM) as a support task. However, CM is an example model that is valuable at design time, implementation, deployment, and runtime. Some approaches thus suggest including CM and application life cycle management (ALM) into the MDD concept [6].

Runtime needs of distributed reconfigurable software-intensive systems. Ahluwalia et al. [1] observe a shift from "monolithic to highly networked, heterogeneous, interactive systems" that has led to a "dramatic increase in both development and system complexity", where at the same time the "demands for safety, reliability, and other qualitative attributes have increased across application domains." Oreizy et al. [14] additionally mention the necessity of "runtime evolution" of modern multi-user, distributed systems. Today many deployed applications gradually evolve over time (ideally without downtime for users) rather than undergo "big bang" version updates. Examples for such applications are e-Commerce Services like Online-Banking applications as well as most "Web 2.0" applications. The problem gets particularly critical in domains like distributed real-time and embedded systems as in automotive and production automation industry applications.

Feedback of runtime experience to design. An underlying trend is to bring development activities closer to the runtime environments, i.e., using data from the deployed system for engineering purposes (e.g., QoS parameters). Additionally, we observe more intensive research activities [17] to design and apply MDD where the models do not stop at development but also support the runtime environment of the system. Recent research investigates mechanisms towards automatic runtime failure detection [17] and ultimately self-healing systems. Garlan et al. [6] describe model-based approaches for self-healing autonomous systems with a similar idea: remove the traditional separation between system creation/-modification and runtime environment with an integrated approach. The authors particularly point out that system-internal exception handling and configuration (hence not always easy to change) is problematic in many modern systems as they are designed to run continuously, and also updates and reconfiguration should be done without system shutdown.

Ontologies for connecting design-time and runtime data models. An ontology is a representation vocabulary for a specific domain or subject matter, e.g., production automation. More precisely, it is not the vocabulary as such that qualifies as an ontology, but the (domain-specific) concepts that the terms in the vocabulary are intended to capture [4]. The infrastructure of MDA provides architecture for creating models and meta-models, defining transformations between these models, and managing meta-data. Although the semantics of a model is structurally defined by its meta-model, the mechanisms to describe the semantics of the domain are rather limited compared to machine-understandable representations using, e.g., knowledge representation languages like RDF[2] or OWL[3]. In addition, MDA-based languages do not have a knowledge-based foundation to enable reasoning (e.g., for supporting QA), which ontologies provide [2]. Beyond traditional data models like UML class diagrams or entity relationship diagrams, ontologies provide methods for integrating fragmented data models into a common model without losing the notation and style of the individual models [7]. The usage of ontologies for knowledge representation and sharing, as well as for and high-level reasoning could be seen as a major step towards the area of agent-based control solutions [13]. Exploitation of semantics and ontologies in the area of agent-based industrial systems has become one of the major research areas in the last few years, primarily because of the success and promotion of semantic web technologies to enable better communication between machines and people [15]. Ontologies are considered here as an essential technology for semantic web development guaranteeing data and information interoperability in heterogeneous and content-rich environments [12].

3 An Integrating Engineering Knowledge Base

In this section, we introduce the Engineering Knowledge Base (EKB), a set of relevant information elements about components in machine-understandable

[2] Resource Description Framework: http://www.w3.org/RDF
[3] Web Ontology Language: http://www.w3.org/2007/OWL

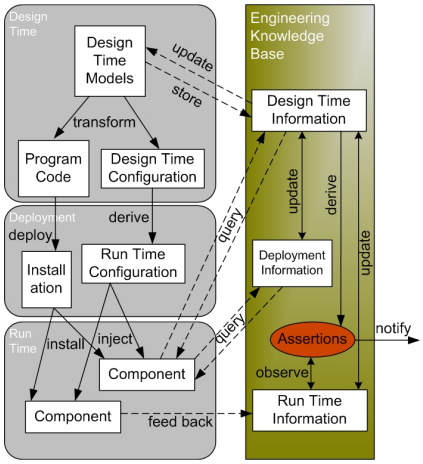


Fig. 2. An Engineering Knowledge Base in Context

format using ontology syntax. Components can query the EKB at runtime to retrieve information for detecting normally hard to identify failures or implausibilities of the current system.

