4th Workshop on Models@run.time at MODELS 09

Denver, USA, October 5th 2009

Proceedings

Editors

Nelly Bencomo
Gordon Blair
Robert France
Cédric Jeanneret
Freddy Muñoz

Organization Committee

Nelly Bencomo
Lancaster University, UK

Gordon Blair
Lancaster University, UK

Robert France
Colorado State University, USA

Cédric Jeanneret
University of Zurich, Switzerland

Freddy Muñoz
INRIA, France

Program Committee

Danilo Ardagna
Politecnico di Milano, Italy

Peter J. Clarke
Florida International University, USA

Anthony Finkelstein
University College London, UK

John Georgas
Northern Arizona University, USA

Oystein Haugen
SINTEF, Norway

Gang Huang
Peking University, China

Jean-Marc Jezequel
IRISA, France

Flavio Oquendo
University of Brittany, France

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

Betty Cheng
Michigan State University, USA

Fabio M. Costa
Federal University of Goias, Brazil

Frank Fleurey
SINTEF, Norway

Jeff Gray
University of Alabama at Birmingham, USA

Jozef Hooman
Embedded Systems Institute, Netherlands

Paola Inverardi
Università dell'Aquila, Italy

Stéphane Ménoret
THALES Group, France

Arnor Solberg
SINTEF, Norway

Additional Reviewers

Franck Chauvel
Peking University, China

Daniel Schneider
Fraunhofer IESE, Germany

Brice Morin
IRISA, France

Rui Silva Moreira
Universidade Fernando Pessoa, Portugal

Preface

Welcome to the 4th Workshop on Models@run.time at MODELS 2009!

This document contains the proceedings of the 4th Workshop on Models@run.time that will be co-located with the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS). The workshop will take place at the feet of the Rocky Mountains in Denver, USA, on the 5th of October, 2009. The workshop is organized by Nelly Bencomo, Robert France, Gordon Blair, Cédric Jeanneret and Freddy Muñoz.

From a total of 19 papers submitted 4 full papers, 6 posters and 2 demos were accepted. This volume gathers together all the 12 papers accepted at Models@run.time 09. 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 2009

Nelly Bencomo
Gordon Blair
Robert France
Cédric Jeanneret
Freddy Muñoz

Content

Session 1: The Use of Computational Reflection

Session 2: Configuration Management

Poster Session

Incremental Model Synchronization for Efficient Run-time Monitoring

Thomas Vogel, Stefan Neumann, Stephan Hildebrandt,
Holger Giese, and Basil Becker

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

Abstract. The model-driven engineering community has developed expressive model transformation techniques based on meta models, which ease the specification of translations between different model types. Thus, it is attractive to also apply these techniques for autonomic and self-adaptive systems at run-time to enable a comprehensive monitoring of their architectures while reducing development efforts. This requires special solutions for model transformation techniques as they are applied at run-time instead of their traditional usage at development time. In this paper we present an approach to ease the development of architectural monitoring based on the incremental model synchronization with triple graph grammars. We show that the provided incremental synchronization between a running system and models for different self-management capabilities provides a significantly better compromise between performance and development costs than manually developed solutions.

1 Introduction

The complexity of today's software systems impedes the administration of these systems by humans. The vision of *self-adaptive software* [1] and *Autonomic Computing* [2] addresses this problem by considering systems that manage themselves given high-level goals from humans. The typical self-management capabilities *self-configuration*, *self-healing*, *self-optimization* or *self-protection* [2] can greatly benefit when besides some parameters, e.g. for configuration purposes, also the architecture of a managed software system can be observed [3].

Each of these capabilities requires its own abstract view on a managed software system that reflects the run-time state of the system regarding its architecture and parameters in the context of the concern being addressed by the corresponding capability, e.g. performance in the case of self-optimization. Monitoring an architecture of a running system in addition to its parameters requires an efficient solution to be applicable at run-time and it results in a considerable increase in complexity. The complexity further increases, as a view has to be usually decoupled from a running system for system analysis. Otherwise, changes that occurred during an analysis might invalidate the analysis results, as the analysis was not performed on a consistent view. Due to the complexity, the development of monitoring activities should be eased or even automated. Moreover, different views on a running system have to be provided efficiently at run-time.

In this context, *Model-Driven Engineering* (MDE) techniques can in principle help. MDE provides expressive model transformation techniques based on meta models which ease the specification of translations between different model types.

Basically and as argued in [4], these techniques could be used at run-time for run-time models and thus also ease the development of architectural monitoring.

In this paper we propose a model-driven approach that enables a comprehensive monitoring of a running system by using meta models and model transformation techniques as sketched in [5], where there was no room for a detailed discussion of the approach. Different views on a system regarding different self-management capabilities are provided through run-time models that are derived and maintained by our model transformation engine automatically. The engine employs our optimized model transformation technique [6, 7] that permits incremental processing and therefore can operate efficiently and online. Furthermore, the approach eases the development efforts for monitoring. For evaluation, the implementation of our approach considers performance monitoring, checking architectural constraints and failure monitoring that are relevant for self-optimization, self-configuration, and self-healing capabilities, respectively.

The paper is structured as follows: The proposed approach is presented in Section 2 and its application in Section 3. The benefits of the approach are discussed with respect to development costs and performance in Section 4. The paper closes with a discussion of related work and a conclusion.

2 Approach

To monitor the architecture and parameters of a running software system, our approach employs *Model-Driven Engineering* (MDE) techniques. MDE techniques are employed to handle the monitoring and analysis of a system at the higher level of models rather than at the API level. Therefore, using MDE techniques, different models describing certain aspects of or certain views on a running system required for different self-management capabilities can be derived and maintained at run-time. Thus, models of a managed system and of its architecture essentially build the interface for monitoring a system. The generic architecture of our monitoring approach is derived from [5] and depicted in Figure 1.

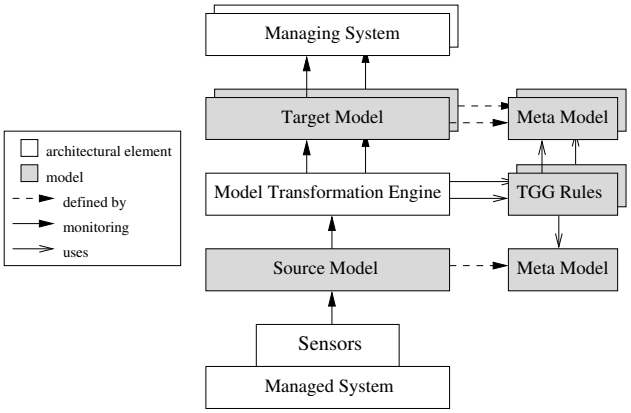


Fig. 1. Generic Architecture (cf. [5])

A *Managed System* provides *Sensors* that are used to observe the system, but that are usually at the abstraction level of APIs. These sensors can be used by any kind of *Managing Systems* for monitoring activities. Managing systems

can be, e.g., administration tools used by humans or even autonomic managers in case of a control loop architecture as proposed, among others, by Kephart and Chess [2]. Since it is difficult to use sensors at such a low level of abstraction, our approach provides a run-time model of a managed system in the form of a *Source Model* to enable a model-based access to sensors. This model is maintained at run-time and updated if changes occur in the managed system.

Nevertheless, a source model represents all capabilities of the sensors. Consequently, it might be quite complex, which makes it laborious to use it as a basis for monitoring and analysis activities by managing systems. As the source model is defined by a *Meta Model*, it can be accessed by model transformation techniques. Using such techniques, we propose to derive several *Target Models* from the source model at run-time. Each target model raises the level of abstraction w.r.t. the source model and it provides a specific view on a managed system required for a certain self-management capability. A target model might represent, e.g., the security conditions or the resource utilization and performance state of a managed system to address *self-protection* or *self-optimization*, respectively. Thus, a managing system being concerned, e.g., with *self-optimization* will use only those target models that are relevant for optimizing a managed system, but does not have to consider aspects or views that are covered by other capabilities such as *self-protection*. Though providing different views on a system, several target models may represent overlapping aspects. Consequently, several managing systems work concurrently on possibly different target models (cf. Figure 1).

The different target models are maintained by our *Model Transformation Engine*, which is based on *Triple Graph Grammars* (TGGs) [6, 7]. *TGG Rules* specify declaratively at the level of meta models how two models, a source and a target model of the corresponding meta models, can be transformed and synchronized with each other. Thus, source and target models have to conform to user defined meta models (cf. Figure 1). A TGG combines three conventional graph grammars: one grammar describes a source model, the second one describes a target model and a third grammar describes a correspondence model. A correspondence model explicitly stores the correspondence relationships between corresponding source and target model elements. Concrete examples of TGG rules are presented in Section 3 together with the application of our approach.

To detect model modifications efficiently, the transformation engine relies on a notification mechanism that reports when a source model element has been changed. To synchronize the changes of a source model to a target model, the engine first checks if the model elements are still consistent by navigating efficiently between both models using the correspondence model. If this is not the case, the engine reestablishes consistency by synchronizing attribute values and adjusting links. If this fails, the inconsistent target model elements are deleted and replaced by new ones that are consistent to the source model. Thus, our model transformation technique synchronizes a source and a target model incrementally and therefore efficiently, which enables its application at run-time. Therefore, for each target meta model, TGG rules have to be defined that specify the synchronization between the source model and the corresponding target

model. Based on declarative TGG rules, operational rules in the form of source code are generated automatically, which actually perform the synchronization.

Thus, our transformation engine reflects changes of the source model in the target models, which supports the monitoring of a managed system. Therefore, relevant information is collected from sensors to enable an analysis of the structure and the behavior of a managed system. As sensors might work in *pull* or *push* oriented manner, updates for a source model are triggered periodically or by events emitted by sensors, respectively. In both cases it is advantageous if the propagation of changes to target models could be restricted to a minimum. Therefore, our model transformation engine only reacts to change notifications dispatched by a source model. The notifications contain all relevant information to identify the changes and to adjust the target models appropriately.

Though the model transformation engine is notified immediately about modifications in the source model, there is no need for the engine to react right away by synchronizing the source model with the target models. The engine has the capability to buffer notifications until synchronization is triggered externally. Hence, the engine is able to synchronize two models that differ in more than one change and it facilitates a decoupling of target models from the source model, which enables the analysis of a consistent view based on target models.

Implementation The implementation is based on the autonomic computing infrastructure *mKernel* [8], which enables the management of software systems being realized with *Enterprise Java Beans 3.0* (EJB) [9] technology for the *Glassfish*[1] application server. For run-time management, *mKernel* provides sensors and effectors as an API. However, this API is not compliant to the *Eclipse Modeling Framework* (EMF)[2], which is the basis for our model transformation techniques. Therefore, we developed an EMF compliant meta model for the EJB domain that captures the capabilities of the API. This meta model defines the source model in our example and a simplified version of it is depicted in Figure 2.

To synchronize a running managed system with our source model, an event-driven *EMF Adapter* has been realized. It modifies the source model incrementally by processing events being emitted by sensors if parameters or the structure of a system have changed. Additionally, the adapter covers on demand the monitoring of frequently occurring behavioral aspects, like concrete interactions, by using pull oriented sensors that avoid the profusion of events.

3 Application

This section describes the application of our model-driven monitoring approach. The meta model for the EJB domain that specifies the source model is depicted in a simplified version in Figure 2. It is divided conceptually into three levels. The top level considers the types of constituting elements of EJB-based systems, which are the results of system development. The middle level covers concrete configurations of EJB-based systems being deployed on a server. Finally, the lower level addresses concrete instances of enterprise beans and interactions

[1] https://glassfish.dev.java.net/
[2] http://www.eclipse.org/modeling/emf/

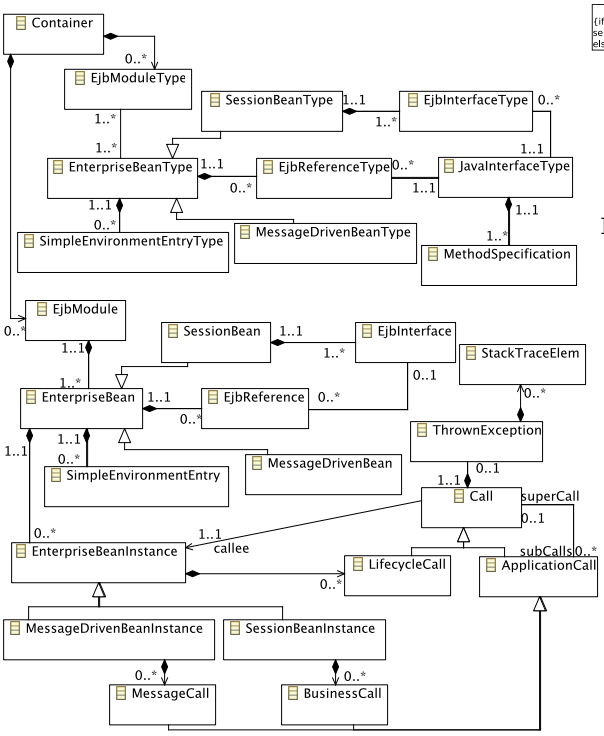


Fig. 2. Simplified Source Meta Model

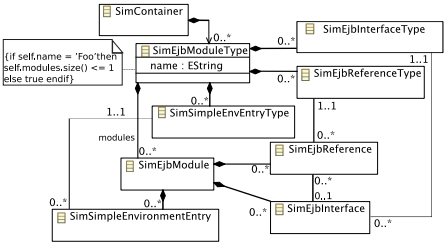


Fig. 3. Simplified Architectural Meta Model

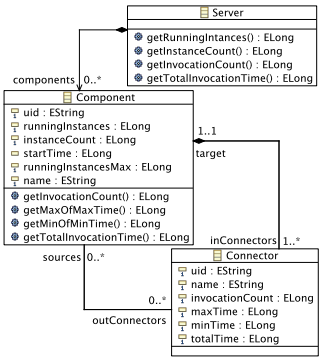


Fig. 4. Performance Meta Model

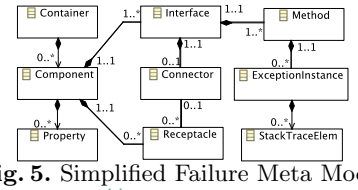


Fig. 5. Simplified Failure Meta Model

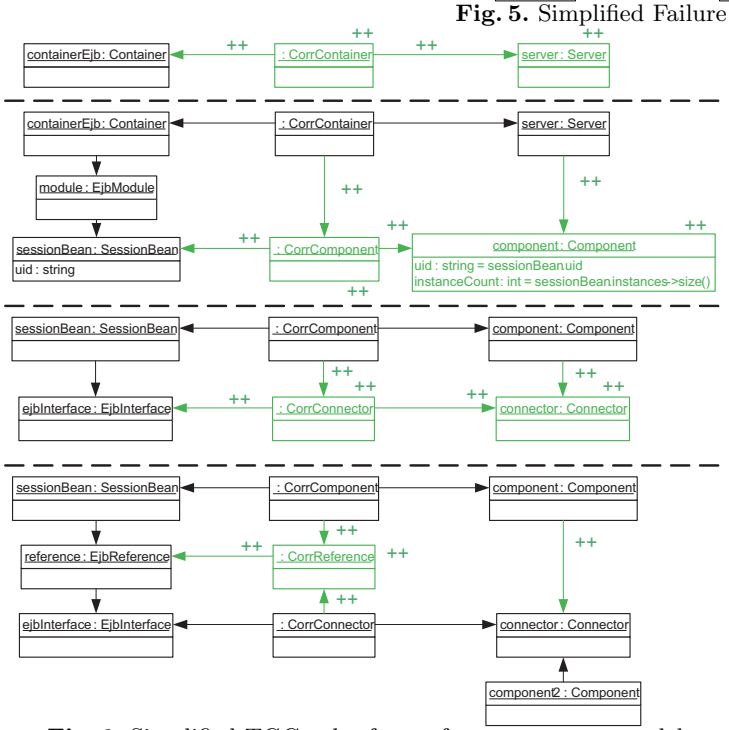


Fig. 6. Simplified TGG rules for performance target model

among them. For brevity, we refer to [8, 9] to get details on the EJB component model and on the three levels. Based on this meta model, a source model provides a comprehensive view on EJB-based systems, which however might be too complex for performing analyses regarding architectural constraints, performance and failure states of managed systems. Therefore, for each of these aspects, we developed a meta model specifying a corresponding target model and the TGG rules defining the synchronization of the source model with the target model. Thus, our model transformation engine synchronizes the source model with three target models aiming at run-time monitoring and analysis of architectural constraints, performance and failure states.

Architectural Constraints Analyzing architectural constraints requires the monitoring of the architecture of a running system. Therefore, we developed a meta model that is depicted in Figure 3 and whose instances reflect simplified run-time architectures of EJB-based systems. It abstracts from the source meta model by providing a black box view on EJB modules through hiding enterprise beans being contained in modules, since modules and not single enterprise beans are the unit of deployment. To analyze architectural constraints, the *Object Constraint Language* (OCL) and checkers like EMF OCL[3] can be used to define and check constraints that are attached to meta model elements, like it is illustrated in Figure 3. The constraint states that at most one instance *SimEjbModule* of a particular *SimEjbModuleType* with a certain value for attribute *name* exists. In other words, at most one module of the module type named *Foo* can be deployed.

Performance Monitoring Like the architectural target meta model, the meta model for target models being used to monitor the performance state of EJB-based systems also abstracts from the source meta model. Figure 4 shows the corresponding meta model. It represents session beans as *Components* and connections among beans as *Connectors* among components. For both entities, information about the instance situation is derived from the source model and stored in their attributes. For each component, e.g., the number of currently running instances or the number of instances that have been created entirely are represented by the attributes *runningInstances* and *instanceCount*, respectively. For each connector, the number of invocations, the maximum and minimum execution time of all invocations and the sum of execution time of all invocations along the connector are reflected by the attributes *invocationCount*, *maxTime*, *minTime* and *totalTime*, respectively. The average execution time of an invocation along a connector can be obtained by dividing *totalTime* with *invocationCount*. Finally, a component provides operations to retrieve aggregated performance data about all connectors provided by the component (*inConnectors*), and a *Server* provides aggregated data about its hosted components.

Based on the structure and attributes of the performance target model, an analysis might detect which components are bottlenecks and which are only blocked by others. Such information might be used, e.g., to decide about relocating busy components to other servers or improving the resource configuration.

The four TGG rules that are required to synchronize the source model with

[3] http://www.eclipse.org/modeling/mdt/downloads/?project=ocl

the performance target model are depicted in a simplified version in Figure 6. For all of them, nodes on the left refer to the source model, nodes on the right to the target model, and nodes in the middle constitute the correspondence model. The elements that are drawn black describe the application context of the rule, i.e., these elements must already exist in the models before the rule can be applied. The elements that are drawn not black and marked with *++* are created by the rule. The first rule in Figure 6 is the axiom that creates the first target model element *Server* for a *Container* in the source model. The correspondence between both is maintained by a *CorrContainer* that is created as well and that is part of the correspondence model. Based on the second rule, for each *SessionBean* of an *EjbModule* associated to a *Container* that is created in the source model, a *Component* is created in the target model and associated to the corresponding *Server*. Likewise to a *CorrContainer*, the *CorrComponent* maintains the mapping between the *SessionBean* and the *Component*. As an example, this rule shows how element attributes are synchronized. The value for the attribute *uid* of a *Component* is derived directly from the attribute *uid* of a *SessionBean*, while *instanceCount* is the number of *SessionBeanInstance* elements the *SessionBean* is connected to via the *instances* link (cf. Figure 2). Moreover, for more complex cases, helper methods operating on the source model can be used to derive values for attributes of target model elements. The third rule is comparable to the second one and it maps an *EjbInterface* provided by a *SessionBean* to a *Connector* for the corresponding *Component*. The last rule creates a link between a *Component* and a *Connector* if an *EjbReference* of the corresponding *SessionBean* is associated to the *EjbInterface* that corresponds to the *Connector*. Comparable rules have been created for all target models, which are not described here for brevity.

Failure Monitoring The last target model is intended for monitoring failures within managed systems. The corresponding meta model is shown in a simplified version in Figure 5. Due to lack of space, we omit a further description of it.

4 Evaluation

In this section our approach is evaluated in comparison with two other feasible solutions that might provide multiple target models for monitoring.

1. **Model-Driven Approach**: The approach presented in this paper.
2. **Non-Incremental Adapter** (NIA): This approach retrieves the current run-time state of a managed system, i.e. a system snapshot, by extracting all structural and behavioral information directly from sensors in a pull oriented manner. Then, the different target models are created from scratch.
3. **Incremental Adapter** (IA): In contrast to the *Non-Incremental Adapter*, this approach uses event-based sensors, which inform a managing system about changes in a managed system in a push oriented manner. These events are processed and reflected incrementally in different target models.

In the following, our approach is evaluated, discussed and compared to these alternative approaches by means of development costs and performance.

Having implemented our approach and the *NIA*, we are able to give concrete values indicating development costs. Using our approach, we had to specify 20

TGG rules to define the transformation and synchronization between the source and all three target models being described in Section 3. On average, each rule has about six to seven nodes, which constitutes quite small diagrams for each rule. However, based on all rules, additional 33371 lines of code including code documentation have been generated automatically. Manually written code in the size of 2685 lines was only required for the *EMF Adapter* (cf. Section 2), that however does not depend on any target meta model and therefore is generic and reusable. Consequently, specifying an acceptable number of TGG rules declaratively is less expensive and error-prone than writing an imperative program that realizes an incremental model synchronization mechanism (cf. about 30k lines of code the *IA* might potentially require). In contrast, the *NIA* required only 902 lines of code, which seems to be of the same complexity like the 20 TGG rules.