Figure 2 illustrates 3 major phases in the life cycle of complex industrial automation systems:

1. Design time: Models that describe the automation system layouts, the recipes of the manufactured products, etc. are transformed into executable program code and design-time configuration instructions.
2. In the Deployment phase, the executable program code is deployed into installable packages and the runtime configuration is derived from the design-time configuration.
3. At runtime the deployed program code for system operation gets installed to a set of components and the runtime configuration gets injected into these components.

This architecture has proven effective for systems whose properties change seldom, since the effort needed for transformation, deployment, and injection is considerable.

However, typical complex industrial automation systems also suffer from failures, e.g., if some components fail or become unavailable. To support failure detection, the components need to be able to perform decisions at runtime, since a complete new iteration of model transformation, program code deployment, and configuration injection would take too long. A major challenge of runtime failure detection is to provide access to relevant design-time information that is usually stripped away during transformation for efficiency reasons. The Engineering Knowledge Base (EKB) provides a place for storing design-time information that seems valuable for supporting runtime failure detection of components, especially

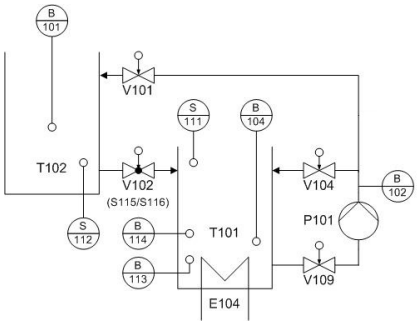


Fig. 3. P&ID of the Educational Process Plant

in the case of handling failures or unplanned situations (but not transformed into runtime code or configuration to limit their complexity).

Components can query the EKB at runtime with semantic web query languages like SPARQL[4] (SPARQL Protocol and RDF Query Language) or SWRL[5] (Semantic Web Rule Language), which provide to the components the full expressive power of ontologies, including the ability to derive new facts by reasoning. In addition, components can feed back interesting observations into the runtime information collection of the EKB and therefore help to improve the design-time models (e.g., by improving estimated process properties with analysis of actual runtime data) and/or check the information based on a certain set of assertions. Furthermore, valuable deployment information can also be stored in the EKB in order to support and enhance for further deployments.

Based on the design-time information, it is possible to define a set of runtime assertions in the EKB. These runtime assertions observe the runtime information fed back into the EKB and can notify a specific role or system if the violation of an assertion has been detected.

4 Real-World Use Case and Results

In this section we describe two real-world failure use cases which could lead to equipment defects and/or waste of used production material. Especially failures which are hard to identify by traditional means can lead to subsequent equipment damage and decrease of process performance. Therefore the identification of such failures is of great importance. We will show that such failures can be identified by usage of the EKB. We will also see that the EKB can provide soft sensors by simple (and therefore easy to implement in PLC (programmable logic controller) programs) reasoning of process data. The runtime assertions will be given in pseudo code samples. The assertions can be evaluated periodically, as it would be in a traditional PLC environment, or in regard to certain events, like the

[4] www.w3.org/TR/rdf-sparql-query
[5] www.w3.org/Submission/SWRL

change of a sensor value. As long as all needed sensor inputs and PLC outputs are available and stable, the current system status is correctly mapped.

The model we use to demonstrate our failure use cases is an educational process plant in the Odo-Struger-Laboratory[6]. Figure 3 shows the P&ID (Piping and Instrumentation Diagram) of the tank model. It consists of two tanks, T101 and T102, which are on different height levels. The upper tank (T102) contains an ultrasonic level sensor (B101) which measures the distance of the liquid surface to the upper tank closure, and a float lever which represents a critical low level of the liquid. The lower tank (T101) consists of three float levers (a critical upper level (B114), a lower level (B104), and a critical lower level lever (B113)), a heater (E104), and a temperature sensor (S111). The two tanks are connected by several pipes which are opened or closed by several valves (the symbols marked with V). Valves are usually not equipped with sensors to determine their current state, due to financial reasons. The liquid transportation from the lower to the upper tank is done by the pump P101, The actual flow caused by the pump is measured by the flow meter (B102). Even as this process plant is an educational model it represents a typical plant configuration in the process industry. Listing 1 shows some of the EKB's triples defining the tank T101 with blabla. For a complete listing of the EKB's triples representing the P&ID model elements please refer to [9].

Listing 1. Example of EKB triples representing P&ID model elements

```
<Tank rdf:ID="T102">
  <contains>
    <Float_Lever rdf:resource="#LS-102"/>
  </contains>
  <contains>
    <Level_Sensor rdf:resource="#LIC-102"/>
  </contains>
  <connected_To>
    <Pipe_Connector rdf:resource="#PC-B102-1"/>
  </connected_To>
  <connected_To>
    <Pipe_Connector rdf:resource="#PC-B102-2"/>
  </connected_To>
  <capacity rdf:datatype="xml:#float">1000.0</capacity>
</Tank>
```

RTF-1: Undetected valve V101 failure in the pump pipe section. In this scenario the liquid in tank T101 shall be pumped into tank T102. Therefore valve V104 must be closed and the valves V109 and V101 must be opened. We now assume that one or both valves that should be open are defect and therefore unable to get into the open position. As the control has no means to control the position of the valves the pump P101 starts. Such a situation can lead to the destruction of the pump, as it is either pumping against the impregnable

[6] The industrial automation systems laboratory of the Automation and Control Institute, Vienna University of Technology

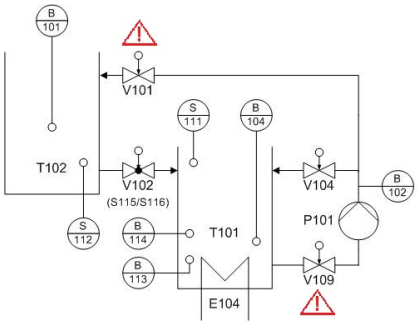


Fig. 4. P&ID showing RTF-1: Undetected valve V101 failure

resistance of valve V101, or it will be soon out of water as valve V109 inhibits further water supply, which leads to destruction by for example rotary pumps.

To avoid such equipment failure it is common to use a flow meter (B102 in Figure 4), which can be used compare the actual flow with the anticipated flow value. If those two values differ too much the pump can be put into an emergency shut down. But if the flow meter is defect or the flow meter is omitted for cost reduction, the failure can not be detected anymore. Such failures can lead to huge costs due to equipment loss, material waste, and additional down times and should therefore be avoided. Engineering knowledge can be used to provide a simple rule to check the functionality of the pump system. As the upper tank T102 has a level sensor it can be checked if the liquid level of the upper tank is rising if the pump is activated. A more correct model would additionally be able to determine if the level rising is proportional to the actual pump power. Through consequent usage of engineering knowledge such "soft sensors" (instead of a real sensor the value has been calculated from the process model) can be created. By comparison of the measured and calculated flow value the system can also determine if the flow meter or the pump is behaving as expected. Listing 2 shows this query in pseudo code. For a complete listing of the query in full SPARQL syntax please refer to [9].