Finally, the approaches are discussed w.r.t. their run-time performance characteristics. The results of some measurements[4] are shown in Table 1. The first column *Size* corresponds to the number of beans that are deployed in a server to obtain different sizes for source and target models. Approximately in the same ratio as the number of deployed beans increases, the number of events emitted by *mKernel* sensors due to structural changes, the number of bean instances, and the calls among bean instances increase. *mKernel* sensors allow to monitor structural (S) and behavioral (B) aspects. Behavioral aspects, i.e., concrete calls, can only be monitored in a pull oriented manner, while structural aspects can additionally be obtained through a push oriented event mechanism.

Size	**NIA**		**Model-Driven Approach**						
	S	B	n=0	n=1	n=2	n=3	n=4	n=5	B
5	8037	20967	0	163	361	523	749	891	10733
10	9663	43054	0	152	272	457	585	790	23270
15	10811	72984	0	157	308	472	643	848	36488
20	12257	105671	0	170	325	481	623	820	55491
25	15311	142778	0	178	339	523	708	850	72531

Table 1. Performance measurement [ms]

The *NIA* uses only pull oriented sensors to retrieve all required information to create the three target models separately, from scratch and periodically. For this approach, the second and third column shows the consumed time in milliseconds (ms) to create the three target models. E.g., having deployed ten beans, it took 9663 ms for the structural aspects and 43054 ms for the behavioral aspects.

For our *Model-Driven Approach*, structural aspects are obtained through events and behavioral aspects through pull oriented sensors. The fourth to ninth column show the average time of processing n events, which includes the corresponding adjustments of the source model, and of synchronizing n modifications of the source model to the three target models incrementally by invoking once the model transformation engine. E.g., for $n = 2$ and at a model size of ten, 272 ms are consumed on average for processing two events and for transferring the corresponding changes in the source model to the three target models on average. Additionally, we decomposed the average times to find out the ratio of event processing times and model synchronization times. On average over all model sizes, 7.2%, 5.9%, 4.4%, 4.8% and 3.7% of the average times are used for model syn-

[4] Configuration: Intel Core 2 Duo 3GHz, 3GB RAM, Linux Kernel 2.6.27.11

chronization for the cases of n from one to five, respectively. Consequently, most of the time is spent on event processing, while our model transformation engine performs very efficiently. The third and last column of Table 1 indicate that for both approaches the behavioral monitoring is quite expensive. However, this is a general problem, when complete system behavior should be observed. However, comparing both approaches, our approach clearly outperforms the NIA as it works incrementally. Moreover, a manual *IA* would not be able to outperform our approach, because, as described above, event processing activities are much more expensive than model synchronization activities and a manual *IA* would have three event listeners, one for each target model, in contrast to the one our approach requires. To conclude, our approach outperforms the alternative approaches when development costs and performance are taken into account.

5 Related Work

The need to interpret monitored data in terms of the system's architecture to enable a high-level understanding of the system was recognized by [10], who use only an ADL-based system representation. Model-driven approaches considering run-time models, in contrast to our one, do not work incrementally to maintain those models or they provide only one view on a managed system. In [11] a model is created from scratch out of a system snapshot and it is only used to check constraints expressed in OCL. The run-time model in [12] is updated incrementally. However, it is based on XML descriptors and it provides a view focused on the configuration and deployment of a system, but no other information, e.g., regarding performance. The same holds for [13] whose run-time model is updated incrementally, but reflects also only a structural view. All these approaches [11–13] do not apply advanced MDE techniques like model transformation. In this context, only first ideas exist, like [14], who apply a QVT-based [15] approach to transform models at run-time. They use *MediniQVT* as a partial implementation of QVT, which performs only offline synchronizations, i.e., models have to be read from files, and therefore leads to a performance loss. Moreover, it seems that their source model is not maintained at run-time, but always created on demand from scratch, which would involve non-incremental model transformations.

Regarding the performance of different model transformation techniques, we have shown that our TGG-based transformation engine is competitive to ATL- [16] or QVT-based ones when transforming and synchronizing class and block diagrams [17]. Though the approach presented in this paper uses different models, meta-models and therefore different transformation rules, similar results can be expected for the case study used in this paper.

6 Conclusions & Future Work

This paper presented our approach to support the model-driven monitoring of software systems. It enables the efficient monitoring by using meta models and model transformation techniques. The incremental synchronization between a run-time system and different models can be triggered when needed and therefore multiple managing systems can operate concurrently. The presented solution outperforms feasible alternatives considering development costs and performance.

The core idea of using model transformation techniques for monitoring and even for adaptation of autonomic systems has been presented in [5], where there was no room for a comprehensive discussion. As the results presented in this paper are promising, we are currently investigating the usage of model transformation techniques for architectural adaptations.

References

1. Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., et al.: Software Engineering for Self-Adaptive Systems: A Research Road Map. Number 08031 in Dagstuhl Seminar Proceedings (2008)
2. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer **36**(1) (2003) 41–50
3. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: Proc. of the Workshop on Future of Software Engineering, IEEE (2007) 259–268
4. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: Proc. of the Workshop on Future of Software Engineering, IEEE (2007) 37–54
5. 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, ACM (2009) 67–68
6. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling **8**(1) (March 2009)
7. Giese, H., Hildebrandt, S.: Incremental Model Synchronization for Multiple Updates. In: Proc. of the 3rd Intl. Workshop on Graph and Model Transformation, ACM (2008)
8. Bruhn, J., Niklaus, C., Vogel, T., Wirtz, G.: Comprehensive support for management of Enterprise Applications. In: Proc. of the 6th ACS/IEEE Intl. Conference on Computer Systems and Applications, IEEE (2008) 755–762
9. DeMichiel, L., Keith, M.: JSR 220: Enterprise JavaBeans, Version 3.0: EJB Core Contracts and Requirements. (2006)
10. Garlan, D., Schmerl, B., Chang, J.: Using Gauges for Architecture-Based Monitoring and Adaptation. In: Proc. of the Working Conference on Complex and Dynamic Systems Architecture. (2001)
11. Hein, C., Ritter, T., Wagner, M.: System Monitoring using Constraint Checking as part of Model Based System Management. In: Proc. of 2nd Intl. Workshop on Models@run.time. (2007)
12. 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)
13. Morin, B., Barais, O., Jézéquel, J.M.: K@RT: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines. In: Proc. of the 3rd Intl. Workshop on Models@run.time. (2008) 127–136
14. Song, H., Xiong, Y., Hu, Z., Huang, G., Mei, H.: A model-driven framework for constructing runtime architecture infrastructures. Technical Report GRACE-TR-2008-05, GRACE Center, National Institute of Informatics, Japan (2008)
15. OMG: MOF QVT Final Adopted Specification, OMG Document ptc/05-11-01
16. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming **72**(1-2) (2008) 31–39
17. Giese, H., Hildebrandt, S.: Efficient Model Synchronization of Large-Scale Models. Technical report, No. 28, Hasso Plattner Institute, University of Potsdam (2009)

Generating Synchronization Engines between Running Systems and Their Model-Based Views

Hui Song[1], Yingfei Xiong[1,2], Franck Chauvel[1], Gang Huang[1], Zhenjiang Hu[3], and Hong Mei[1]

[1] Key Laboratory of High Confidence Software Technologies (Ministry of Education)
Peking University, Beijing, China
{songhui06,franck.chauvel,huanggang,meih}@sei.pku.edu.cn
[2] Department of Mathematical Informatics, University of Tokyo, Tokyo, Japan
xiong@ipl.t.u-tokyo.ac.jp
[3] GRACE Center, National Institute of Informatics, Tokyo, Japan
hu@nii.ac.jp

Abstract. The key point to leverage model-based techniques on runtime system management is to ensure the correct synchronization between the running system and its model-based view. In this paper, we present a generative approach, and the supporting tool, to make systematic the development of synchronization engines between running systems and models. We require developers to specify "what kinds of elements to manage" as a MOF meta-model and "how to manipulate those elements using the system's management API" as a so-called access model. From these two specifications, our SM@RT tool automatically generates the synchronization engine to reflect the running system as a MOF-compliant model. We have applied this approach on several practical systems, including the JOnAS JEE server.

1 Introduction

The increasing need of continuously available systems (IT systems, e-business, or critical systems) requires to perform management activities such as configuration, evolution or corrective maintenance *at runtime*.

Management activities (automated or not) are build on a loop [1]: *monitoring* the running system, *analyzing* the collected data, *planning* the needed reconfigurations, and *executing* those reconfigurations. For monitoring and executing, existing platforms such as JEE [2], Fractal[3], and Android [4] provide adequate facilities through devoted APIs, such as the JMX API [5] for JEE systems. For analysis and planning, researchers proposed many generic approaches, utilizing model-based techniques like architecture styles [6, 7], model checking [1], model-based self-repair [8], or model-based artificial intelligence [9], etc.

The key-point to leverage model-based analysis and planning at runtime is to obtain a model-based view of a running system and to ensure the proper synchronization between the system and its model-based view.

However, despite their importance, such synchronization engines are still hand-crafted in a tedious and error-prone manner. Existing approaches [7, 10, 8, 11] include hand-written synchronization engines. To do so, developers have to care about how to maintain a model, how to manipulate the system through the

management API, and how to propagate the changes between them to ensure their consistency. All these functionalities have to be considered simultaneously.

The contribution of this paper is to make systematic the development of such synchronization engines between models and running systems. Our approach reflects a simple model-driven process: For a specific system, we require the developers to specify *what elements can be managed*, and *how to manipulate them* through the management API. From these two specifications, our approach automatically generates a synchronization engine that maintains a dynamic MOF-compliant model for the running system. This model enables the standard model-based techniques (like OCL and QVT) to be used for runtime management. We implement this approach as a tool named SM@RT [4], and apply it on several practical systems.

The rest of this paper is organized as follows. Section 2 illustrates the difficulty for developing a synchronization engine by hand whereas Section 3 presents an overview of our generation process. Section 4 and Section 5 explain how developers specify the system and how to generate the synchronization engine. Section 6 describes and discusses our case studies. Finally, Section 7 presents some related approaches and Section 8 concludes the paper.

2 Motivating Example

This section illustrates the complexity of developing a synchronization engine (SE) between a running systems and its model view.

We choose the JOnAS [12] JEE application server as a running example. A JOnAS server contains a lot of manageable elements such as EJBs, data sources (proxies to databases), etc. Each data source maintains a pool of connections to the underlying database. If the number of cached connections tends to reach the capacity of the connection pool, the database access may be delayed and the pool capacity must be enlarged. In the same manner, if the number of cached connections is always zero, the data source can be removed to release resources.

JOnAS provides a low-level interface (the JMX [5] management API) for the monitor and execution of manageable elements. But complex analysis and planning must still be performed by hand or by using external tools. Model-driven techniques and tools can help such analysis and control tasks. Take the above management scenario as an example, the administrators could use a model visualization tool (like GMF [13]) to help better understand the system, or use OCL constraints to automatically verify the server reconfiguration.

Like other model-based technologies, GMF and OCL can be only applied on MOF-compliant models (as shown in Figure 1), which is constituted by standard model elements. But the JMX API represents the running system as a specific kind of Java objects, the Managed Beans (MBeans). The integration of model-based techniques thus requires an SE which reflects the running system into a MOF-compliant model, and ensures a bidirectional consistency between the system and the model. For instance, in our JEE scenario, the SE must build a model element for each data sources on the JEE AS. When the management

[4] "SM@RT" for Supporting Models at Run-Time: `http://code.google.com/p/smatrt`

agent deletes a model element, the SE must detect this change, identify which data source this removed element stands for, and finally invoke the JMX API to remove this data source.

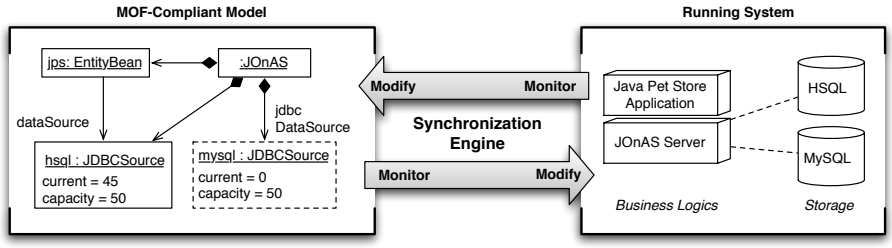


Fig. 1. A common structure of the synchronization engines

However, such an SE employs a complex mechanism and its development is therefore time consuming and error-prone. For the above scenario, the SE has to perform many functionalities: reading and writing models, monitoring and executing system changes, maintaining the mapping between model elements and system elements, handling conflicts between changes, and planning the proper subsequent changes to make the model and system consistent. In addition, SEs share many commonalities, and developing the SE from scratch is a waste of time and labor. Actually, except for monitoring and executing system changes, all the other functionalities are independent to the specific systems, and thus it is possible to achieve common solutions for them.

3 Approach Overview

We provide a generative approach to assist the development of synchronization engines. As shown in Figure 2, the inputs of our approach include a *system meta-model* specifying what kinds of elements can be managed and an *Access Model* specifying how to use the API to monitor and modify those manageable elements. Our SM@RT tool generates a SE which reflects automatically the running system into a MOF-compliant model that conforms to the system meta-model.

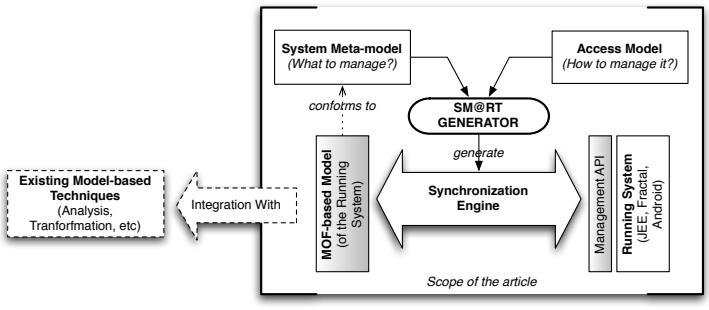


Fig. 2. Generation of Synchronization Engine: Approach Overview

Our approach is applicable on the following premises. First, we require the target system to provide a management API: our tool does not instrument non-manageable systems, nor extends inadequate APIs. Second, we reflect a *direct model* for the system (that means the model is homogeneous with the system structure: each model element stands for one system element). If an abstract model is needed, a model transformation could be used to transform this direct model into the needed forms, which is beyond the scope of this paper.

4 Modeling Management APIs

In order to generate an SE for a specific system, we need to know *what can be managed in this system*, and *how to managed it*. In this section, we present how to specify these two kinds of information as models.

According to Sicard et al. [8], a manageable running system is constituted of *managed elements*. Managed elements have *local states*. They could be *composed* by other managed elements, and they could have *connections* between each other. These four concepts can be described using the following four concepts in EMOF meta-model [14], i.e. *Class*, *Attribute*, *Aggregation*, *Association*, respectively. Figure 3 is an excerpt of the meta-model we defined for JOnAS.

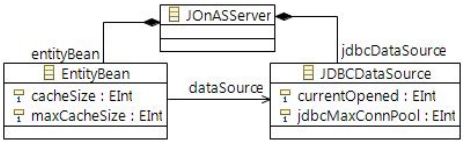


Fig. 3. The system meta-model of the JEE example

The manageable elements can be manipulated through the management API. For example, we can use `getAttribute` method of JMX to fetch the current opened connections of a data source. For a system, we require developers to specify how to invoke its management API to manipulate each type of elements, and we name this as an "access model". More formally, the access model is a partial function:

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

where $MetaElement$ is the set of all the elements in the system meta-model (classes, attributes, etc.), $Manipulation$ is the set of all types of manipulations, which are summarized in Table 1, and $Code$ is a piece of Java code.

Figure 4 shows two sample items in the access model for JOnAS. The first item defines the common code for *getting* the values of *int-typed attributes*. We obtain an instance of an MEJB remote element (Line 4), and the logic is defined as an `Auxiliary`. Then we invoke the `getAttribute` method provided. The first parameter is the reference to the current management element. The second parameter is the property name. The second sample is for adding a new data source into a server, or "loading a data source according to a specific name" in the JEE language. We first find the model element added by management

Table 1. All kinds of manipulations. For each kind of operation, we list its *name*, the types of *meta elements* it could applied, the *parameters* it required for execution, and a brief *description*. In the table, *Property* standards for attribute, aggregation and association, and the following "1" or "*" refers to the single-valued or malti-valued properties, respectively. The *Auxiliary*s are common operations defined by users, and can be used during the definition of code, as shown in the example.

name	meta element	parameter	description
Get	Property (1)	-	get the value of the property
Set	Property (1)	`newValue`	set the property as `newValue`
List	Property (*)	-	get a list of values of this property
Add	Property (*)	`toAdd`	add `toAdd` into the value list of this property
Remove	Property (*)	`toRemove`	remove `toRemove` from the list of this property
Lookfor	Class	`condition`	find an element according to `condition`
Identify	Class	`other`	check if this element equals to `other`
Auxiliary	Package	-	user-defined auxiliary operations

Fig. 4. Invoking JMX interface

```
1  //Sample 1, get the value for any kind of attributes
2  MetaElement=AnyClass::AnyIntTypedSingleValuedAttribute,
3  Manipulation=Get, Code=BEGIN
4    Management mgmt=$sys::auxiliary.getMainEntry();
5    Integer res=(Integer) mgmt.getAttribute($sys::this,$meta::prpt_name);
6    $sys::result=rest.intValue();
7  END
8  //Sample 2, add a new data source
9  MetaElement=JOnASServer::jdbcDataSource, Manipulation=Add
10 Code: BEGIN
11   String dbName=$model::newAdded.name;
12   Object[] para = {dbName,Boolean.TRUE};
13   String[] sig = {"java.lang.String","java.lang.Boolean"};
14   Management mgmt=$model::auxiliary.getMainEntry();
15   $sys::newAdded=(ObjectName)mgmt.invoke(dbserver, "loadDataSource", para, sig);
16 END
```

agents, and get the data source name (Line 11) from this element. Finally we use this name to invoke the `loadDataSource` operation (Lines 12-15).

When defining how to manipulate the systems, developers may need system information (like "what is the current system element", Line 5), system type information (like the property name, Line 5), and the inputted information by the external management agent (like the appointed name for the to-be-created data source, Line 11, such information is preserved in the corresponding model element). We defined three kinds of *specification variables*, the `system`, `meta` and `model` variables, to stand for the above three kinds of information, in order to keep developers from the details about the generation and the SE.

5 Generating the Synchronization Engine

This section presents the SEs we generated to maintain the causal links between model and system. We first explain how the generated SEs work, and then introduce how we generate the engines.

The first question for a synchronization mechanism is "*when* and *where* to synchronize". Since the model is the interaction point between the system and the management agent (MA), synchronization should be triggered *before MA read the model* and *after they write the model*. In addition, for each reading or writing, the MA only cares about part of the model. And thus, we only

synchronize the involved part of model with the running system. Such *on-demand* synchronization preserves correctness and increases performance.

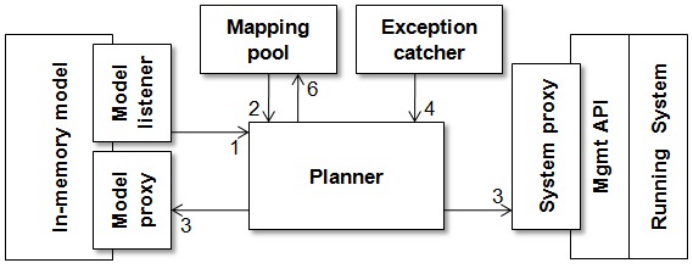


Fig. 5. Structure of the generated SE

Figure 5 shows the structure of our SE, implementing the on-demand synchronization approach we discussed before. The model we provide is in an in-memory form conforming with Ecore [13]. Each model element is represented by a Java object in the type of `EObject`. External management agents read or write this model by invoking the standard get or set methods on these model elements. The **Model Listener** listens to these model operations. For a reading operation, the listener interrupts the operation, asks the planner to do synchronize, and finally resume the original operation with the refreshed model. For a writing operation, it waits until the operation finished, and asks the planner to synchronize this modified model with the system. The **Mapping pool** maintains a one-to-one mapping between the model elements and the system elements, as a reference for the synchronization. The **Model and System proxies** are used to read the current model and system, and write the required changes (i.e. the synchronization result) back. The **Exception Catcher** implements a simple conflict handling strategy, i.e. when a conflict causes failures during the model or system manipulation, it catches the thrown exceptions and warns the management agent. Based on these auxiliary parts, the central **planner** execute a set of synchronization strategies:

$$SynchStrategy : ModOp \times MOFElem \rightarrow (ModOp \bigcup SysOp \bigcup MapOp)*$$

Each strategy defines that when a specific kind of model operations (get, set, etc.) happened on a specific part of the model (model elements, single-valued attributes, etc.), the engine will execute a *sequence of operations*. These operations manipulate the model, the system, and the mapping pool, in order to make them consistent.

Due to the space limitation, we do not explain each strategy, but use the following sample to illustrate how they work. For the JOnAS sample, in the beginning, the model contains only one element standing for the JOnAS server. The management agent invokes `get` method on this root element to see its data sources. The model listener interrupts this `get` operation, and informs the planner. Follow the synchronization strategy for *get* operations on *multi-valued aggregations*, the planner performs the following operations: It first checks the mapping pool to see that `root` stands for the JOnAS server, and then invoke

`list` on this server (See Table 1), which returns a set of `ObjectName`s pointing to the current data sources. The planner then invokes the `create` operation on the model proxy to create a new model element for each of these data sources, and refresh the mapping pool for these new model elements. Finally, the original `get` operation continues, and returns a set of newly created model elements.

Our SM@RT tool automatically generates the above SEs. The tool has two parts, a common library and a code generation engine. The common library implements mapping pool, the exception catcher, and the planner, with the synchronization strategies hard-coded inside. The code generation engine is an extension of the Eclipse Modeling Framework (EMF), and it generates the model listener, model proxy, and system proxy specific to the target system. Specifically, it generates a Java class for each of the MOF classes in the system meta-model, implementing the `EObject` interface defined by Ecore. Then it overrides the model processing methods in `EObject`, inserting the logic for listening operations and launching the synchronization planner. Finally, it wraps the pieces of API invocation code in the access model into a set of system manipulation methods, which constitutes the system proxy.

6 Case Studies

We applied our SM@RT tool to generate SEs for several practical systems, and performed several runtime management scenarios on these models, utilizing existing MOF-based model-driven techniques.

6.1 Reflecting JOnAS JEE systems

Our first case study is the full version of the running example we used before. We reflect all the 21 kinds of JEE manageable elements (including applications, EJBs, data sources, transaction services, etc.) as a MOF-compliant model, and visual it to provide a graphical management tool for JOnAS administrators.

We first define the system meta-model and the access model for JOnAS as explained in the previous sections. The resulting system meta-model contains 26 classes, 249 attributes, 21 aggregations and 9 associations. The resulting access model defines 28 pieces of code like the sample in Figure 4.

From the system meta-model and the access model, the SM@RT tool automatically generates the SE for JOnAS as a Java library. We connected this library with GMF to visualize the reflected model (just in the same way as visualizing any common Ecore models), as shown in Figure 6.

In this snapshot, the rectangles stand for the JOnAS manageable elements and the lines stand for the association between these elements. From this diagram, we see that there are two applications running on the `pku` server, which runs on one JVM, and contains several resources, including a data source named `HSQL1`. We select the data source, and the *property view* on the right side shows its attribute values. All the elements, associations and attributes depict the *current* system state. That means if we select this model element again (that causes GMF to refresh the attributes), some attribute values may change, and if we select the canvas (that causes GMF to refresh the root element), some elements

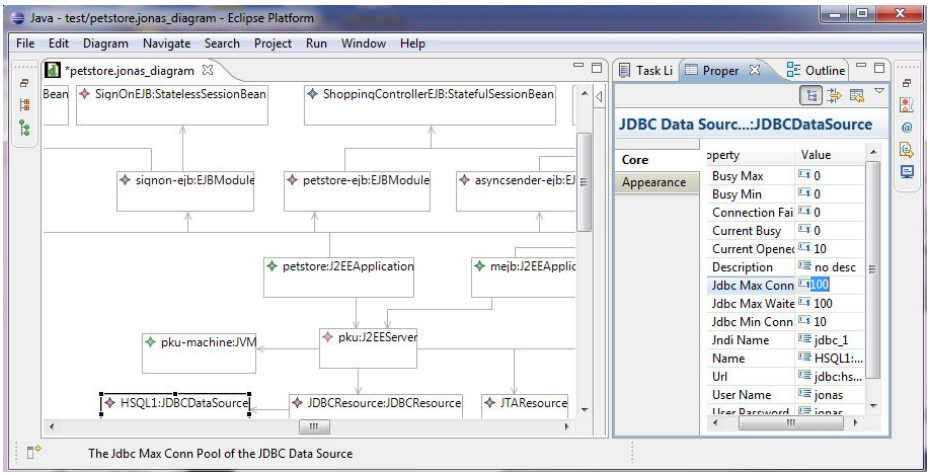


Fig. 6. A snapshot of the visualized model of JOnAS

may disappear and new elements may appear. We can also directly use this diagram to change the system. For example, if we increase the `JDBCMaxConnPool` from 100 to 200, the underlying pool will be enlarged consequently. If we create a new model element in the type of `J2EE Application`, and set its `fileName` attribute as the address of an EAR file, the synchronization engine deploys this EAR file into the system, and some new model elements will appear in the diagram, standing for the modules and EJBs inside this newly-added application.

6.2 Other case studies

Table 2. Summary of experiments

target system	API	meta-model *(elements)*	access model *(items)*	access model *(LOC)*	generated *(LOC)*	contrast *(LOC)*	techs
JOnAS	JMX	305	28	310	18263	5294	GMF
Java classes	BCEL	29	13	124	10518	3108	UML2
Eclipse GUI	SWT	31	23	178	11290	-	EMF
Android	Android	29	9	67	8732	-	OCL

Table 2 summarizes all the case studies we have undertaken. For each case, we give the target system and its management API, the numbers of elements in the system meta-model, the items in the access model and the total lines of code in these items. After that, we list the sizes of the generated synchronization engines. For the first two cases, we also list the size of the hand-written synchronization engines for comparison. Finally, we list the model-driven techniques we applied upon the generated SEs. The second case is a reproduction of the Jar2UML tool[5], which reflects the class structure in a Jar file as a (read-only) UML model. The third case supports dynamic configuration of an Eclipse window, like changing a button's caption or a label's background color. The fourth case is about using OCL rules to check the package structure of Android mobile phone systems.

[5] `http://ssel.vub.ac.be/ssel/research/mdd/jar2uml`, a use case of MoDisco [11]

6.3 Discussion

Feasibility The case studies above illustrate the feasibility of our approach: it generates SEs for a wide range of systems, and the generated SEs enable existing model-driven techniques for runtime management.

Generation Benefits Our generation approach improves the development efficiency of SEs. Among the complex functionalities of SEs (see Section 2), we only require developers to care about the monitoring and controlling of the system. Specifically, we reduce 94.1% hand-written code for the JOnAS case (310 vs. 5294 LOC), and 98% for the Java case (62 vs. 3108 LOC).

Synchronization Performance The performance of the generated SE is acceptable. For the JOnAS case, we deploy the JOnAS server and the synchronization engine on a personal computer with 3.0GHz CPU and 2.0GB memory. We spend 3.17 seconds in average to show the diagram shown in Figure 6, with 98 manageable elements in total, and we spend less than one second to refresh an element or change an attribute. The performance is similar to the default web-based management tool, the JOnAS Admin. For the Android case, we spend 1.7 seconds to perform the OCL adaptation rule.

7 Related Work

Many researchers are interested on model-based runtime management. The representative approaches include "runtime software architecture" [6, 15], "models at runtime" [16], etc. Currently, these approaches focus on the problems and ideas of model-based management, and implement their ideas on specific systems and models. Alternatively, we focus on the reflection of models for different systems, and try to provide automated support.

Some researchers also focus on reflecting different systems into standard models. Sicard et al. [8] employ "wrappers" to reflect systems states into Fractal models. Researchers of MoDisco Project [11] focus on developing "discoverers" to discover MOF-compliant models from systems. The "wrappers" and "discoverers" are similar to our SEs, but our work support developers in constructing SEs from a higher level, not by directly writing code in ordinary programming language. Another difference between our work and MoDisco is that our SEs support writing the model changes back to the system.

Bencomo et al. [17] also use model-to-text generation to automate system management. But currently they generate the configuration files specific to the Gridkit platform, while we try to generate SEs for various systems.

Our synchronization mechanism is related to the approaches on model synchronization [18]. The difference is that these approaches use the same model processing interface to manipulate the two participants of synchronization, but we try to integrate ad hoc management APIs into the synchronization process.

8 Conclusion

To efficiently leverage the use of model-based techniques at runtime, it is necessary to have a model-based view of the running system. In this paper, we

report our initial attempt towards the automated generation of synchronization engines that reflect running systems into model-based views. We require developer to specify "what to manage on the system" as a MOF meta-model, and specific "how to use the related API to do so" as an access model. From these specifications, we automatically generate the synchronization engine that reflects the system as a direct MOF compliant model. We have successfully applied our approach on several practical systems, and enabled several typical model-based techniques at runtime. As future work, we plan to give more support for developers to specify the running systems and their APIs. We also plan to perform further analysis such as model checking to ensure a deeper correctness and completeness of the generated causal link.

References

1. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: Future of Software Engineering (FOSE) in ICSE. (2007) 259–268
2. Shannon, B.: Java Platform, Enterprise Edition 5, Specifications (April 2006)
3. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.: The Fractal Component Model and its Support in Java. Software Practice and Experience **36**(11-12) (2006) 1257–1284
4. DiMarzio, J.: Android: A Programmers Guide. McGraw-Hill Osborne Media (2008)
5. Hanson, J.: Pro JMX: Java Management Extensions. (2004)
6. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE. (1998) 177–186
7. Garlan, D., Cheng, S., Huang, A., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54
8. Sicard, S., Boyer, F., De Palma, N.: Using components for architecture-based management: the self-repair case. In: ICSE '08: Proceedings of the 30th international conference on Software engineering, New York, NY, USA, ACM (2008) 101–110
9. Chauvel, F., Barais, O., Borne, I., Jézéquel, J.M.: Composition of qualitative adaptation policies. In: Automated Software Engineering Conference (ASE 2008). (2008) 455–458 Short paper.
10. Batista, T., Joolia, A., Coulson, G.: Managing Dynamic Reconfiguration in Component-Based Systems. In: Software Architecture: 2nd European Workshop, EWSA 2005, Pisa, Italy, June 13-14, 2005: Proceedings, Springer (2005)
11. MoDisco Project `http://www.eclipse.org/gmt/modisco/`
12. JOnAS Project. Java Open Application Server `http://jonas.objectweb.org`
13. Budinsky, F., Brodsky, S., Merks, E.: Eclipse Modeling Framework, project address: `http://www.eclipse.org/modeling/emf`
14. Catalog of OMG Modeling and Metadata Specifications `http://www.omg.org/technology/documents/modeling_spec_catalog.htm`
15. 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
16. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering (FOSE) in ICSE '07. (2007) 37–54
17. Bencomo, N., Grace, P., Flores, C., Hughes, D., Blair, G.: Genie: Supporting the model driven development of reflective, component-based adaptive systems. In: ICSE. (2008) 811–814
18. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In: MoDELS. (2006) 543–557

Leveraging Models From Design-time to Runtime. A Live Demo.*

Brice Morin[1], Grégory Nain[1], Olivier Barais[1,2], and Jean-Marc Jézéquel[1,2]

[1] INRIA, Centre Rennes - Bretagne Atlantique
Brice.Morin@inria.fr | Gregory.Nain@inria.fr
[2] IRISA, Université Rennes1
barais@irisa.fr | jezequel@irisa.fr

Abstract. This paper describes a demo which leverages models at design-time and runtime in the context of adaptive system, some details about the underlying approach as well as some implementation details. Our tool allows deploying and dynamically reconfiguring component-based applications, in a guided and safe way, based on the OSGi platform. It combines reflexive and generative programming techniques, based on models, to achieve this goal.

1 An Overview of the Demo

The demonstration illustrates our tool-chain on a simple HelloWorld application. In this section, we briefly present the different tools of the chain, as well as the scenario to be demonstrated during the workshop. More details about the tools are given in Sections 3 and 4 and in previous publications [1,2,3].

1.1 Quick Overview of the Tool-chain

Our tool leverages models at design-time but also at runtime in order to deploy and dynamically reconfigure component-based applications. We rely on the SCA (Service Component Architecture, see http://www.eclipse.org/stp/sca/) editor to graphically design architectural models in the Eclipse IDE. We have extended this editor with code generation capabilities. This way, we can generate at design-time all the platform-specific infrastructure we need to be able to leverage models at runtime. We currently target OSGi [4] and Fractal/FraSCAti [5] as runtime platforms.

In the remainder, we will particularly focus on the OSGi version of the tool. Once all the infrastructure has been generated, it is possible to start the tool responsible for managing adaptive systems. This tool is based on [1,2] and is an extension of the early prototype [3], demonstrated last year at the workshop [6],

* The research leading to these results has received funding from the European Community's Seventh Framework Program FP7 under grant agreements 215412 (DiVA, http://www.ict-diva.eu/) and 215483 (S-Cube, http://www.s-cube-network.eu/).

for managing Fractal-based systems. Unlike pure OSGi or Spring DM (see Section 2), this tool leverages models both for the initial deployment and for subsequent dynamic reconfigurations, preventing programmers from writing low-level reconfiguration scripts.

1.2 Components to be Manipulated during the demo

The HelloWorld application manipulates three kinds of component types: Client, Server and VoiceServer. It can be composed of any number of Client component instances and any number of (Voice)Server component instances, collaborating according to different schemes. The demo will manipulate up to 34 component instances and 33 bindings (connections) among these components.

Server components provide the IHelloWorld interface and optionally require [0..1] a VoiceServer. Depending on their real implementation, the Server components may "say hello" in different languages: English, Spanish, etc, by returning the corresponding String and optionally ask a voice server to actually say "hello", "hola", etc.

Client components have two required ports, both typed with the IHelloWorld interface:

- default: a single mandatory required port [1..1]
- others: a multiple optional required port [0..*]

Each client component is associated with a GUI: a window composed of a button and a text area. By clicking on the button, the client component simply asks the default server to say hello and, if any, all the other servers to also say hello. The results are printed in the text area, as illustrated in Figure 1.

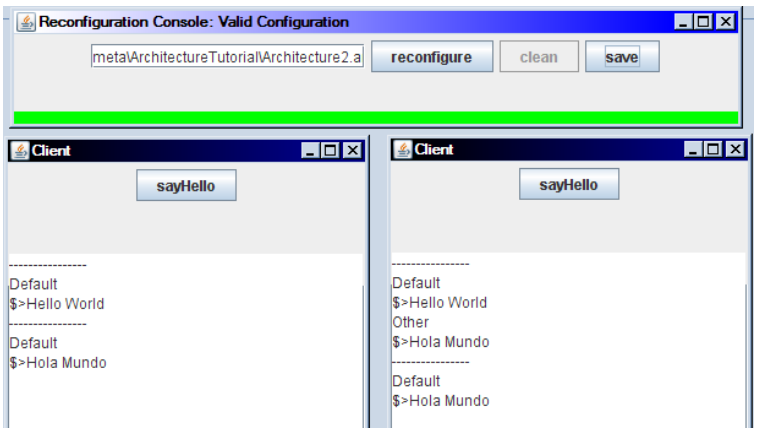


Fig. 1. Two clients after the initial deployment and a reconfiguration

Using these two types of components, it is possible to define an arbitrary number of valid and invalid configurations. For example, a configuration where a client component is not connected to a default server is invalid, since the default port is mandatory.

1.3 Scenario

We now outline the scenario to be demonstrated at the workshop.

Models At design-time At design-time, we will briefly introduce the SCA graphical editor and show how configurations are designed. Then, we will automatically generate all the code we need for the remainder of the demo.

Models At Runtime At runtime, we will demonstrate how the initial configuration is deployed and perform several attempts of model-driven reconfigurations, with valid and invalid configurations. Moreover, we will show how our tool manages unpredicted changes at runtime *e.g.*, when a component is directly uninstalled at the platform level.

1. **Start**: It displays a simple GUI to be able to load configurations
2. **Load the initial configuration**: Using the GUI, load the initial configuration (architectural model composed of two clients and two servers). It displays the GUI associated with the two clients.
3. **Say Hello**: Click on the "sayHello" button of each client. Depending on the way clients are connected with servers, this will produce a different result.
4. **Load another configuration and Say Hello**: This configuration is valid and actually produces a reconfiguration of the system (*e.g.*, one client disappear, the other uses the servers in a different way).
5. **Load another configuration**: This configuration is not valid (default port not connected). This does not produce a reconfiguration of the system.
6. **Servers crash!** The server components are manually (via the OSGi textual console) removed from the system. The tool detects this change and notifies the user that the current configuration is no more consistent.
7. **Load another configuration**: If the configuration is valid, the system will be reconfigured into a consistent configuration.

2 Background

This section briefly introduces the different technologies related to our tool and highlights their limitations.

2.1 OSGi

The OSGi (Open Services Gateway initiative) consortium is composed of famous companies from different domains: automotive industry (BMW, Volvo), mobile industry (Nokia), e-Health (Siemens), SmartHome (Philips), etc. It provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle. See http://www.osgi.org for more details about OSGi.

Typically, an OSGi bundle (component) is responsible for managing its dependencies by itself. It can try to set its dependencies when it is starting, by searching required services from the OSGi service registry, or by registering to the OSGi event mechanism to be notified when required services appear or disappear. For example, the following code fragment sets the *helloworld* reference when the client component starts.

```
/*
 * Client Component, in OSGi
 */
public class Client implements BundleActivator, IClient {

  private BundleContext context;
  private IHelloWorld helloworld;

  public void start(BundleContext context) {
    this.context = context;

    //registering the component as IClient
    context.registerService(IClient.class.getName(), this, null);

    //setting the helloworld reference
    ServiceReference[] refs = context.getAllServiceReferences(null,
        "(objectClass="+IHelloWorld.class.getName()+")");
    helloworld = (IHelloWorld)context.getService(refs[0]);
  }
}
```

OSGi provides very flexible and powerful mechanisms for managing the life-cycle of components. However, since the dependencies should be handled inside the components themselves, it is very difficult to separate the business logic from the adaptive logic and implement complex adaptation policies involving several collaborating components.

2.2 Spring DM

Spring DM (Dynamic Modules) allows managing configurations of OSGi-based applications by deploying the initial configuration from a XML file, dynamically adding, removing, and updating modules in a running system. Moreover, it has the ability to deploy multiple versions of a module simultaneously. See http://www.springsource.org/osgi for more information on Spring DM.

Typically, a Spring bean (component) is a POJO with getters and setters for the reference that can be accessed and setted. Unlike OSGi, components are not responsible for setting their dependencies by themselves. On the contrary, these references are set from the outside, by calling the appropriate getters/setters. For example, the following code fragment illustrates the Client component.

```
/*
 * Client Component, in Spring
 */
public class Client implements IClient {

  private IHelloWorld helloworld;

  public IHelloWorld getHelloworld(){
    return helloworld;
  }

  public void setHelloworld(IHelloWorld helloworld){
    this.helloworld = helloworld;
  }
}
```

The initial runtime configuration is instantiated, deployed and started by loading the XML file describing this initial configuration, similarly to an ADL (Architecture Description Language). This file mainly describes the component types

(factories) needed to instantiate component instances (beans in the Spring terminology), and how beans are wired together.

Spring DM does not allow the declarative updating of this XML file in order to dynamically reconfigure the system. On the contrary this should be done programmatically by modifying the properties of the components and calling the `modifyConfiguration(ServiceReference ref, Dictionary props)` method provided by the Configuration plugin, or directly calling the getters/setters provided by the components.

This is a major drawback since the initial configuration and the subsequent reconfigurations are not handled in a consistent way. On the one hand, the initial configuration is instantiated from a declarative specification describing the overall configuration of the system. On the other hand, the dynamic reconfigurations are realized in an imperative style, which requires the developer to set the dependencies in a programmatic way. In the case of a large reconfiguration, this requires to implement a long script describing how all the properties are updated. Moreover, it is very difficult to understand or validate the new configuration *a priori*, since no explicit representation (model or XML file) of this new configuration exists.

3 An Overview of SC@rt

This section describes SC@rt (SCA at runtime), to be demonstrated (live demo) during the workshop. The main objective of our approach is to leverage models at design-time in order to support the initial deployment of the system, but also at runtime, to provide a high-level support for dynamic reconfigurations.

3.1 Designing Architecture with the SCA Editor

The first step consists in designing the component types to be manipulated by the application, in the SCA editor. Here, we only manipulate two types: client and server. This diagram simply consists of non-connected components describing their provided and required ports.

Then, it is possible to define different configurations of the hello world application. In practice, this consists in creating a new diagram, and copy/pasting component types from the type diagram in order to instantiate component instances. These instances can then be connected together on compatible ports. Figure 2 illustrates one possible configuration.

3.2 Generating the Code of Component Types

After the type diagram has been designed, we propose to generate the code of the type components. These type components are factories responsible for instantiating component instances. We have extended the SCA editor so that the code generation is simply invoked by a right click on the diagram. The code of the factory is rather systematic and only varies on some well identified points. We use String Template (see http://www.stringtemplate.org/) to generate this code.

3.3 Leveraging Architecture Models to Deploy and Reconfigure the System

Figure 3 illustrates our causal connection between an architectural model and a running system, based on our previous works [1,2].

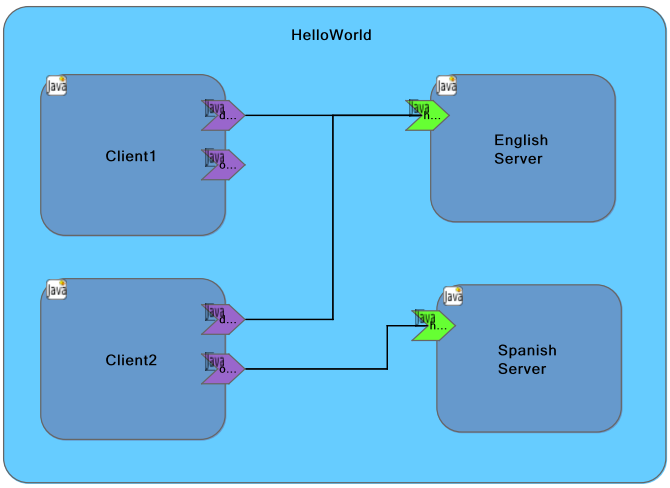


Fig. 2. One possible configuration of the Hello World application

The key idea is to keep an architectural model synchronized with the running system [6]. This reflection model, is updated by observers integrated in the execution platform (Figure 3, 1) when significant changes appears in the running system (addition/removal of components/bindings).