Listing 2. Pseudo code query for detecting failures of V101

```
while (P101.isActive())
  for (B102.getSensorEvent() as x)
    for (B102.getSensorEvent() as y)
     if (x.getTimestamp() < y.getTimestamp())
       if ( NOT (y.getLevel() > x.getLevel()) )
         << raise alarm >>
```

RTF-2: Leakage in valve V102 with subsequent material loss. Such a failure can lead to enormous costs due to material waste, lower quality end product (perhaps even to low to sell), and wasted production time. Especially the wasted production time is costly as process engineering processes (like refining and chemical processes in general) usually need long times to be completed. Such

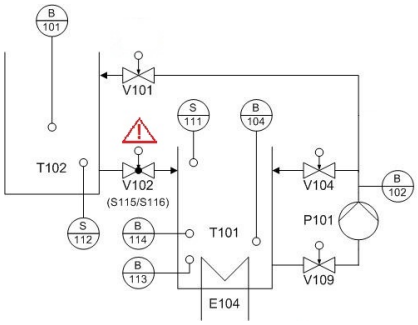


Fig. 5. P&ID showing RTF-2: Leakage in valve V102

failures are hard to detect by conventional alarming mechanisms, as traditional alarms are only issued if a process value reaches a critical threshold.

In this scenario we assume that valve V102 is leaky (Figure 5), which is why the process material in tank T102 is slowly petering out. In an ordinary control program would only wait for a longer time until the liquid threshold defined in the program is attained. There is no way to detect the leakage by simple sensor or alarm processing, only well grounded personnel in the control room can detect such a failure in a traditional PLC system.

Listing 3. Pseudo code query for detecting leakage in valve V102

```
while (NOT (P101.isActive()) )
  if (V102.isClosed())
    for (B102.getSensorEvent() as x)
      for (B102.getSensorEvent() as y)
        if (x.getTimestamp() < y.getTimestamp())
          if ( NOT (y.getLevel() == x.getLevel() )
            << raise alarm >>

while (P101.isActive())
  if (V102.isClosed() AND V104.isClosed())
    if (V109.isOpen() AND V101.isOpen())
      for (B102.getSensorEvent() as x)
        for (B102.getSensorEvent() as y)
          if (x.getTimestamp() < y.getTimestamp())
            if ( NOT (y.getLevel() =
              x.getLevel() * P101.getPumpPower()) )
                << raise alarm >>
```

Once again the usage of engineering knowledge leads to rules capable to detect this failure. The first rule is, if the pump P101 is not pumping and valve V102 is closed, then the fluid level of tank T102 has to be constant. If there is a leakage in the valve then the level would sink. Assuming that the flow meter B102 is working correct the second rule is, if the valves V102 and V104 are closed,

and the valves V109 and V101 are open, and the pump P101 is pumping, then the fluid level of tank T102 must rise proportional to the actual pump power.

The first rule is capable of detecting a leakage in tank T102, but the second rule can only state that some condition is violated. The solution to this problem is rather easy and efficient. If the second rule fires than stop pump P101 and reevaluate rule one for leakage detection. Another benefit of this behavior is that the material loss is limited to the content of tank T102 or stopped altogether. Listing 3 shows this query in pseudo code. For a complete listing of the query in full SPARQL syntax please refer to [9].

5 Summary and Further Work

In this paper we described an ontology-based approach to provide relevant design-time and runtime engineering knowledge stored in a so called Engineering Knowledge Base (EKB). The EKB provides a better integrated view on relevant engineering knowledge contained in typical design-time and runtime models in machine-understandable form to support runtime failure detection. This approach is useful in the industrial automation domain, and can more generally be used for other (distributed) engineering systems. We illustrated our approach with two use cases of runtime failure detection from a real-world case study in the area complex industrial automation systems. The use cases identified the needs for a complex decision system for failures that are only detectable by combining sensor information with engineering knowledge and showed a feasible query based approach suitable for the task.

Major result of the evaluation of the proposed EKB approach was the possibility to define assertions in the EKB which are checked based on the runtime information input of the running components. This can be seen as external Quality Assurance (QA) without interfering with the original production system and therefore it has proven to be easier to enrich existing applications without the need to make changes to legacy systems (smoother migration path). Further, the quality of information presented to an operator is improved since all information both from design-time as well as from runtime is available, leading to more intelligent runtime analysis and decision support.