When a target architectural model is defined (*e.g.*, by modifying a copy of the reflection model), it is first validated using classic design-time validation techniques, such as invariant checking or simulation. This new model, if valid, represents the target configuration the running system should reach.

Then, we generate a reconfiguration script by first comparing the source configuration (the reflection model) with the target configuration (Figure 3, 3) and generating an ordered set of reconfiguration commands. This set of commands describes a safe reconfiguration script (no life-cycle errors or dangling bindings) which is submitted (Figure 3, 4) to the running system in order to actually reconfigure it. Note that when a new component is added into the model, we generate, compile and package some parts of its code at runtime (the Activator and the MANIFEST.MF)[3]. This way, component instances are handled as OSGi bundle, making it possible to properly install, uninstall, start and stop them.

Finally, the reflection model is automatically updated when the reconfiguration commands are correctly executed. Commands not correctly executed do not update the reflection model and are logged so that they can be post-processed *e.g.*, to implement a roll-back. If all the commands are correctly executed, the updated reflection model becomes equivalent to the target model (Figure 3, 1).

[3] the business logic of the component is already present and compiled in the factory components

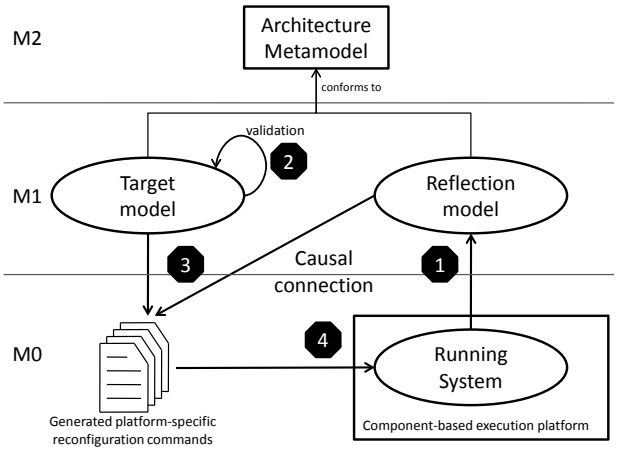


Fig. 3. Strong synchronization from runtime to model. Delayed synchronization (after validation) from model to runtime

4 Implementation Details

This section gives some implementation details of the prototype we have described.

4.1 Reconfiguration Commands

The dynamic reconfiguration process is based on the Command design pattern. Basically, we represent a reconfiguration script as a list of commands, as explained in [1,2]. These commands are ordered according to their priority: *i*) remove binding, *ii*) remove component, *iii*) add component and *iv*) add binding.

We define the parameter of each command as public attributes. The `check` method is responsible for verifying that the parameters of the command have been well initialized. The `execute` method actually performs an atomic reconfiguration on the running system (*e.g.* add a component). Finally, the `doAck` method is called when the command has successfully been executed.

```
1  public abstract class PlatformCommand {
2
3    private boolean ack = false;
4
5    abstract public int getPriority();
6
7    //Checks the consistency of the command
8    abstract public boolean check();
9
10   //Executes the command i.e., a performs a runtime adaptation
11   abstract public void execute();
12
13   public boolean ack(){
14     return ack;
15   }
16
17   /* Acknowledge the command i.e., update the reflection model
18      Should be overriden by concrete commands. */
19   public void doAck(){
```

```
20      ack = true;
21    }
22  }
```

4.2 OSC@rt: Models@Runtime over OSGi

The current prototype does not directly use SCA models at runtime. Instead, it uses architectural models conforming to a core architecture metamodel described in [3], to reduce the memory overhead at runtime implied by the causally connected model. However, we have defined a bi-directional model transformation between SCA and our metamodel, in Kermeta [7].

The main class of this prototype is the `OSGiCausalLink` class illustrated (fragment) in the following scripts.

```
public class OSGiCausalLink implements BundleActivator, CausalLink,
    EventHandler, BundleListener {
  private art.System system; //the reflection model
  private art.System updateModel; //a new model to switch to

  private Checker checker;

  public void reconfigure() {
    if (checker.check(updateModel)){
      computeMatch(system, updateModel);
      getCommands();
      for(PlatformCommand cmd : commands){
        cmd.execute();
      }

      Timer timer = new Timer();
      AckTimerTask att = new AckTimerTask();
      att.commands = commands;
      timer.schedule(att, ackPeriod);
    }
  }
```

The main method of the `OSGiCausalLink` class is `reconfigure`. This method loads a new architectural model and leverages this model to automatically adapt the running system. This prevents the programmer from writing long imperative reconfiguration scripts. This method first performs a model comparison between the reflection model and the new model to determine the commonalities (what has not changed) and the differences between these two models (what has been added or removed). Then, by analyzing the result of the model comparison, a sequence of reconfiguration commands is instantiated and finally executed. A separate thread (`AckTimerTask`) checks whether all the commands have been acknowledged within a given period, or not.

```
private void addBinding(TransmissionBinding b, ComponentInstance client) {
  AddBindingOSGi cmd = new AddBindingOSGi();
  cmd.b = b;
  cmd.client = client;
  commands.add(cmd);
}

private void doAddBinding(TransmissionBinding b, ComponentInstance client) {
  client.getBinding().add((TransmissionBinding)b);
}
```

The `addBinding` method shows how a command is instantiated. When a new binding is detected in the model, we simply instantiate and initialize the right command. Note that, at this point, the reflection model has not been updated. In other words, the new binding has not been actually added to the reflection model. The binding will be added to the reflection model, in the `doAddBinding` method, only when the command will be properly acknowledged.

```
public void handleEvent(Event event) {
  PlatformCommand cmd = (PlatformCommand)event.getProperty("command");

  if (event.getTopic().startsWith("binding/ok/")){
    TransmissionBinding b = (TransmissionBinding)event.getProperty("binding");
    doAddBinding(b, ((AddBindingOSGi)cmd).client);
    cmd.doAck();
  }
  else if (event.getTopic().startsWith("binding/nok/")){
    TransmissionBinding b = (TransmissionBinding)event.getProperty("binding");
    System.err.println("Problem␣when␣binding␣"+b);
  }
  ...
}
```

We rely on the standard `EventAdmin` service provided by OSGi to acknowledge commands. When a command properly execute, an event containing the command is posted on the appropriate topic *e.g.*, "binding/ok". When receiving this event (`handleEvent(Event event)`), we simply call the `doAck` method of the command and update the reflection model (*e.g.*, by calling the `doAddBinding` method). When a command does not execute properly, and event is posted on another topic (*e.g.*, on "binding/nok"). In this case, the command is not acknowledged and the reflection model is not updated. In this case, the `AckTimerTask` generates a report specifying which commands have been acknowledged and which commands have not.

```
public void bundleChanged(BundleEvent event) {
  Bundle b = event.getBundle();

  if (event.getType()==BundleEvent.UNINSTALLED) {
    ComponentInstance cpt = runtime2model.get(b);
    system.getRoot().getSubComponent().remove(cpt);
    runtime2model.remove(b);
  }
  else if (event.getType()==BundleEvent.STOPPED){
    ComponentInstance cpt = runtime2model.get(b);
    cpt.getBinding().clear();
    removeAllDanglingBindings(cpt);
  }
}
```

Our causal link implements the standard `BundleListener` interface specified in the OSGi standard. This way, we are automatically notified when components are stopped or uninstalled. When a component is stopped, we clear all its dependencies and clear all the dependencies of all the components using this stopped component, by calling the `removeAllDanglingBindings`, and remove all the associated bindings from the reflection model. Then, if the component is uninstalled, we simply remove it from the reflection model.

5 Conclusion

This paper has presented our tool-chain, to be demonstrated at the workshop. This tool chain leverages models at design-time but also at runtime to deploy and dynamically reconfigure component-based applications, in a guided and safe way. At design-time, we use the SCA editor to design the architecture, and code generation techniques to produce all the infrastructure we need at runtime. At runtime, the reconfiguration process is totally driven by the design models. Before adaptation, we check that the target configuration is valid. In this case, we generate the corresponding reconfiguration script, based on the command pattern. Our tool also handles (in a limited way) unpredicted events, and is able to update the reflection model accordingly and notifies the user when this model becomes invalid.

In future work, we plan to improve the adaptation process by considering more constraints during the generation of commands. Currently, commands are ordered in a very simple way in order to avoid common errors like dangling bindings. We would like to consider other constraints, specified by the architect, such as *Component A* should be stopped *before Component B*, etc.

References

1. Morin, B., Barais, O., Nain, G., Jézéquel, J.M.: Taming Dynamically Adaptive Systems with Models and Aspects. In: ICSE'09: 31st International Conference on Software Engineering, Vancouver, Canada (May 2009)
2. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.M., Solberg, A., Dehlen, V., Blair, G.: An aspect-oriented and model-driven approach for managing dynamic variability. In: MODELS'08: ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems, Toulouse, France (October 2008)
3. Morin, B., Barais, O., Jézéquel, J.M.: K@rt: An aspect-oriented and model-oriented framework for dynamic software product lines. In: 3rd International Workshop on Models@Runtime, at MoDELS'08, Toulouse, France (oct 2008)
4. The OSGi Alliance: OSGi Service Platform Core Specification, Release 4.1 (May 2007) http://www.osgi.org/Specifications/.
5. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.: The FRACTAL Component Model and its Support in Java. Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems **36**(11-12) (2006) 1257–1284
6. N. Bencomo, G. Blair, R.F.: Proceedings of the international workshops on models@run.time (2006-2008)
7. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In L. Briand, S.K., ed.: Proceedings of MODELS/UML'2005. Volume 3713 of LNCS., Montego Bay, Jamaica, Springer (October 2005) 264–278

Evolving Models at Run Time to Address Functional and Non-Functional Adaptation Requirements*

Andres J. Ramirez and Betty H.C. Cheng

Michigan State University
Department of Computer Science and Engineering,
3115 Engineering Building, East Lansing, MI 48824
{ramir105,chengb}@cse.msu.edu

Abstract. Increasingly, applications need to dynamically self-reconfigure as new environmental conditions arise at run time. In order to self-reconfigure, an adaptive system must determine which target system configuration will yield the desired behavior based on current execution conditions. However, it may be impractical to evaluate all potential system configurations in a reasonable time frame. This paper presents a model-based approach that leverages evolutionary computation to automatically generate, at run time, target system models that balance tradeoffs between functional and non-functional requirements in response to run-time monitoring of environmental conditions. Specifically, this approach generates graph-based representations of architectural models for potential target system configurations. The current run-time system models serve to constrain the degree of change and novelty in the newly generated models. This approach is applied to the dynamic reconfiguration of a set of remote data mirrors, where operational and reconfiguration costs are minimized, while maximizing data reliability and network performance.

Key words: dynamic reconfiguration, non-functional requirements, evolutionary computation, run-time models.

1 Introduction

It is increasingly important for applications to dynamically adapt as requirements change and new environmental conditions arise [1]. In addition, it is important for adaptive systems to self-reconfigure with little or no human input to help prevent costly downtimes while code is being modified. To address this concern, IBM proposed autonomic computing where a system manages itself to achieve a system administrator's high-level goals through self-* properties such as self-configuration and self-optimization [2]. To self-reconfigure, an adaptive system must automatically determine which target system configuration will

* This work has been supported in part by NSF grants CCF-0541131, CNS-0551622, CCF-0750787, CNS-0751155, IIP-0700329, and CCF-0820220, Army Research Office W911NF-08-1-0495, Ford Motor Company, and a grant from Michigan State University's Quality Fund.

yield the desired behavior in response to current system and environmental conditions, while also taking into consideration tradeoffs between functional and non-functional requirements. It may be impractical, however, to evaluate all potential target systems in a reasonable amount of time. This paper presents a model-based approach that leverages evolutionary computation to generate, at run time, target system models that balance tradeoffs between functional and non-functional requirements in response to changing environmental conditions.

Self-adaptive systems comprise three key enabling technologies: monitoring, decision-making, and reconfiguration. Monitoring enables an application to be aware of its environment to detect conditions that warrant reconfiguration. Decision-making analyzes monitoring information to determine how the application should be reconfigured. Reconfiguration enables an application to modify itself to fulfill its requirements. Many self-adaptive systems apply model-based techniques to determine which target system configuration will yield the desired system behavior in response to current environmental conditions [3–7]. While powerful, these approaches typically use scenarios identified at *design time* to guide self-reconfigurations. Furthermore, as the complexity of adaptive logic grows, maintaining the set of models and reconfiguration plans may become unmanageable and potentially inconsistent. Recently, researchers have applied evolutionary computation techniques to the design of self-adaptive systems [8–10]. While these approaches enable developers to explore richer sets of behavioral models that satisfy system requirements, they are only applicable at design time due to the significant amount of time required to generate these models.

This paper presents Plato-MDE, an evolutionary computation-based approach for generating target system models at *run-time* in response to changing requirements and environmental conditions. Each target system model represents a potential system configuration that may be reached through a sequence of reconfiguration steps. Plato-MDE evaluates each generated target system model to determine its suitability given current system conditions. In addition, Plato-MDE leverages current system models to constrain the degree of change in the generated target models. As a result, Plato-MDE enables an adaptive system to implicitly control the complexity and novelty of the reconfiguration itself at run time. Moreover, rather than prescribing explicit reconfiguration plans at design time in anticipation of possible reconfiguration scenarios, developers need only specify the relative importance of each functional and non-functional concern to apply Plato-MDE.

Plato-MDE supports a model-based approach that leverages information from run-time system models to optimize the generation of target system models. In particular, Plato-MDE applies domain-independent evaluation functions to compare the structure and configuration of each generated target system model against a current architectural model of the executing system. The structural and configuration differences identified from this analysis enable Plato-MDE to implicitly constrain the novelty of generated target system models, and thus control the complexity and cost of the reconfiguration itself. For example, to minimize reconfiguration costs, Plato-MDE might focus on generating target system mod-

els that are structurally similar to the current system model, thereby reducing the number of structural changes required to reconfigure the system. Plato-MDE accomplishes these objectives at run-time by applying *genetic algorithms* [11] to automatically balance tradeoffs in functional and non-functioanl requirements. As a result, Plato-MDE evolves target system models at run-time, where better solutions tend to eventually dominate the solution space.

We applied Plato-MDE to the dynamic reconfiguration of an overlay network for diffusing data to a collection of remote data mirrors [12]. Specifically, Plato-MDE was able to evolve target system models that not only maintained connectivity across the network of remote data mirrors such that data could be diffused to every node, but also minimized operational and reconfiguration costs while maximizing data reliability and network performance. Furthermore, Plato-MDE was able to leverage run-time system models to control the complexity and novelty of the generated target system reconfigurations, thus implicitly controlling reconfiguration costs at run time. The remainder of this paper is organized as follows. Section 2 overviews genetic algorithms. Section 3 overviews Plato-MDE. In Section 4 we present a case study in which we apply Plato-MDE and provide preliminary results. Section 5 compares Plato-MDE to other self-adaptation approaches. Lastly, Section 6 summarizes our results and presents future work.

2 Background: Genetic Algorithms

A genetic algorithm is a stochastic search-based technique for optimization problems that comprises a population of individuals, each encoding a candidate solution in a chromosome representation [11]. Fitness functions are used in each iteration of the algorithm to evaluate an individual's encoded solution. This fitness information enables a genetic algorithm to select a subset of promising individuals for further processing. Specifically, the operation of *crossover* is used to exchange building blocks between two fit individuals, hopefully creating offspring with higher fitness values than either parent. The crossover operator loses genetic variation in a population throughout generations, possibly leading to premature convergence and suboptimal solutions. In order to counter this effect, *mutation* re-introduces genetic variation by randomly changing parts of an individual's encoded solution according to specific mutation rates [11]. Generally, genetic algorithms are executed until the algorithm converges upon a single solution or the alloted execution time is exceeded.

3 Plato-MDE

Plato-MDE is a genetic algorithm-based approach developed for generating target system models at run time in response to changing environmental conditions, while balancing tradeoffs between functional and non-functional requirements. Plato-MDE extends Plato [13] with a model-based approach that focuses on generating architectural models and properties of the connectors between the components at run time. In contrast, Plato did not consider structural differences between the current application's architecture and the generated target system

reconfiguration models. This extension enables Plato-MDE to implicitly control the cost of a reconfiguration at run time. As the data flow diagram (DFD) in Figure 1 illustrates, several inputs and configurations must be supplied in order to apply Plato-MDE to the decision-making process of a self-adaptive system. At a high-level of abstraction, Plato-MDE accepts data from the monitoring infrastructure and outputs a set of target system models that specify new suitable reconfigurations. As an initialization step, developers must first configure Plato-MDE for the application's domain and specify how the quality of a target system model should be evaluated. Next, we describe the use of Plato-MDE in detail.

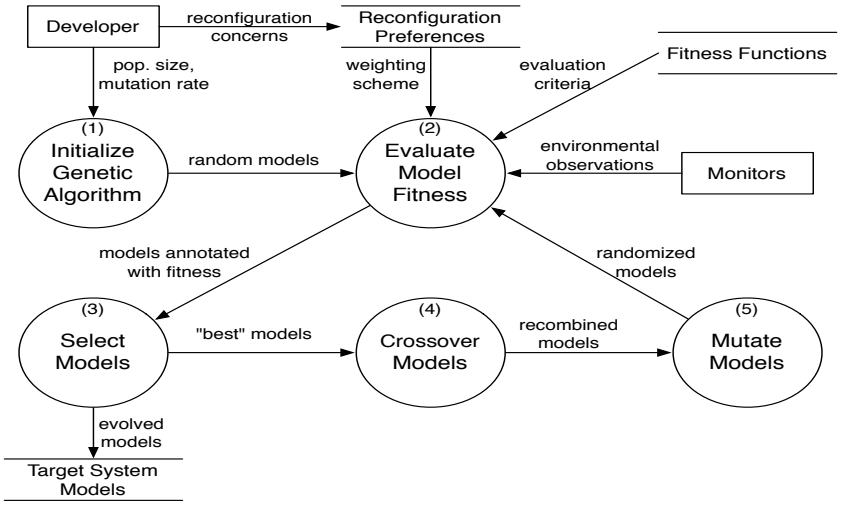


Fig. 1. Data Flow Diagram for Plato-MDE

Step 1. Initialize Genetic Algorithm. To apply Plato-MDE, a developer must configure operational parameters that determine how the genetic algorithm will execute as well as what will be used to assess the quality of the output it produces. In particular, developers must specify parameters such as population size, maximum number of generations, and mutation rates for Plato-MDE. The population size reflects how many potentially different target system models are being examined at any single point in time. Similarly, the number of generations limit the amount of time that Plato-MDE may spend generating target system models. Lastly, mutation rates indicate the degree of randomness that Plato-MDE will apply to generate target system models. Experimentation is typically required to discover suitable parameters for different application domains.

The first step in Plato-MDE, highlighted in Figure 1 as *(1)*, creates a population of random individuals. In Plato-MDE, each individual encodes a graph-based model representation of the target application's architecture. Specifically, the application is abstracted to a set of components and their interconnections, both annotated with sets of reconfigurable properties that describe their configurations and state. For example, a property in a networked application may specify whether a link is active or not and which communication protocol is currently

selected. This representation, similar to architectural models, is appropriate for abstracting relevant details of the executing system [3, 5, 7].

Step 2. Evaluate Model Fitness. Fitness functions, akin to utility functions, are used to map an individual's encoded solution to a numerical value proportional to its overall quality [11]. As Figure 1 illustrates in *(2)*, Plato-MDE applies fitness functions to assess the suitability of a particular target system model based on current system conditions supplied by the application's monitoring infrastructure. Plato-MDE applies domain-dependent fitness functions to evaluate target system models from a domain-specific perspective, such as approximating the performance and reliability of a specific network based on a protocol's configuration. In addition, Plato-MDE applies domain-independent fitness functions to evaluate target system models from structural and behavioral perspectives. For instance, Plato-MDE can approximate reconfiguration costs by identifying the structural and configurational changes between the current system model and the generated target system reconfiguration. Therefore, to minimize reconfiguration costs at run time, Plato-MDE could assign higher fitness values to target system models whose structure and configuration are most similar to the current system model.

Developers may also supply a weighting scheme that will be associated with specific fitness functions to indicate the relative importance of different reconfiguration priorities. Moreover, developers can also introduce high-level code to rescale the weighting scheme of individual fitness functions if requirements are likely to change while the application executes. Updating reconfiguration priorities at run time enables Plato-MDE to generate different reconfiguration plans that address changes in requirements.

Step 3. Selection. A selection strategy determines which individuals in the population should be explored further in future generations. As step *(3)* in Figure 1 illustrates, Plato-MDE applies a tournament selection strategy [11] to determine which target system models to compare. Specifically, two target system models are selected at random from the population and their relative fitness value is compared. Whichever target system model has a higher fitness value *survives* and moves onto the next generation. This selective pressure, similar to natural selection in living organisms, drives Plato-MDE to concentrate its search towards more promising target system models that are suitable for current system conditions. Once the maximum number of generations are executed, the most fit target system model is selected as the result.

Step 4. Crossover. The goal of the crossover operator is to construct new solutions by recombining key building blocks from existing solutions in the current population [11]. Similarly, as Figure 2 illustrates, Plato-MDE applies a customized crossover operator that works on architectural models by exchanging the key elements between two target system models, referred to as parents, to produce two potentially new offspring target system models at run time. Specifically, the Plato-MDE crossover operator generates two new target system models by randomly exchanging the components, interconnections, and properties of both parents and recombining them into offspring individuals. As a result, the

crossover operator enables Plato-MDE to combine elements of good solutions to form even better solutions.

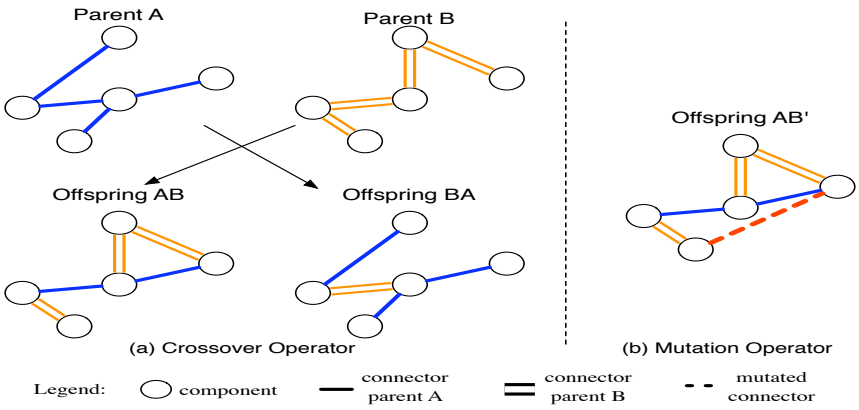


Fig. 2. Crossover and Mutation operators in Plato-MDE

Step 5. Mutation. The goal of the mutation operator is to introduce variation into the population and prevent premature convergence [11]. As Figure 1 shows in step *(5)*, Plato-MDE applies a custom mutation operator to randomly change properties of components and interconnections in an architectural model. Specifically, the mutation operator accepts an architectural model as input and randomly reassigns component and interconnection properties. For example, Figure 2(b) shows how a previously nonexistent interconnection has been created between two components (dashed line) in the architectural model. As a result, the mutation operator enables Plato-MDE to explore additional target system models at run time that cannot be generated solely through the crossover operator.

4 Case Study

This section presents a case study where we use Plato-MDE within a simulated industrial application whose primary objective is to diffuse data to a set of 25 remote data mirrors [12] across dynamic and unreliable networks. In this application, Plato-MDE generates target system models of an overlay network used to diffuse data to every remote data mirror. In contrast to previous experiments [13], this case study leverages run-time system models to constrain the degree of change involved in a particular reconfiguration. Note that the experiment presented in this section was executed on a MacBook Pro with a 2.53GHz Intel Core 2 Duo Processor and 4GB of RAM. In addition, we performed 30 trials of the experiment, for statistical purposes, and present the averaged results.

4.1 Remote Data Mirroring

Remote data mirroring is a technique for duplicating and storing data at one or more secondary sites to physically isolate copies from failures that may affect the primary copy [12]. A key benefit of remote data mirroring is that important

data continues to be accessible even if one copy is lost or becomes unreachable. Designing and deploying remote data mirror solutions, however, is a complex task due to the competing objectives of maximizing performance while minimizing operational costs and data loss potential [12]. For instance, each network link used to propagate data incurs an operational cost and is characterized by measurable throughput, latency, and loss rates. Moreover, each network link distributes data in one of two propagation modes. In *synchronous* propagation the secondary site receives and applies each write before the write completes at the primary site [12]. In *asynchronous* propagation, updates are batched and periodically distributed to secondary sites. While synchronous propagation provides better data reliability than asynchronous propagation, it tends to consume large amounts of network bandwidth in the process. In contrast, asynchronous propagation fails to provide the same level of reliability as synchronous propagation, but tends to achieve better network performance.

In this case study we apply Plato-MDE to dynamically reconfigure a set of 25 remote data mirrors diffusing data across a dynamic and unreliable network. In particular, Plato-MDE must maintain connectivity across the network of remote data mirrors while minimizing operational and reconfiguration costs, and maximizing data reliability and network performance.

4.2 Applying Plato-MDE to Remote Data Mirroring

In Plato-MDE, every individual in the population encodes an architectural model that specifies potential reconfigured target systems. For this case study, each component in the encoded architectural models represents a remote data mirror capable of producing data at a specific rate. Similarly, every interconnection in the encoded architectural models represents an overlay network link capable of propagating data between remote data mirrors. Therefore, in addition to specifying whether each connection is active or inactive, each connection is also associated with one of seven possible propagation methods [13]. It is important to note that with n overlay network links and m propagation methods, over $2^{\frac{n(n-1)}{2}} * m^n$ potential configurations exist. Thus, with a complete overlay network of 25 remote data mirrors, approximately $7^{300} * 2^{300}$ potential target system models exist, far too many configurations to exhaustively evaluate in a reasonable amount of time.

Plato-MDE extracts data from the application's monitoring infrastructure to maintain an architectural model of the executing system. Many different metrics can be gathered, however, for this case study, the monitoring infrastructure measures the throughput, latency, bandwidth, and data loss rates of each overlay network link that can be used to propagate data between remote data mirrors. Plato-MDE leverages this current system model to evaluate each generated target system model and approximate the effects of different network configurations. To this end, we applied a set of domain-dependent fitness functions to evaluate network configurations in terms of operational costs, network performance, and data reliability. Plato-MDE also applied simple model checks to ensure generated system models did not violate either budget or connectivity constraints. In addition, domain-independent fitness functions compute the degree of change

between pairs of architectural models by identifying the structural and configurational changes between them, enabling Plato-MDE to implicitly control reconfiguration costs.

4.3 Experimental Results

The goal for the initial overlay network design was to minimize operational costs, possibly at the expense of incurring poor network performance and data reliability. To generate this type of network, we supplied Plato-MDE with a vector of reconfiguration priorities where all coefficients were set to zero except for cost. As Figure 3(a) illustrates, Plato-MDE produced a spanning tree overlay network where every node is connected but no link redundancy is provided. This overlay network design minimizes operational costs by activating the minimum number of network links required to maintain connectivity and enable remote data mirrors to diffuse data. However, this overlay network design does not provide much data reliability. In particular, a single link failure in the overlay network would disconnect the set of remote data mirrors and data may be lost.

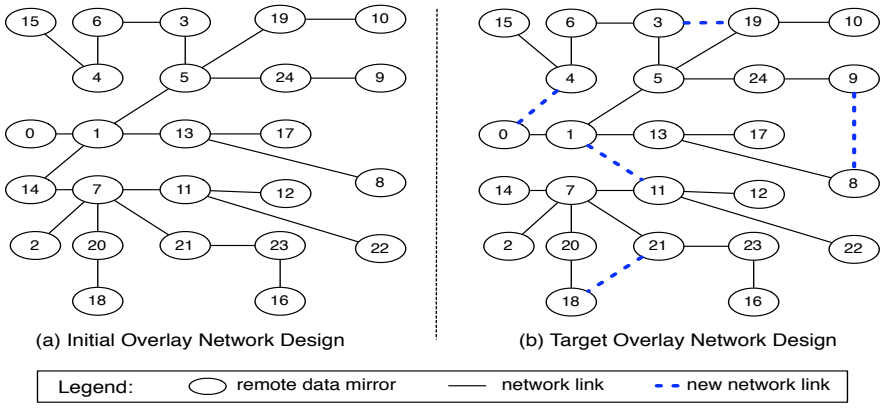


Fig. 3. Source and Target Network Design Models.

Next, we randomly selected an active link propagating data in the initial overlay network and set its operational status to *faulty*. This network link state corresponds to a link failure, thereby disconnecting the network of remote data mirrors and prompting Plato-MDE to reconfigure the overlay network. While Plato-MDE could have been invoked with the same vector of reconfiguration priorities to generate another overlay network design that re-establishes connectivity across the set of remote data mirrors while minimizing operational costs, the reconfiguration priorities were rescaled in an attempt to prevent future link failures from disconnecting the set of remote data mirrors. Specifically, the new vector of reconfiguration priorities changed the importance of minimizing operational costs to 12%, maximizing network performance to 12%, maximizing data reliability to 38%, and target model similarity to 38%. With this new vector of reconfiguration priorities, Plato-MDE produced target system models that reused the underlying network structure while adding redundant links and setting most propagation methods to synchronous mode.

Figure 3(b) shows an example overlay network design produced by Plato-MDE to re-establish connectivity within the set of remote data mirrors. This target system model satisfies the two primary design concerns specified in the vector of reconfiguration priorities: increased data reliability and reduced reconfiguration overhead. By increasing the importance of data reliability, Plato-MDE generated overlay networks with redundant links and set most propagation methods in the overlay network links to synchronous mode. Figure 3 also illustrates how the target overlay network reuses a significant portion of the underlying initial network. While Plato [13] would generate target system models without taking into account the complexity or cost of the reconfiguration, Plato-MDE preserved most of the initial network's structure to implicitly reduce the cost of reconfiguration at run time. Plato-MDE took approximately 30 seconds or less to begin converging upon suitable target system reconfigurations, which is within the acceptable time frame for remote data mirroring.

5 Related Work

Several approaches for enabling self-adaptive behavior leverage architectural models at run time to evaluate system conditions and select the most suitable reconfiguration in response to current environmental conditions. For instance, the C2 framework [7] applies software architectural models to plan, coordinate, and implement reconfigurations at run time. In addition, both the Performance Management Framework (PMF) [5] and the Rainbow Adaptation Framework [3, 4] instantiate architectural models with run-time monitoring information to determine when and how to reconfigure a system. While Plato-MDE adopts a similar approach for determining how the application should be reconfigured, several key differences exist. For instance, Plato-MDE is capable of generating *any* target system model reachable through a series of reconfiguration steps. In contrast, C2 [7] relies on a repository of pre-generated target models, and PMF [5] and Rainbow [3, 4] generate new target models through predetermined combinations of their reconfiguration steps. Furthermore, C2 [7], PMF [5], and Rainbow [3, 4] encode their reconfiguration priorities at design time. Plato-MDE, on the other hand, can update reconfiguration preferences at run time to address changes in requirements and environmental conditions. Lastly, while PMF [5] and Rainbow [3, 4] evaluate the utility of target reconfigurations to predict their impact upon the system, Plato-MDE also leverages this utility information to guide the search towards more promising target system models.

6 Conclusions

We have presented Plato-MDE, a model-based approach that leverages evolutionary computation to generate, at run time, target system models that balance tradeoffs between functional and non-functional requirements in response to current system conditions. Plato-MDE extends Plato [13] with domain-independent model-based fitness functions that analyze the structural differences between current and target system models to implicitly control reconfiguration costs at run time. We have successfully applied Plato-MDE to the dynamic reconfiguration of a set of remote data mirrors, where generated target system models

enable data diffusion among remote data mirrors by maintaining network connectivity while minimizing costs and maximizing network performance and data reliability. Future directions for this work include exploring how to decentralize the architecture of Plato-MDE to reduce potential performance bottlenecks.

References

1. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Betty H.C. Cheng: Composing adaptive software. Computer **37**(7) (2004) 56–64
2. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1) (2003) 41–50
3. 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
4. Cheng, S.W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 International Workshop on Self-adaptation and Self-Managing Systems, New York, NY, USA, ACM (2006) 2–8
5. Caporuscio, M., Marco, A.D., Inverardi, P.: Model-based system reconfiguration for dynamic performance management. Journal of Systems and Software **80**(4) (September 2007) 455–473
6. Mikalsen, M., Paspallis, N., Floch, J., Stav, E., Papadopoulos, G.A., Chimaris, A.: Distributed context management in a mobility and adaptation enabling middleware (madam). In: SAC'06: Proc. of the 2006 ACM symposium on Applied Computing, New York, NY, USA, ACM (2006) 733–734
7. Oreizy, P., Gorlick, M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems **14**(3) (1999) 54–62
8. Goldsby, H.J., Betty H.C. Cheng, McKinley, P.K., Knoester, D.B., Ofria, C.A.: Digital evolution of behavioral models for autonomic systems. In: Proceedings of the fifth IEEE International Conference on Autonomic Computing, Washington, DC, USA, IEEE Computer Society (2008) 87–96 (Best Paper Award)
9. Goldsby, H.J., Betty H.C. Cheng: Automatically generating behavioral models of adaptive systems to address uncertainty. In: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, Berlin, Heidelberg, Springer-Verlag (2008) 568–583 (Distinguised Paper Award)
10. Knoester, D.B., Ramirez, A.J., Cheng, B. H.C., McKinley, P.K.: Evolution of robust data distribution among digital organisms. In: Proceedings of the 11th annual conference on Genetic and Evolutionary Computation (GECCO '09), Montreal, Canada (July 2009) 137–144 (Nominated for Best Paper)
11. Holland, J.H.: Adaptation in Natural and Artificial Systems. MIT Press, Cambridge, MA, USA (1992)
12. Ji, M., Veitch, A., Wilkes, J.: Seneca: Remote mirroring done write. In: USENIX 2003 Annual Technical Conference, Berkeley, CA, USA, USENIX Association (June 2003) 253–268
13. Ramirez, A.J., Knoester, D.B., Cheng, B. H.C., McKinley, P.K.: Applying genetic algorithms to decision making in autonomic computing systems. In: Proceedings of the Sixth International Conference on Autonomic Computing (ICAC'09), Barcelona, Spain (June 2009) 97–106 (Best Paper Award)

On the Role of Features in Analyzing the Architecture of Self-Adaptive Software Systems

Ahmed Elkhodary, Sam Malek, Naeem Esfahani

Department of Computer Science
George Mason University
{aelkhoda, smalek, nesfaha2}@gmu.edu

Abstract. In traditional software families, feature-orientation has been shown effective for bridging the semantic gap between a software system's requirements and its architecture. Over the past few years, the emergence of self-adaptive software systems, which are significantly more challenging to build than traditional systems, has gained the attention of the software engineering research community. In this paper, we show that using features at runtime could alleviate some of the key challenges of building such systems. The underlying insights are that: (1) features allow representation of the engineer's knowledge about some facets of the system that can be used to enhance the adaptation logic, and (2) features can serve as an abstraction to deal with the heterogeneity of the underlying architectural models, analytical algorithms, and implementation platforms. We describe the role of features in a self-adaptive framework that we have developed, entitled FeatUre-oriented Self-adaptatION (FUSION). We also report on our preliminary experience with FUSION that demonstrates the benefits of using features in different stages of self-adaptation.

Keywords: Self-Adaptive Systems, QoS Analysis, Feature-Oriented Modeling

1 Introduction

Feature-orientation has shown to be an effective paradigm for achieving systematic evolution and large-scale reuse in traditional software families [1-3]. In particular, it leverages feature modeling as an intuitive formalism to bridge the semantic gap between end-user requirements and software architecture.

From an end-user's perspective, a feature model decomposes the system's requirements into meaningful units of functionality, known as *features*. A feature serves as an abstraction that is independent of how the functionality is realized by the system. From a software architectural perspective, feature modeling abstracts low-level architectural variability into coarse-grained features that are easier to manage. In turn, it maximizes the reuse potential in the construction of software families. It also helps to ensure the validity of software family members, since features have a

mapping to the low-level architectural constructs and each mapping is functionally validated by the engineer.

In parallel with and largely unaffected by advances in feature-orientation and software product line research, we have witnessed the emergence of self-adaptive systems [4]. Such systems are capable of changing their behavior at runtime to achieve certain functional and QoS goals, which are often specified by the users. Building self-adaptive software systems is significantly more challenging than traditional software systems. In particular, finding the right abstractions that can bridge the gap between end-user goals on one end and their dynamic realization in the software architecture on the other end is challenging.

Given the central role feature-orientation has played in the development of traditional software systems, it is natural to believe its importance to only grow in the even more complex domain of self-adaptive systems. Features are often used during the requirements engineering phase to model the variation points in the software system. At design-time, the engineer develops a mapping for each feature to part of the underlying software architecture that realizes it. This mapping often crosscuts the different parts of the architecture [5]. We advocate an additional role for features that manifests itself at runtime. We believe features provide an appropriate abstraction for modeling the adaptation points (i.e., runtime variability) in the software system [12]. Particularly, features are used in our approach to incorporate the engineer's knowledge of some facets of the system (e.g., the semantic relationship between functional capabilities, QoS properties of concern), which augment the traditional software architectural models to mitigate the challenges of achieving self-adaptation.

In this paper, we describe the role of features in a self-adaptive framework that we have developed, entitled FeatUre-oriented Self-adaptatION (FUSION). Our preliminary experience with FUSION has shown the advantages of using features in the different stages of self-adaptation:

- Features are intuitively understood by both end-users and engineers, making them a convenient medium for eliciting adaptation preferences.
- FUSION's analysis operates on a feature-based representation of the system, decoupling it from the heterogeneity of architectural and analytical models, application domain, and implementation platform.
- Features allow FUSION to correlate results obtained from multiple analytical models to discover interactions and conflicts in the system.
- FUSION uses inter-feature relationships to reduce the configuration space significantly and make the analysis efficient.
- By encapsulating the engineer's knowledge in the mapping of features to the architecture and enforcing feature model constraints, FUSION ensures correct functioning of the system during and after the adaptation.

The rest of this paper is organized as follows. Section 2 uses a motivating example to present some of the key challenges our approach intends to resolve. Section 3 provides a high-level overview of the FUSION framework. Section 4 describes FUSION's underlying feature-oriented model. Sections 5 to 8 describe respectively how features affect the monitoring, analysis, planning, and execution activities in FUSION. Finally, the paper concludes with an overview of our future research.

2 Challenges

We illustrate the concepts in this paper using an online Travel Reservation System (TRS). Fig. 1c shows the software architecture of TRS using the traditional component and connector view. TRS aims to provide the best airline ticket prices in the market. To make a price quote for the user, TRS takes trip information from the users, and then discovers and queries the appropriate travel agent services. The travel agents reply with their price quotes, which are sorted and presented in an ascending order. In addition to the functional goals, the system is required to attain a number of QoS goals such as performance, reliability and security. To that end, solutions for each QoS perspective were developed, e.g., caching for performance, redundancy for reliability, and checkpoints for security.

A system such as TRS needs to be self-adaptive to deal with unexpected situations, such as traffic spikes or security attacks. Therefore, the self-adaptation logic of TRS needs to select from the available adaptation choices. For instance, enable caching to improve performance during a traffic spike or enable authentication to prevent a security attack. To do so, heterogeneous analytical models are required. For example, security engineers may use attack graphs [6] to prevent intrusions and find the best counter measures, while performance engineers may use queuing network models to assess the latency goals. For a complex system engineers may need to connect analytical models of multiple layers of abstraction (i.e., network, software, user, etc.) to characterize software behavior.

Therefore, applying the existing models of adaptation in the development of self-adaptive systems, such as TRS, is challenged by the following:

Challenge 1: There is no effective mechanism for identifying the interactions and conflicts among the goals in a system using the results obtained from several independent analytical models. For instance, consider the conflict between

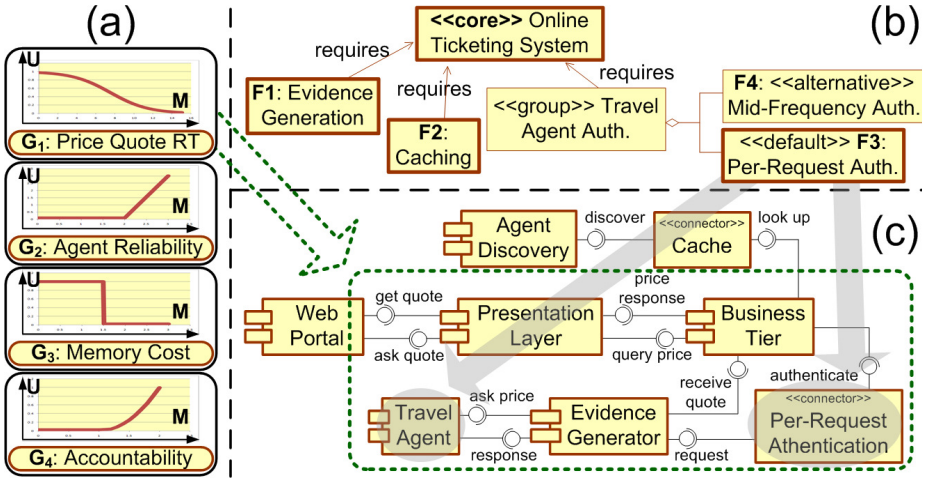


Fig. 1. Travel Reservation System: (a) goals, (b) features, thick border indicates a feature that is enabled, (c) software architecture corresponding to the enabled features.

authentication and the quality of price quotes in TRS. *Business Tier* component waits for a limited time to receive quotes from the *Travel Agent* components before timing out. Since the authentication protocol introduces an additional delay, a heavy authentication protocol may force more timeouts on the *Business Tier*, and hence reduce the number of quotes received by TRS. Building analytical models that could relate the interaction among the system's capabilities and their impact on the system's conflicting goals is often infeasible, as they require representation of complex real-world entities, such as users, networks, service providers, and so on.