Further important practical issues are to investigate the effort needed to import the data from the relevant models into the Engineering Knowledge Base and improving the performance of data access and reasoning at runtime. The use of assertions for checking QoS parameters like system throughput is open to further research too.

Acknowledgment

This work has been supported by the Christian Doppler Forschungsgesellschaft and the BMWFJ, Austria. This work has been partially funded by the Vienna University of Technology, in the Complex Systems Design and Engineering Lab.

References

1. Ahluwalia, J., Krger, I.H., Phillips, W., Meisinger, M.: Model-based run-time monitoring of end-to-end dead-lines. In: 5th ACM international conference on Embedded software (EMSOFT '05). pp. 100–109. ACM (2005)
2. Baclawski, K., Kokar, M.K., Kogut, P.A., Hart, L., Smith, J., Letkowski, J., Emery, P.: Extending the unified modeling language for ontology development. International Journal of Software and Systems Modeling (SoSyM) 1(2), 142–156 (2002)
3. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language Reference Manual. Addison-Wesley (1999)
4. Chandrasekaran, B., Josephson, J.R., Benjamins, V.R.: What are ontologies, and why do we need them? IEEE Intelligent Systems and Their Applications 14(1), 20–26 (1999)
5. Chen, P.P.S.: The entity-relationship model - toward a unified view of data. ACM Transactions on Database Systems (TODS) 1(1), 9–36 (1976)
6. Garlan, F., Lopez de Vergara, J., Fernandez, D., Munoz, R.: A model-driven configuration management methodology for testbed infrastructures. In: IEEE Network Operations and Management Symposium NOMS 2008. pp. 747–750. IEEE (2008)
7. Hepp, M., De Leenheer, P., De Moor, A., Sure, Y.: Ontology Management: Semantic Web, Semantic Web Services, and Business Applications. Springer-Verlag (2007)
8. Love, J.: Process Automation Handbook: A Guide to Theory and Practice. Springer-Verlag London (2007)
9. Melik-Merkumians, M., Moser, T., Schatten, A., Zoitl, A., Biffl, S.: Knowledge-based runtime failure detection for industrial automation systems. Tech. rep., Christian Doppler Laboratory for Software Engineering Integration for Flexible Automation Systems Vienna University of Technology, Austria (2010), http://cdl.ifs.tuwien.ac.at/files/MRT2010_tr.pdf
10. Melik-Merkumians, M., Wenger, M., Hametner, R., Zoitl, A.: Increasing portability and reuseability of distributed control programs by i/o access abstraction. In: 15th IEEE International Conference on Emerging Techonologies and Factory Automation - Work in Progress Track. p. accepted for publication (2010)
11. Melik-Merkumians, M., Zoitl, A., Moser, T.: Ontology-based fault diagnosis for industrial control applications. In: 15th IEEE International Conference on Emerging Techonologies and Factory Automation - Work in Progress Track. p. accepted for publication (2010)
12. Merdan, M.: Knowledge-based Multi-Agent Architecture Applied in the Assembly Domain. Ph.D. thesis, Vienna University of Technology (2009)
13. Obitko, M., Marík, V.: Ontologies for Multi-Agent Systems in Manufacturing Domain. In: DEXA '02: Proceedings of the 13th International Workshop on Database and Expert Systems Applications. pp. 597–602. IEEE Computer Society, Washington, DC, USA (2002)
14. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: International Conference on Software Engineerings (ICSE 2008). pp. 177–187. IEEE Computer Society (2008)
15. Tim Berners-Lee, J.H., Lassita, O.: The semantic web. Scientific American 284, 34–43 (2001)
16. Vlter, M., Stahl, T.: Model-driven Software Development. John Wiley (2006)
17. Wuttke, J.: Runtime failure detection. In: Companion of the 30th International Conference on Software Engineering. pp. 987–990. ACM, Leipzig, Germany (2008), 1370219 987-990