Challenge 2: To satisfy multiple goals, self-adaptation logic needs to search in a configuration space that is equivalent to the combined complexity of all analytical models involved. As an example, consider how TRS would make use of N authentication components for authenticating the network traffic between its M software components, which may be deployed on P different hardware platforms. In this case, analyzing the impact of authentication alone on the system's goals would require exploring a space of $(M^P \text{ possible deployments})^{N \text{ possible ways of authentication}} = M^{NP}$ *possible configurations*. Such problem is computationally expensive to solve at runtime for any sizable system. This is while authentication is only one concern out of many in any typical system.

Challenge 3: Ensuring the correct functioning of the software system during and after the adaptation is a challenging task. This is often dependent on the application and cannot be represented effectively in the general purpose architectural modeling languages. For instance, consider the problem of representing a constraint in TRS that requires the same authentication protocol to be used on the end-to-end execution flow from the *Web Portal* all the way to the *Travel Agent* and back (depicted in Fig. 1c). Prior to switching to a new protocol, the system is required to negotiate new credentials among all of the components involved in the execution flow. The fact that this authentication protocol crosscuts multiple components is difficult to abstract and represent at the architectural level.

Challenge 4: Effecting a new architecture for a running system may require making changes at the different levels of system stack (e.g., application, middleware, and network). For instance, when a specific authentication protocol is used at the application layer, security engineers may recommend the use of certain IP services at the network layer. In addition, since the recommended IP services come with a performance hit, the engineers may prefer to leave that as an option.

These four challenges have been the prime motivation for our work. As discussed in the remainder of this paper, by adopting a feature-oriented approach, we are able to mitigate these challenges.

3 Overview of Feature-Oriented Self-Adaptation

Changes in the system or its environment trigger the process of self-adaptation. Fig. 2 depicts a high-level overview of FUSION's four main activities: *Monitor*, *Analyze*, *Plan* and *Execute*. These activities are consistent with existing self-adaptive framework's that are based on the feedback control loop reference model [4,7].

However, unlike the majority of existing approaches [8-11] that base the analysis and adaptation on the architectural models, we adopt a feature-based model of adaptation.

At runtime, these activities are performed in the following logical flow:

- *Monitor*: Collects data through instrumentation of the running system. If a functional failure or a violation of QoS objective is detected, it correlates the data into symptoms that can be analyzed.
- *Analyze*: When a problem is detected, it searches for a configuration that resolves it. It may perform a trade-off analysis between multiple conflicting goals.
- *Plan*: Chooses a path of adaptation steps towards the target configuration. The path has to abide by the system constraints. In addition, adaptation steps must not cause further failures in the system.
- *Execute*: Takes the required actions to effect the changes delineated in the plan. This may require adding, removing, and replacing the components and the way they are interconnected in the running architecture.

In the remainder of this paper we describe how using feature-orientation affects and improves the behavior of these activities. Each activity addresses one of challenges introduced in Section 2. The *Feature Based Models*, shown in the middle of Fig. 2, is how the engineer's knowledge of the system's characteristics and its domain is captured and provided for the activities.

4 Feature-Based Models

A feature is an abstraction of a capability provided by the system. A feature may affect either the system's functional (e.g., ticket discounts) or non-functional (e.g., authentication protocol) properties.

Conceptually, features elicited for runtime variability serve a different purpose than traditional ones. The main motivation behind a runtime feature is to account for variability in the system's execution context rather than the end-user requirements. That is, to give the system enough flexibility to cope with an environment where no one solution works perfectly at all times. The goal is to identify critical features required for the system given such variability in the context.

The proposed features are in essence variation points in the architecture rather than requirements. Exposing them as features makes them independent of a particular implementation platform or application domain. For example, in a rule-based system a feature may correspond to a set of rules, in a service-oriented system it may correspond to a set of services in a workflow, in an adaptive system it may correspond to a set

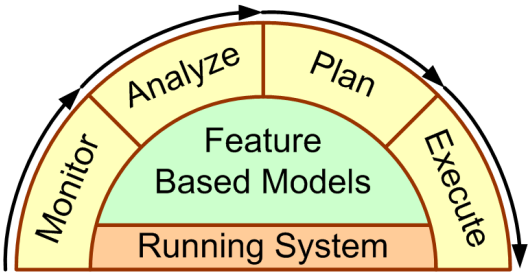


Fig. 2. High-level overview of FUSION.

of adaptation strategies, and so forth. Fig. 1b shows a particular realization of features: a feature is an abstract representation of an architectural variant. As depicted in Fig. 1b, features map to a subset of the system's software architecture. In other words, features crosscut the system's software architecture.

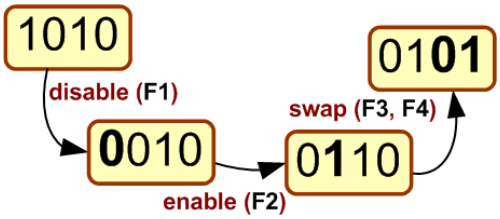


Fig. 3. Feature-based adaptation.

4.1 Runtime Variability

Fig. 1b shows a simple feature model for TRS. There are four features in the system and one common core. The features in the example use two kinds of relationships: dependency, and mutual exclusion. The dependency relationship indicates that a feature requires the presence of another feature. For example, enabling the *Evidence Generation* feature requires having the *Core* feature enabled as well. Mutual exclusion is another relationship, which implies that if one of the features in a mutual group is enabled, the others must be disabled. For example, *Per-Request Authentication* and *Mid-Frequency Authentication* cannot be enabled at the same time as they belong to the same mutual group. Feature modeling supports several other types of inter-feature relationships [1] that we do not discuss for brevity.

In FUSION, at runtime we use the feature model to identify the current system configuration in terms of a feature-selection string. In a feature-selection string, enabled features are set to "1"; disabled features are set to "0". For example, one possible configuration of TRS would be "1101", which means that all features from Fig. 1b are enabled except *Per-Request Authentication* (i.e., F_3).

The adaptation of a system in FUSION is modeled as a transition from one feature-selection string to another (more details in section 7). Each transition takes one of the three forms: enable and disable an optional feature, or swap two mutually exclusive features. Fig. 3 shows three transitions that take the TRS system from feature selection "1010" to "0101".

4.2 Goals

In FUSION, system failure is defined as inability to satisfy one or more system goals. We have adopted a simple, yet very expressive, approach for modeling the system's goals. A goal has a *utility* function for which a system quality *metric* can be optimized. The metric is a measurable quantity (e.g., response time) that can be obtained from a running system. The *utility* function expresses the engineer's preferences for the *metric*. For instance, G_1 (*Price Quote Response Time*) in Fig. 1a specifies a response time metric value to be collected from sensors in the system. The corresponding utility function specifies the user's preferences for different values of price quote response time.

FUSION calculates the system's expected utility for a new feature selection F' for the system as follows:

$$\sum_{g \in G} U_g\left(M_g(F')\right)$$

, where U returns the utility associated with achieving a given metric M of goal g.

A utility function can be used to express hard constraints. In that case the utility function would be a step-function such as the utility of G_4 depicted in Fig. 1a. A utility function may take on more advanced forms (e.g., sigmoid curve), and express more complex preferences, such as G_1, G_2, and G_3.

FUSION places one constrain on the specification of utility functions: they need to return zero for the range of metric values that are not acceptable to the user. When a utility associated with a goal reaches zero, FUSION considers that goal to be violated and initiates adaptation

5 Monitor

As mentioned in challenge 1 of Section 2, quantifying the impact of adaptation choices on the system's conflicting goals are typically difficult (e.g., recall the trade-off between the authentication protocol and the quality of price quotes). We believe this difficulty is partially due to the gap between the system's goals and the low-level units of adaptation (e.g., add/remove component) at the architecture-level. In other words, the adaptation occurs through low-level architectural changes, while the goals are high-level concerns. Achieving a particular goal may require a series of low-level changes at the architecture level. As a result, identifying the impact of low-level changes on the system's goals becomes extremely difficult.

In FUSION, the units of adaptation are features, which are inherently less granular than low-level architectural constructs. In turn, since *Monitor* collects the data at a higher level (i.e., feature level), it is significantly easier to observe and identify the conflicts among goals. In particular, the monitored data in FUSION can be used to determine two kinds of interactions:

1. *Goal interactions* with respect to one feature. A goal interaction occurs when two goals are affected by enabling a feature. For instance, F_1 (*Evidence Generation*) has a positive effect on G_4 (*Accountability*) and negative effect on G_1 (*Price Quote Response Time*), since *Evidence Generation* adds a mediator component to witness the exchange of messages between TRS and travel agents.
2. *Feature interactions* with respect to one goal. A feature interaction occurs when enabling two features modifies the behavior of one or both features. For example, enabling both features F_1 (*Evidence Generation*) and F_3 (*Per-Request Authentication*) has a negative ramification on G_1 (*Price Quote Response Time*) that is beyond the individual impact of each. *Per-Request Authentication* changes the behavior of *Evidence Generation,* since it causes additional overhead in mediating exchange of authentication credentials between TRS and travel agents.

6 Analyze

Analyze conducts runtime analysis to find a configuration of the system that resolves the violated goals. As mentioned in challenge 2 of Section 2, performing such analysis at the architectural-level is often computationally very expensive for any sizable system. FUSION uses features to encode the engineer's knowledge of the adaptation choices that are practical. In turn, *Analyze* operates on the feature selection space, which is significantly smaller than the architecture selection space.

For instance, in the TRS example, the engineer has exposed only the authentication strategies that are foreseen to be useful as features. Fig. 1b shows the two authentication strategies that are modeled as features in the TRS: F_3 and F_4. This automatically reduces the configuration space from M^{NP} (recall example of challenge 2) to 2^F, where F is the number of variant features that affect the authentication concern in the system. Clearly it is reasonable to assume that $M >> 2$ and $N \times P >> F$ for any sizable system.

In addition, using the inter-feature relationships (e.g., mutual exclusions, dependencies) we can further reduce the feature selection space. For instance, Fig. 1b shows a mutual exclusive relationship between F_3 and F_4. This relationship captures the engineer's application knowledge that applying two authentication protocols to the same execution scenario is not a valid configuration. Such relationships reduce the space of valid feature selections significantly.

We can further scope down the analysis to only the features that affect the violated goals. *Analyze* first finds features that have a significant impact (positive or negative) on the violated goal. It then finds any other goals that are affected by the selected features. As a result, FUSION's feature-based analysis is significantly more efficient than the alternative of assessing all of the system goals for the entire space of adaptation choices at the architectural level.

7 Plan

As you may recall from challenge 3 in Section 2, adaptation planning is a major source of difficulty, due to its application dependent nature. This is one of the key shortcomings of existing self-adaptation frameworks, which either ignore or revert to ad-hoc techniques during the planning stage. In FUSION, the engineer models this knowledge in terms of features and their dependencies. This is used to devise a plan that ensures the system's correct functioning during and after the adaptation.

Fig. 3 shows an *adaptation plan* in FUSION, which consists of a series of transitions from the current feature selection to a new one. Since many paths can be traversed to reach a target feature selection, *Plan* uses the feature model to pick a path that abides by feature model constrains in every intermediate step. In TRS for example, enabling F_3 and F_4 at the same time produces a feature selection that violates the mutual exclusion relationship in the feature model. If two features are mutually exclusive, the system should never be in a state were both features are enabled. Similarly, a dependent feature should not be enabled without its prerequisite. In other

words, the path should not cause transition to an invalid feature selection that could jeopardize the system's functionality.

In addition, guided by utility functions, *Plan* can pick a path that minimizes violation of goals as much as possible. For instance, suppose that enabling F_1 causes 5% decrease in the utility of G_1. If G_1 is already 1% away from violating its constraint, enabling F_1 right away will cause a violation. In such a case, the adaptation plan first enables another feature, suppose F_2, to increase G_1's utility (e.g. up to more than 5% away from the constraint) before enabling F_1.

8 Execute

Execute carries out the process of changing the system's configuration. However, as mentioned in challenge 4 of Section 2, effecting a new architecture may require making changes at different levels of system stack (e.g., application, middleware, network). FUSION uses features as platform-independent effectors. Each feature is associated with a *feature mapping*, which relates the feature to a part of the running system. A feature mapping is a set of rules that specify the changes that need to take place in the lower levels of system stack. For mutual exclusive features, one mapping is created for each mutual group.

Fig. 4 shows how FUSION integrates with the system using a feature mapping interface. In part (a), the feature mapping interacts with multiple platforms at the application level. In part (b), the feature mapping rules extend to different levels of system stack. In both cases, the role of *Execute* is limited to invoking one feature-mapping interface at a time (i.e., enable/disable/swap a feature) regardless of *how* and *where* changes are taking place. For example, enabling a feature may correspond to delopying new components in the application, selecting a new resource allocation policy in the middleware, switching off certain network interfaces, and so on. The feature mapping interface invokes effectors in the running system to apply the changes as specified.

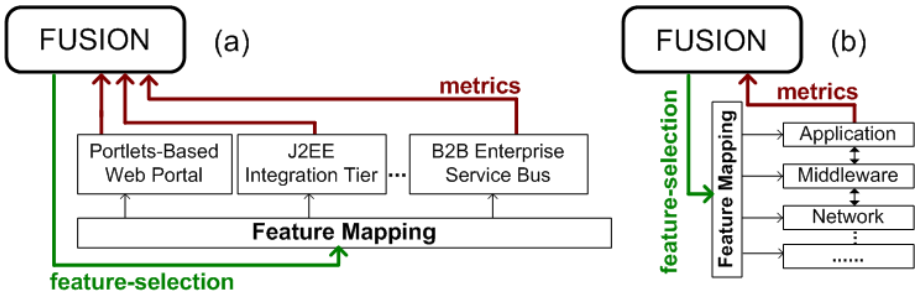


Fig. 4. FUSION uses feature-mapping to integrate with (a) heterogeneous implementation platforms, and (b) different levels of system stack.

9 Conclusion

We described the role of features in a self-adaptive framework, called FUSION. We showed how feature modeling alleviates some of the key challenges of building self-adaptive systems. The underlying insight guiding our research is that: (1) by using features to incorporate the engineer's knowledge of some aspects of the system we can enhance the adaptation logic, and (2) features can serve as an abstraction to deal with the heterogeneity of the underlying architectural models, analytical algorithms, and implementation platforms. As part of our future work we intend to empirically evaluate and compare the FUSION framework against other self-adaptation frameworks. In particular we plan to quantitatively assess the benefits and drawbacks of using feature abstractions for self-adaptation in the context of real-world applications.

Acknowledgments. This work is partially funded by contract W9132V-07-C-0006 with US Army Geospatial Center, as well as grant CCF-0820060 from National Science Foundation. We would like to thank Mark Pullen for his guidance and support in this research.

References

1. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures, Addison-Wesley Professional, 2004.
2. Kang, K.C., et al.: Feature-oriented domain analysis (FODA) feasibility study. Carnegie-Mellon University, Pittsburgh, PA, Software Engineering Institute, 1990.
3. Kang, K.C., et al.: FORM: A feature-oriented reuse method with domain-specific reference architectures. In: Annals of Software Engineering, vol. 5, 1998, pp. 143–168.
4. Cheng, B. H. C. et al., Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Software Engineering for Self-Adaptive Systems, Lecture Notes on Computer Science Hot Topics, 2009, pp. 1-26.
5. Lee, K., Kang, K., Kim, M., Park, S.: Combining feature-oriented analysis and aspect-oriented programming for product line asset development. In: 10th International Software Product Line Conference, 2006, pp. 10 pp.-112.
6. Foo, B., Wu, Y., Mao, Y., Bagchi, S., Spafford, E.: ADEPTS: adaptive intrusion response using attack graphs in an e-commerce environment. In: Dependable Systems and Networks, 2005. DSN 2005. Proceedings International Conference on, 2005, pp. 508-517.
7. Andersson, J., de Lemos, R., Malek., S., and Weyns, D.: Modeling Dimensions of Self-Adaptive Software Systems. In: Software Engineering for Self-Adaptive Systems, Lecture Notes on Computer Science Hot Topics, 2009.
8. Garlan, D., Cheng, S., Huang, A., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, Oct. 2004.
9. Oreizy, P., et al.: An Architecture-Based Approach to Self-Adaptive Software. In: IEEE Intelligent Systems, vol. 14, 1999, pp. 54--62.
10. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: International Conference on Software Engineering, 2007, pp. 259-268.
11. Malek, S., Mikic-Rakic, M., Medvidovic, N.: An extensible framework for autonomic analysis and improvement of distributed deployment architectures. In: Proceedings of the workshop on Self-managed systems, ACM New York, NY, USA, 2004, pp. 95-99.
12. Lee, J., Kang, K.: "A feature-oriented approach to developing dynamically reconfigurable products in product line engineering," Software Product Line Conference, 2006 10th International, 2006, pp. 10 pp.-140.

Models at Runtime: Service for Device Composition and Adaptation

Nicolas Ferry[1,2], Vincent Hourdin[1,3], Stéphane Lavirotte[1], Gaëtan Rey[1], Jean-Yves Tigli[1,†], Michel Riveill[1] ⋆
{ferry, hourdin, lavirott, rey, tigli, riveill}@polytech.unice.fr

[1] Laboratoire I3S (Université de Nice - Sophia Antipolis / CNRS) 930 route des Colles - B.P. 145 06903 Sophia-Antipolis Cedex - France
[2] CSTB, 290, route des Lucioles, B.P. 209 06904 Sophia-Antipolis Cedex - France
[3] MobileGov, 2000, route des Lucioles - 06901 Sophia Antipolis - France
[†] currently delegated as INRIA researcher in the team PULSAR

Abstract. Our works on software architectures for highly dynamic environments, such as ubiquitous computing, led us to consider models at runtime. Indeed, the biggest challenge of these environments is adaptation, and its reactivity is a key concern. In this paper, we describe models of our dynamic service composition and adaptation approach, and the benefits of the use of metamodels. We explore how metamodels can be used at runtime, to enforce conformity of transformations results, when adaptation is seen as model transformation.

1 Introduction

Ubiquitous computing, as described by Mark Weiser [1], relies on computers present everywhere, at any times and in any things. Indeed with recent years advance in mobile communication technologies and the miniaturization of computer hardware, processing units are becoming invisible and a part of the environment. Software infrastructure appears dynamically populated by functionalities of those devices. The topology of this infrastructure is also dynamic, due to arbitrary node mobility. So that the software infrastructure of an ubiquitous application is not known *a priori*.

One of the biggest challenges of highly dynamic environments, like in ubiquitous or pervasive computing, is to handle the modifications of the infrastructure at runtime. Indeed, these systems have to adapt continuously to their environment [2]. An important constraint is that the adaptation process is driven by the environment and not by the application. Reactivity is a major concern, compared to other systems: the adaptation process must quickly be launched, in reaction to changes in the environment, and must finish before new changes happen.

However, before we can create an "universal application" able to alter its behavior and functionalities in reaction to changes of the environment, a first challenge is to maintain its predefined functionalities despite those variations,

⋆ This work is part of the Continuum Project (French research) ANR-08-VERS-005

without knowing what kind of device are going to be discovered. A continuity of service has to be ensured to mobile users in environments with variable dynamics and heterogeneous resources.

Such adaptation is equivalent to a program transformation which is a kind of model transformation [3]. But in the field of ubiquitous computing it must be a highly reactive transformation, forcing model-checking to happen at runtime, as we will study in the following section (Sect. 2).

In Sect. 3, we present our runtime approach as well as involved models and metamodels. Furthermore, we introduce our adaptation mechanism as a model transformation from an application to another. Then, in Sect. 4, we explain how adaptation is decomposed into three transformations between different models of the approach. Finally, (Sect. 5), we explain how metamodels could help us to check the validity of the adaptation process and the executing application.

2 Reactivity

Reactivity is a key concern of ubiquitous computing, both for adaptation triggering and for adaptation time. We consider that adaptive applications are always in one of the three states presented in Fig. 1. States (1) and (3) are normal execution states of the application, where it is consistent with its environment. It means that the application's behavior is based on what is relevant in its environment and this is the expected behavior for a particular situation. During the transitional state (2) the application is in its adaptation (transformation) phase and unavailable. It is considered in an inconsistent state because the application is not in line with its environment. Moreover, the time spent in this state, related to the speed of the adaptation process, has to be consistent with the dynamic of the changing environment. In other words, it is essential that:

- the system does not stay in the previous state (1) too much time before reacting to environment changes,
- the system is not unavailable for too long while adapting,
- adaptation is fast in order to obtain an application consistent with the environment. Otherwise, the system could become unstable and may never reach a normal execution state before new evolutions occurs in its environment.

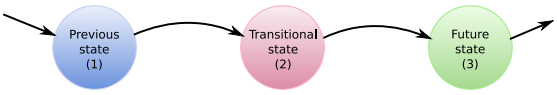


Fig. 1. The three states of an adaptive application

The complexity of the adaptation mechanism must be as low as possible and based on mechanisms that do not try to consider the whole environment (which

can be assimilated to the world) but only what is relevant to the application. The surroundings of the application should be sensed to a sufficient degree to achieve the necessary adaptation.

In some cases, adaptive systems must offer bounded response times. To get an application from one state to another, adaptation must also be consistent with the inherent dynamic of the application. Various studies have suggested response times supposed adequate to users on various systems. In the field of HCI, Berard [4] proposed that the latency of a device must be at least two times less than the user latency which is 100ms. In the area of domotics we consider that an acceptable lag is about 1s. For robotics or ubiquitous systems, latency depends on a cycle based on three steps: perception, processing, action. For the cycle, we consider that an acceptable lag is about 100ms. Thus, adaptation has to be triggered briefly, as much as it has to be effective quickly.

In the next section, we propose an approach for self-adaptive application in the field of ubiquitous computing based on models, in which adaptation is a reactive mechanism.

3 Our Runtime Approach

Our vision of ubiquitous systems rely on three layers: the environment, accessible through the software infrastructure, the application, as a composition of service and the adaptation mechanism. In this section, we present these layers' models, with their associated meta-models.

3.1 Services for Devices: a Software Infrastructure

For many years, service oriented architectures (SOA) have been used in home automation, mobile and ubiquitous computing to represent as services the sets of functionalities offered by devices. They offer lots of features discussed in [5] such as encapsulation, dynamicity, discoverability and interoperability. They evolved from standard SOA to SOA for device (SOAD) by adding two main features: *decentralized reactive discovery* and *asynchronous communications*.

Decentralized reactive discovery has been popularized by projects such as SLP [1] or Jini. It suppresses the need of a service registry tracking active services in a network domain. Services advertise their presence and clients create search requests using multicasted or broadcasted messages. Asynchronous communications used by SOAD like Jini are events notifications, providing reactivity to devices often interacting with humans or the environment.

In addition, when Web technologies are used to implement SOAD, interoperability between all entities is enabled, whether they are heterogeneous devices or simple software services. Only two implementations of Web services for devices currently exist: UPnP[2] and DPWS[3]. Figure 2 represents a model of a UPnP

[1] The Service Location Protocol.
[2] Universal Plug and Play Forum: http://www.upnp.org/
[3] Device Profile for Web Services. http://www.ws4d.org/

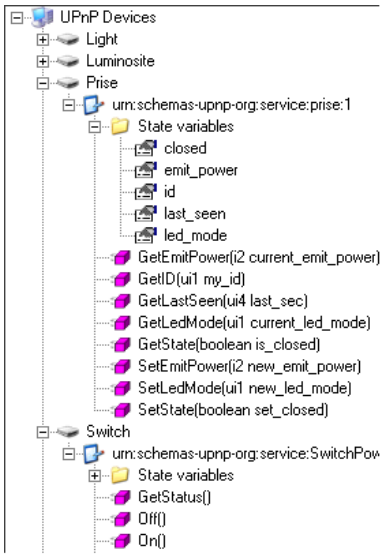


Fig. 2. An infrastructure model

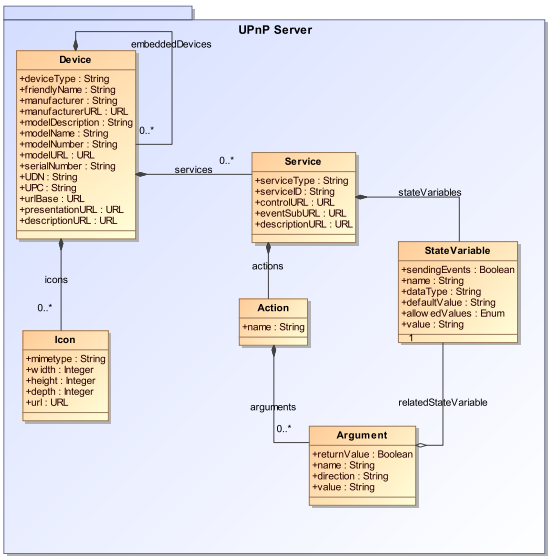


Fig. 3. Infrastructure Metamodel

architecture (what we use as software infrastructure), with four devices, and some details about the services they offer. Next to it, Fig. 3 represents the metamodel of the UPnP the infrastructure. All environmental quantity that can be used in the application is obtained through the use of services present in the infrastructure and conform to this metamodel.

Evolutions of WSOAD allow to create reactive dynamic distributed applications, suitable for ubiquitous computing environments. Using services for devices, any modification occurring in the software infrastructure can be integrated at runtime into the model of the software infrastructure. Thus, the application can react quickly to those unpredictive variations.

3.2 Dynamic Service Composition

To create applications from this infrastructure of services for devices, we use the Service Lightweight Component Architecture (SLCA) [6]. It allows to dynamically orchestrate and compose services for devices using lightweight component assemblies executing in containers. The container provides minimal technical services, also known as non-functional concerns helpers. Obviously, we created external tools that can generate client components from Web services for devices descriptions. We call them proxy components.

Containers manage assemblies of components fully dynamically and we use models such as Fig. 4 to manage them at runtime. Component types can be loaded and unloaded, component instances and bindings between them can be added or removed at runtime. Proxy components are generated, loaded and instantiated dynamically and automatically, following the presence of services of

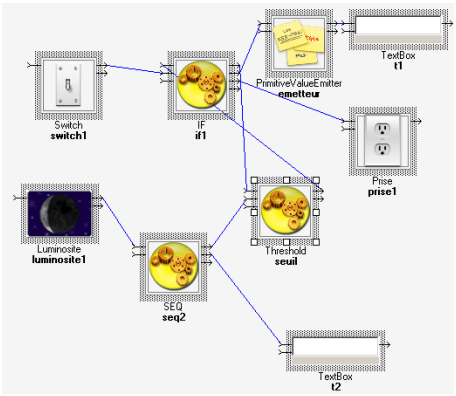


Fig. 4. A SLCA model

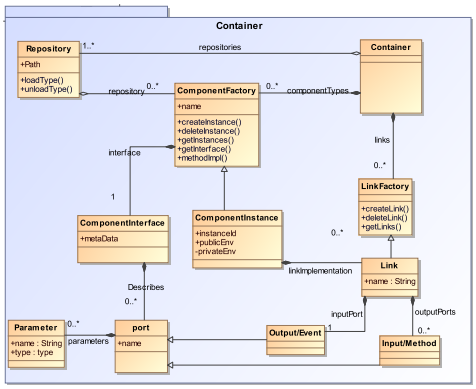


Fig. 5. SLCA Metamodel

the infrastructure. There is a direct mapping between the infrastructure metamodel and the application (SLCA) metamodel. Moreover, this mapping is highly reactive.

Applications or new functionalities are created from existing services on the infrastructure by managing an assembly of components. Proxy components are combined together or with purely functional components to transform information. SLCA components and services for devices communicate mostly using event-based communication patterns, which, more than decoupling entities and increasing dynamicity, enables the reaction to infrastructure changes efficiently.

Figure 5 represents the metamodel of this composition architecture, for a container and its internal dynamic assembly of components.

3.3 Auto-adaptation

Now that applications are created from the dynamic infrastructure, we need to adapt their behavior to changes of their environment in a reactive way. We created a paradigm called Aspect of Assembly [5] that allows us to adapt composite services according to specified rules. Aspects of assembly (AA) are pieces of information describing how an assembly of components will be structurally modified, thus keeping black-box property of components. Modifications include adding components and bindings between them. Since the mapping between the infrastructure and the application is done dynamically, an expert user, in order to build an application, has to write and select a set of AAs. Aspects of Assembly consist of two parts, like regular aspects found in Aspect-Oriented Programming (AOP) [7]: pointcut and advice. Pointcuts describe to which components the modifications described by advices have to be *weaved* (applied).

If some of the required components expressed in a pointcut are not available, the advice won't be weaved until they become all available. Since service discovery is a reactive process and that containers notifications are events too,

AAs can be weaved in response to the appearance of a service on the infrastructure. Moreover, AA composition provides associativity, commutativity and idempotence properties when several aspects are enabled to be weaved at the same time [5].

Pointcut are defined as sets of filters on base assembly meta data — for example component ID or types (see Fig. 7). Those filters construct lists of parameters, called the joinpoints, satisfying the list of variables of the associated advice. They are the set of components on which the advice will be weaved. For each generated list, the advice is duplicated, and the variables are syntactically replaced in the advice to match the base assembly joinpoints. For our experiments, we choose for convenience to express filters in using some simple pattern matching on component instances names.

Advice in AA is not a piece of code which will be weaved into components. It defines a set of component instances and links that will be weaved inside a targeted assembly of components. Advices are specified in a DSL using interaction specification defined in [5]. A link or connector between two components consists of an input event and an output method from components. More than links, rules can create assemblies thanks to some predefined operators which are components with a well-known semantic like a condition (`if`) or indeterminism (`parallel`) (see Fig. 7). Finally all those specifications are translated into a set of elementary modifications—add or remove components and links—with respect to blackbox properties of COTS components.

As we have seen, AAs are written as a pointcut and an advice. They can be written and added to the system at design-time or at runtime. Checking that an AA is conform to its metamodel (Fig. 7) is easy. However, at runtime, new informations are available, like joinpoints and the status of the application, making the checking more complicated and more relevant for a situation.

In the following section we will present how AAs are weaved and how this can be compared to a highly reactive model transformation.

4 Aspects of Assembly Weaving Process

The weaving process can be seen as an horizontal endogenous model-to-model transformation [8]. It re-factors an SLCA application into another.

With this approach, the transformation is done at runtime. While the adaptation is being done, the application is in transitional state, but can still be used. Only the adaptation mechanism is busy during this lapse of time. This is a major concern for service continuity and for application's reactivity in ubiquitous computing. Moreover, in potentially very large systems, using the AOP approach allows designers to write adaptation rules focusing only on a part of the system, reducing the complexity of the adaptation process. Since aspects can

Fig. 6. Aspects of Assembly model

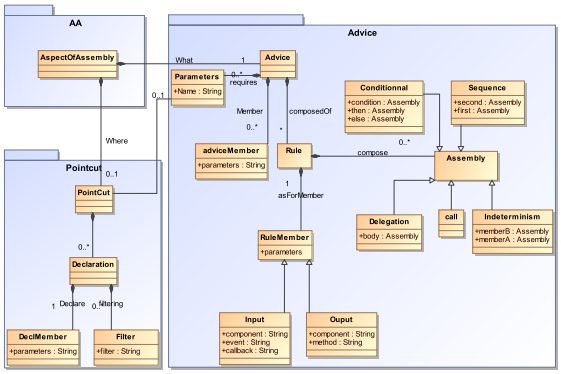


Fig. 7. Aspects of Assembly Metamodel

be duplicated in several places, the number of aspects to consider may be lower than in other approach. The complexity of the adaptation mechanism is reduced to provide a better reactivity.

Other works have explored the use of models at runtime for aspect-based adaptation [9,10]. They need the dynamicity to validate new adaptation rules at runtime, before making them automatic scripts. But because our mechanism is build on less steps (especially model transformations) we seem more able to be reactive. In fact there are only few works on highly reactive mechanism for adaptation in ubiquitous systems. OpenORB [11] proposes a way to adapt a system at runtime in a very reactive way but it does not consider the whole classical mechanism for context-awareness. CORTEX [12] also addresses the concern of reactivity but as OpenORB, it does not manage potential conflicts between several adaptations.

4.1 The Weaving Process

In this section we study more precisely the weaving process that can be decomposed into three steps (Fig. 8). First, pointcut matching is a function that has a set of components from the initial assembly and pointcuts from a set of selected AA as input. Its goal is to find the components, called *joinpoints*, on which advices will be woven. Based on pointcut matching results, an advice can be weaved several times in the same weaving process.

The second step is called the advice factory. It generates instances of advices, replacing variable components in advices of selected aspects by joinpoints given by the first step. Instances of advices describe modifications to be weaved in an actual assembly of components.

Finally, the composition engine merges all instances of advices with the initial assembly in order to generate a single instance of advice that will be weaved as the final assembly. This merging mechanism is able to resolve conflicts between

various instance of advices [5]. Thus we can build applications by composing several aspects of assembly at the same time, at runtime.

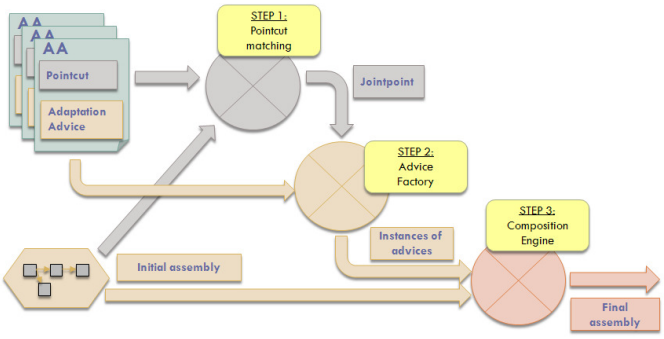


Fig. 8. Detailed weaving process

In order to be more reactive, the weaver can be triggered in two ways. The first is user-driven and changes the set of AAs given as input to the weaver, by selecting/deselecting or adding/removing aspects of assembly at runtime. When the set of AAs is modified, the weaver is triggered, leading to adaptation if an added AA can be applied or if a removed AA was. The second way of triggering adaptation is driven by infrastructure changes. When a new device appears or disappears in the environment, its proxy component is dynamically instantiated in or removed from the assembly. The adaptation process is triggered and only AAs that are able to apply according to newly available components are applied.

An important point for the reactivity of such a mechanism is that the system doesn't requires any information about the state of the software infrastructure. With a dynamic of its own, the infrastructure imposes his pace. The inclusion of this type of triggering mechanism allows us to build opportunistically from an infrastructure services and devices, applications with a "bottom-up" approach. Applications are built in line with their infrastructure in a reactive way.

4.2 Synthesis

Our approach allows us to build applications able to adapt at runtime to their software infrastructure. Moreover, coupling the event paradigm and AOP allows us to transform our applications and our models in a very reactive way. Our approach is decentralized and distributed since each weaver is associated to a composite service and deals with AAs in order to consider only relevant part of the environment. This caused a sharp reduction of adaptation's complexity.

We validated our approach with some experiments on the cost in time of a weaving process on randomly generated component assemblies. Some of those results can be found in [13]. As result of these experiments, we found that with two AAs conflicting we are able to weave about 40 components (e.g to merge

20 instance of advice) in 100ms. Without conflicts we are able to compose more than 100 components (e.g. to compose 50 instance of advice) in 100ms.

5 The Contribution of Metamodels

More than a reactive model-to-model endogenous transformation [8] at runtime, our adaptation process can be seen as a parametrized transformation. All selected and applicable AAs are parameters that define the transformation taking place. Without these parameters, the transformation does not involve any modification of the initial assembly. In fact, this transformation is decomposed into three low cost transformations (Fig. 9) to be done at runtime.

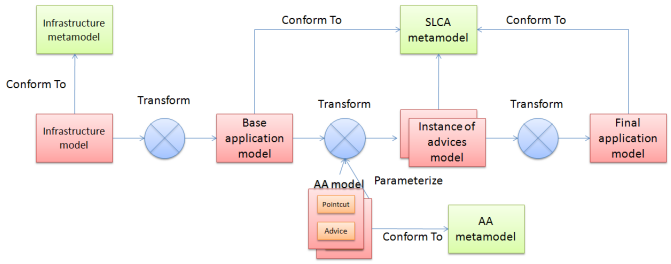


Fig. 9. The adaptation process as three transformations

1. The first transforms the software infrastructure model, conform to the infrastructure metamodel, into a component assembly conform to the SLCA metamodel. This is an horizontal model-to-model and one-to-one exogenous transformation.
2. The second transforms the application obtained in 1. into instances of advices which are the sub-assemblies to weave in the final application. These instances are also conforming to the SLCA metamodel. This transformation is parametrized by a set of selected AAs conform to the AA metamodel. This is an horizontal one-to-many parametrized endogenous transformation.
3. Finally, the last transformation merges all the instances of advices into a single application also conform to the SLCA metamodel. This is a many-to-one horizontal endogenous model transformation.

As we have seen, our mechanism for adaptation in ubiquitous environment is a set of highly reactives models transformation at runtime. The introduction of metamodels defining the three levels of our approach has allowed us in the first place to abstract the work already undertaken and to ease its reusability. Secondly, the ANR RNTL FAROS project has enabled us to validate our metamodels and to check models conformity at design-time. We are now working on setting up mechanisms to check the conformity at runtime of our applications through those metamodels.

6 Conclusion

We presented our approach for dynamic service composition and adaptation based on aspects of assembly, that is equivalent to a highly reactive model transformation at runtime. Models allow us to check the conformity of applications, infrastructure, and adaptation parameters at design-time. Incidentally, what would be the impact of such checks at runtime on the reactivity of our approach? Moreover, how can we benefit from our three-step transformation to check partial conformity of models during the whole adaptation process? Finally, we will explore runtime aspect-oriented modelling (AOM) and its reactivity.

References

1. Weiser, M.: The computer for the twenty-first century. Scientific American **265**(3) (Sep 1991) 94–104
2. Berry, D., Cheng, B., Zhang, J.: The four levels of requirements engineering for and in dynamic adaptive systems. In: 11th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ). (2005)
3. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. **45**(3) (2006) 621–645
4. Crowley, J., Coutaz, J., Bérard, F.: Perceptual user interfaces: things that see. Communications of the ACM **43**(3) (2000)
5. Tigli, J.Y., Lavirotte, S., Rey, G., Hourdin, V., Cheung-Foo-Wo, D., Callegari, E., Riveill, M.: WComp Middleware for Ubiquitous Computing: Aspects and Composite Event-based Web Services. Annals of Telecommunications (AoT) **64**(3–4) (Apr 2009) 197–214
6. Hourdin, V., Tigli, J.Y., Lavirotte, S., Rey, G., Riveill, M.: SLCA, composite services for ubiquitous computing. In: Proceedings of the 5th International Mobility Conference, Singapore Chapter of ACM (2008)
7. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., marc Loingtier, J., Irwin, J.: Aspect-oriented programming. In: ECOOP, SpringerVerlag (1997)
8. Mens, T., Czarnecki, K., Gorp, P.V.: A taxonomy of model transformations, Germany, Internationales Begegnungs und Forschungszentrum für Informatik (2005)
9. Fleurey, F., Dehlen, V., Bencomo, N., Morin, B., Jézéquel, J.: Modeling and Validating Dynamic Adaptation. In: 3rd International Workshop on Models@ Runtime (MODELS.08), France, Springer (2008)
10. Morin, B., Fleurey, F., Bencomo, N., Jezequel, J., Solberg, A., Dehlen, V., Blair, G.: An aspect-oriented and model-driven approach for managing dynamic variability. In: 11th International Conference on Model Driven Engineering Languages and Systems (MODELS), Springer (2008)
11. Grace, P., Coulson, G., Blair, G., Porter, B.: A distributed architecture meta-model for self-managed middleware. In: Proceedings of the 5th workshop on Adaptive and reflective middleware (ARM'06), ACM (2006)
12. Verissimo, P., Cahill, V., Casimiro, A., Cheverst, K., Friday, A., Kaiser, J.: Cortex: Towards supporting autonomous and cooperating sentient entities. In: Proceedings of European Wireless. (2002) 595–601
13. Ferry, N., Lavirotte, S., Tigli, J.Y., Rey, G., Riveill, M.: Context Adaptative Systems based on Horizontal Architecture for Ubiquitous Computing. In: International Mobility Conference, France, ACM (September 2009)

A Model-Driven Configuration Management System for Advanced IT Service Management

Holger Giese, Andreas Seibel, and Thomas Vogel

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

Abstract. A popular guideline to manage today's complex and heterogeneous IT systems is the IT Infrastructure Library (ITIL), which provides a catalogue of best practices for IT Service Management (ITSM). However, state-of-the-art implementations of ITIL rely on a set of XML-based standards. To ease manageability and effectively exploit a Configuration Management System (CMS), which is the integral part of ITSM, we suggest in this paper a model-driven CMS by applying Model-Driven Engineering (MDE). Metamodel based models improve the manageability by providing a suitable abstraction, which enables direct user interaction as well as the application of MDE techniques such as model transformations. Furthermore, vital elements of a model-driven CMS are runtime models, which capture the managed system. In addition, this paper reports on a first prototype implementation of a model-driven CMS that exploits runtime models, their automatic maintenance, model-based analysis on these runtime models, and automatic adaptation of the managed system by facilitating changes on runtime models.

1 Introduction

A recent observation is the increase of administration costs due to the increasing complexity and heterogeneity of IT systems whereas these IT systems still need to be manageable. At the same time, IT systems need to be delivered even at a higher speed and managed at minimum costs [1], which forces IT system providers managing them efficiently. A popular guideline to manage today's complex and heterogeneous IT systems is the *IT Infrastructure Library* (ITIL) v3 that provides a catalogue of best practices for *IT Service Management* (ITSM) containing common definitions by using a common terminology. The integral part of ITSM is the *Configuration Management System* (CMS), which is primarily a toolset and storage for *Configuration Items* (CI). CIs are manageable elements of the managed system, which can be services, incidents, problems, hardware, software, buildings, persons, etc. In conclusion, a CMS supports management which leads to increasing quality and a more economic management of IT systems.

Several commercial ITIL implementations exist, e.g., IBM Service Management [2–5], which can be seen as state-of-the-art in ITSM. These approaches are quite powerful and comprehensive. Nevertheless, they rely on a set of XML-based standards and do not leverage the full strength of *Model-Driven Engineering* (MDE). On the other side, research approaches do not focus on ITSM as

proposed by ITIL [6–11]. These approaches embrace aspects of runtime models and autonomic computing in different domains to manage systems autonomically. However, we think that autonomic computing in ITSM is currently only partially feasible, e.g., in parts of *Service Operation* to keep the system running [12]. Nevertheless, it is an important vision for ITSM.

In this paper we fill this gap by focusing on easing manageability and effectively exploiting a CMS by applying MDE. Meta model based models improve the manageability by providing a suitable abstraction, which enables direct user interaction as well as the application of MDE techniques such as model transformations. Furthermore, vital elements of a model-driven CMS are runtime models, which capture the managed system. In addition, this paper reports on a first prototype implementation of a model-driven CMS that exploits runtime models, their automatic maintenance, model-based analysis on these runtime models, and automatic adaptation of the managed system by facilitating changes on runtime models. However, we cannot provide a complete ITSM approach that performs as competitor for state-of-the-art approaches. Thus, our prototype implementation focuses on parts of *Change Management*, *Release & Deployment Management* and *Service Asset & Configuration Management* although the proposed CMS is open to extensions for other management processes.

The paper is structured as follows: In Section 2 we outline a model-driven CMS for ITSM by applying MDE, which conforms to ITIL. We show a prototype implementation facilitating runtime models in Section 3. An application example is shown in Section 4 and we close the paper with conclusions and future work in Section 5.

2 Model-Driven Configuration Management System

In this section, we first outline a common CMS extracted from ITIL and state-of-the-art implementations (cf. [2–5]). Based on these insights, we propose applications of MDE. This implies the application of runtime models, management models and MDE techniques to automate several processes.

2.1 Common Structure of a Configuration Management System

Additionally to CIs, a CMS contains logical dependencies between CIs that have to be captured as well as information that is required by management processes, such as *Change Management* or *Release & Deployment Management*. Managing CIs in the CMS is the task of *Service Asset & Configuration Management*. Consequently, it is tightly coupled to the CMS. Figure 1 shows a CMS within the context of ITSM. A CMS consists of a federated *Configuration Management Database* (CMDB) and a set of *Managed Data Repositories* (MDRs), which are technically CMDBs. An MDR focuses on a specific domain of the managed system, e.g., database servers or certain applications and thus contains detailed information about CIs in that domain. MDRs gather information about CIs from different sources within the managed system, e.g., through management interfaces. The federated CMDB is responsible for providing all relevant CIs of the managed system and their logical dependencies at a higher level of abstraction.

Each MDR federates its CIs to the federated CMDB where a coherent and consistent set of CIs is reconciled. The CIs of the federated CMDB do not need

to capture all details about the managed elements but at least basic information and references to the related CIs in the MDRs, where detailed information about them are captured. Furthermore, the federated CMDB provides interfaces, which are used by management tools to query and update CIs and management information.

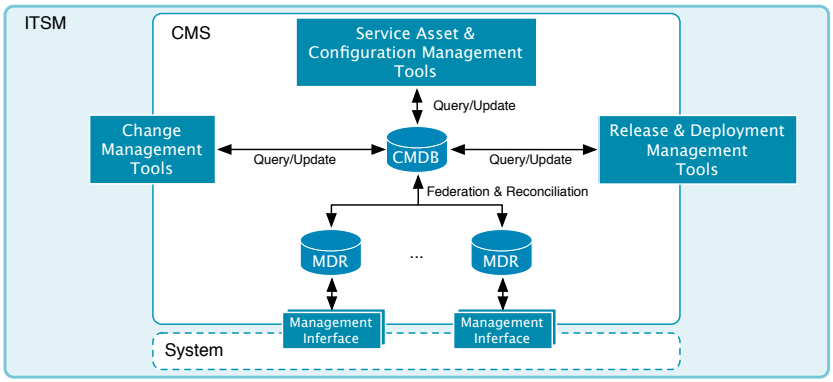


Fig. 1. A Common CMS within the Context of ITSM

2.2 Application of Model-Driven Engineering

Based on these insights, we suggest to apply techniques known from MDE in the sequel of this section. MDE is considered as the integration of metamodel based models and MDE techniques such as transformation, synchronization, merge, comparison, and analysis on models.

Management Tools Management tools can use management models as an underlying formalism, which enables management tools to provide model editors with a suitable concrete syntax to facilitate management by providing well-defined abstractions of the information to be managed. This is beneficial whenever the information that needs to be managed is complex and a visual representation would facilitate its understanding and management. Within management tools MDE techniques can be applied without any limitations. If multiple representations of management models are beneficial, transformation, synchronization or merging are suitable applications. The analysis of models can be applied for advanced reasoning. The outcome can be used in dashboards or reports, which supports decision making. In general, management tools can benefit from MDE as software engineering does from model-based *Integrated Development Environments* (IDE).

Federated CMDB A runtime model is applicable within a federated CMDB capturing the managed system. Thus, CIs that are part of the IT infrastructure of the managed system have to be captured in the runtime model. In addition, the runtime model has at least to capture well-defined interconnections between its CIs and changeable configuration properties that influence the operation of the item in the managed system. We further call such a runtime model a *configuration model*. Logical dependencies between CIs can be captured within the configuration model or in an explicit dependency model which should be managed

automatically by means of model analysis or at least manually. For each management process, an appropriate management model is beneficial, which has to capture all required management information and the ability to capture relationships to CIs of the configuration model. For example, a management model for *Change Management* might capture detailed information about planned changes that contain relationships to CIs that are part of the configuration model and on which the planned changes have to be performed.

MDR A runtime model is also applicable to MDRs which, however, only captures a specific subset of CIs of the managed system. Moreover, this runtime model is usually vendor specific, which means that it contains vendor specific information which is not captured in the configuration model of the federated CMDB. We further call the MDR runtime model a *vendor specific configuration model*.

Query/Update Connecting a management tool to the federated CMDB can be conducted by just copying the required models into the management tool or by applying a transformation/merge that provides a more comprehensive model tailored to the underlying management process. A transformation/merge combines several models of the federated CMDB into a single model that can be further used in a model editor with suitable concrete syntax. Furthermore, the changes that are made to the models in the management tool have to be transformed back to the models of the federated CMDB. This task is called update.

Federation & Reconciliation Transformation is applicable to federating the vendor specific configuration model of each MDR into a *partial configuration model* within the federated CMDB. A partial configuration model is a subset of the configuration model of the federated CMDB. The configuration model is derived by reconciling partial configuration models by applying model merge.

2.3 Vision of Autonomic IT Service Management

Considering the application of MDE, we can increase the level of autonomy in ITSM to make progress in closing the control loop for autonomic computing in ITSM. The CMS is able to automatically derive a runtime model in the form of a configuration model and the other direction can be reached by automatically propagating changes back into the system based on changes of the configuration model. We approach both directions in our prototypical implementation in the following section. Thus, a model-driven CMS fulfills the pre-requisite to autonomic computing in ITSM. To increase the autonomy, an autonomic manager is required, which automatically decides and derives changes based on findings of a model analysis. The analysis is performed on the configuration model and on models capturing *Service Level Agreements* (SLAs) or *Key Performance Indicators* (KPIs) and it discovers malfunctions in the managed systems that have to be resolved by subsequent changes. However, as proposed in [12], full autonomy in ITSM is currently only feasible in *Service Operation* considering small changes that are used to keep the system running at a certain quality level. More pervasive changes that are defined in *Service Transition* tends to be related to evolution and thus are currently quite difficult to be automated.

3 Prototypical Implementation

In an undergraduate seminar we started implementing several aspects of the model-driven CMS, as proposed in the previous section. Currently, we have implemented an MDR for *Enterprise Java Beans 3.0* (EJB) servers and applications, a federated CMDB based on Eclipse CDO[1], and simple management tools for *Service Asset & Configuration Management*, *Change Management* and *Release & Deployment Management*, which are implemented within Eclipse and EMF[2].

3.1 MDR for EJB Servers and EJB Applications

The vendor specific configuration model of the MDR represents all EJB servers[3] that can be discovered in the IT infrastructure and the EJB applications hosted by these servers. Each server provides the *mKernel* [13] extension, which is used as an interface for managing deployed EJB applications. Beside the existence of the servers, the vendor specific configuration model also captures details about EJB modules, that are hosted on the server, like the enterprise beans and their interconnections that are part of the EJB modules. In certain intervals or whenever changes within EJB-based applications occur, the vendor specific configuration model is updated accordingly by facilitating the *mKernel* extension.

3.2 Federated CMDB based on Eclipse CDO

The federated CMDB implementation is based on Eclipse CDO, which is in general an EMF model repository based on a database persistence layer. Thus, the federated CMDB is a structured model repository that stores EMF models in a database. Our configuration model reflects the architecture of the managed system containing software components[4], connectors between software components and hardware components with links between them as interconnections and deployment relationships between components. All of these elements are considered as CIs of the managed system. Additionally, all components can be related with configuration properties and are related with logical dependencies. In general, our configuration model has similarities to a UML deployment diagram. In addition to the configuration models, we foster an asset model within the federated CMDB. The asset model defines configuration variability of all authorized CIs of the managed system. The CIs in the asset model can be considered as types of the CIs in the configuration models. We further distinguish between *as-is configuration models*, which reflect snapshots of the actual configuration of the managed system, and *to-be configuration models*, which define snapshots of the authorized configuration of the managed system.

3.3 Service Asset & Configuration Management Tool

Service Asset & Configuration Management is essential to a CMS. Therefore, we have implemented a tool that provides up-to-date as-is configuration models gathered from vendor specific configuration models of diverse MDRs at different points in time. Our implementation is able to apply the federation of multiple

[1] Connected Data Objects; http://www.eclipse.org/modeling/emft/?project=cdo
[2] Eclipse Modeling Framework; http://www.eclipse.org/emf
[3] Currently, we support only the Glassfish v2 server; https://glassfish.dev.java.net/
[4] EJB servers, modules, and enterprise beans are represented as a software component.

MDRs and the reconciliation of partial configuration models into a coherent and consistent as-is configuration model within the federated CMDB. The whole sequence is shown in Figure 2.

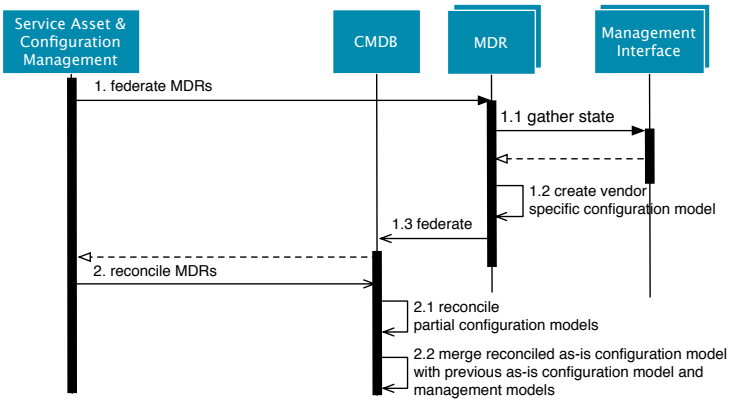


Fig. 2. Sequence Diagram of Federation and Reconciliation

First, each MDR or a subset of them is triggered to federate its vendor specific configuration model. This implies gathering the current state of the system through *mKernel* management interfaces. Based on the gathered state, a vendor specific configuration model is created which is subsequently automatically transformed into a partial configuration model.[5] Whenever all partial configuration models are available in the federated CMDB, the reconciliation is triggered which has to merge the partial configuration models into a new as-is configuration model. Afterwards, related management models and all elements of the previous as-is configuration model, i.e. logical dependencies which were not discovered by MDRs, are merged into the reconciled as-is configuration model.[6]

We further defined several KPIs in the context of *Service Asset & Configuration Management*, e.g., we measure the degree of discrepancy between the latest to-be and as-is configuration model, which is the number of coverages between the to-be configuration model and the as-is configuration model divided by the number of considered elements in the to-be configuration model. Therefore, we have specified a simple KPI model that is used to manage the KPIs and analysis rules to execute the KPIs. The KPIs are analyzed by applying the analysis rules to the KPI model, the to-be configuration model and the as-is configuration model. The outcome is visualized in a report that is created with Eclipse BIRT[7]. Another example is a KPI that measures the number of configuration misuses in the latest as-is configuration model by using the same analysis technique. Therefore, we first check for inconsistencies between the latest as-is configuration model and the asset model and subsequently count the number of

[5] The feasibility of model transformations and synchronization at runtime has already been shown in [14].

[6] Reconciliation of multiple MDRs is not implemented because we currently only support a single MDR.

[7] Business Intelligence and Reporting Tools; http://www.eclipse.org/birt/phoenix

inconsistencies that were found. We can also create analyses based on multiple as-is configuration models. The outcome is a report that maps the results for each analysis of an as-is configuration model on a timescale.

3.4 Change Management Tool

Changes to the system are indispensable due to several reasons: unsatisfied SLAs, changing requirements to the service realized through the managed system, etc. Thus, an appropriate tool for *Change Management* requires to define changes that have to be performed on the managed system. We have implemented a change management tool that is able to model changes directly on a configuration model that is queried from the federated CMDB. Therefore, the latest as-is configuration model of the federated CMDB is queried and then manually changed with a model editor. The resulting as-is configuration model is then sent back to the federated CMDB as an authorized to-be configuration model.[8] The whole sequence is shown in the sequence diagram of Figure 3.

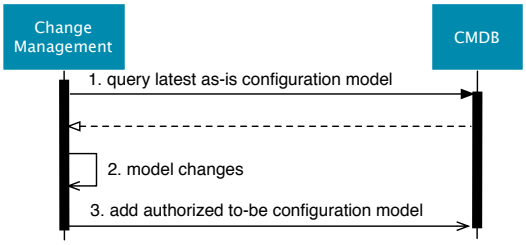


Fig. 3. Sequence Diagram of Applying Changes

3.5 Release & Deployment Management Tool

Release & Deployment Management is about planing, defining and rolling out sets of changes, e.g., a release[9] into the managed system. Therefore, the latest authorized to-be configuration model is queried from the federated CMDB and compared to the latest as-is configuration model using EMF Compare[10]. This comparison results in a model based on the EMF Compare metamodel that is afterwards transformed to an enhanced change model that is tailored to our domain. This change model supports the definition of basic operations such as (un)deployment, setting or changing configuring properties, etc. The change model is then propagated to all MDRs for execution. Each MDR consequently executes the change model by interpreting the operations as API calls for the connected *mKernel* management interfaces. The whole sequence is shown in Figure 4. Note that not all sub-processes of this management process are implemented since we were focusing on the automatic execution of changes that have been specified in the configuration model. Supported changes of the *mKernel* management interface are (un)deployment of EJB modules, changing configuration properties, and finally the creation and removal of interconnections amongst beans.

[8] Actually, the changes have to be validated and tested before they are deployed to the managed system. However, this was not the focus of the project.

[9] A release is a set of changes.

[10] http://www.eclipse.org/modeling/emft/?project=compare

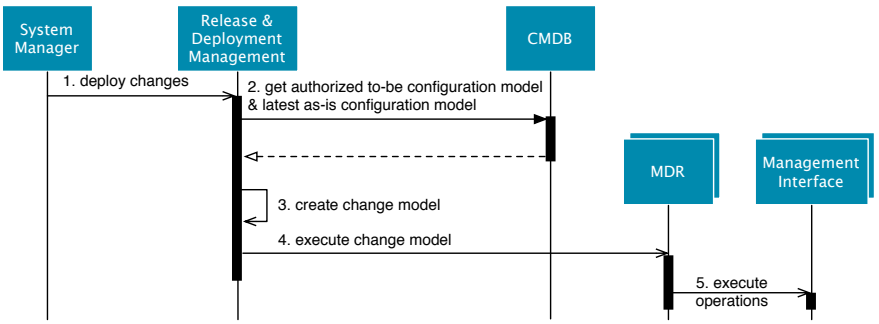


Fig. 4. Sequence Diagram of Deploying Changes

4 Application Example

Based on our prototype, this section describes a concrete example for the *Release & Deployment Management* (see Section 3.5). The dark-shaded elements in the *ConfigurationModel* in Figure 5[11] depicts the latest as-is configuration of a managed shopping system. In the current state, the shopping system is composed of a server *Server1* hosting an EJB module *WarehouseComponent*. This module packages the enterprise beans *WarehousingBean* and *ShipmentBean*, each of which provides a connector. The light-shaded elements of the *ConfigurationModel* reflect the authorized changes that should be executed. These changes were manually modeled during the *Change Management* (see Section 3.4). These changes consist of a deployment of the module *ShoppingCartComponent* containing a *ShoppingCartBean* and of attaching the *ShoppingCartBean* to the connectors provided by the *WarehousingBean* and *ShipmentBean*. Thus, the *ConfigurationModel* reflected by the dark and light-shaded elements is the to-be configuration model of the shopping system.

Starting the roll out of changes, the authorized to-be and the latest as-is configuration model are queried from the CMDB (see Figure 4 in Section 3.5). To obtain the authorized changes, both models are compared using *EMF Compare* (see Activity 1 in Figure 5), which results in a *EMF Compare Model* reflecting the differences between both models. However, this model is generic and only contains syntactical information about changes. Therefore, we automatically derive a semantically rich change model from the *EMF Compare Model* through a model transformation using appropriate transformation rules for the EJB domain (see Activity 2 in Figure 5). In our example, the transformation results in a *ChangeModel* as depicted in Figure 5. It reflects the authorized changes of deploying the *ShoppingCartComponent* and of attaching the *ShoppingCartBean* to two connectors, which has been described above.[12] Finally, the *ChangeModel* and the to-be configuration model are sent to the responsible MDR that abstract

[11] This configuration model uses a simplified metamodel and is shown as abstract syntax.

[12] Usually we use unique IDs in the *ChangeModel* for identifying the related components and connectors in the operations. For sake of readability, we use unique names here.

from the management interface provided by *mKernel*. This MDR inteprets both models and derives operations from both, which are finally executed using the *mKernel* API.

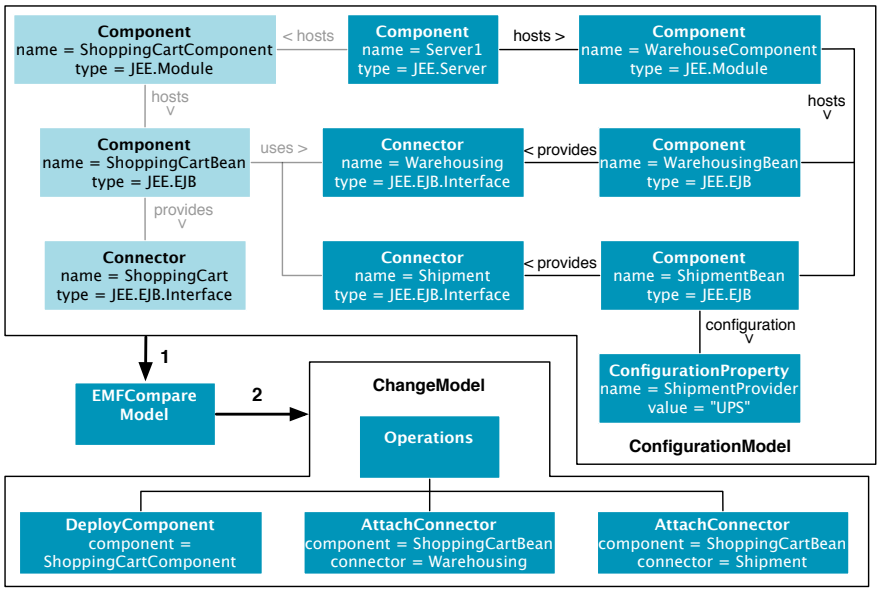


Fig. 5. Models for *Release & Deployment Management*

5 Conclusion & Future Work

We have outlined a common CMS supporting ITSM, as proposed by ITIL v3 and state-of-the-art implementations. Based on a common CMS, we suggested possible applications of MDE and further showed a prototype implementation of a model-driven CMS with basic management tool support. The prototype implementation provides a federated CMDB based on Eclipse CDO, an MDR for EJB servers, EJB applications and basic tool support for *Service Asset & Configuration Management*, *Change Management* and *Release & Deployment Management*. The prototype further implements a closed control loop that automatically derives a runtime model (as-is configuration models) from the managed system and also automatically applies configuration changes to the system based on change models, which are automatically derived from an adapted runtime model (to-be configuration model).

As future work we want to further elaborate our approach to validate our hypothesis of improving the efficiency of ITSM by providing a model-driven CMS. We further plan to implement additional MDRs to improve the coverage of the configuration model of the federated CMDB and further implement reconciliation of partial configuration models. We further want to add additional management tools for other management processes and refine the existing ones. Another future direction is to improve the autonomy of the model-driven CMS by providing an autonomic manager that to some extent supports *Service Operation* with simple policies.

Acknowledgment We thank Alexander Krasnogolowy, Mark Liebetrau, Steven Reinisch, Janek Schumann, Martin Sprengel, and Sebastian Waetzoldt for their contributions to the prototype implementation and Gregor Gabrysiak for reviewing this paper.

References

1. Salehie, M., Tahvildari, L.: Autonomic computing: emerging trends and open problems. In: Proc. of the Workshop on Design and Evolution of Autonomic Application Software, ACM (2005) 1–7
2. Lindquist, D., Madduri, H., Paul, C.J., Rajaraman, B.: Ibm service management architecture. IBM Syst. J. **46**(3) (2007) 423–440
3. Ward, C., Aggarwal, V., Buco, M., Olsson, E., Weinberger, S.: Integrated change and configuration management. IBM Syst. J. **46**(3) (2007) 459–478
4. Brittenham, P., Cutlip, R.R., Draper, C., Miller, B.A., Choudhary, S., Perazolo, M.: It service management architecture and autonomic computing. IBM Syst. J. **46**(3) (2007) 565–581
5. Johnson, M.W., Hately, A., Miller, B.A., Orr, R.: Evolving standards for it service management. IBM Syst. J. **46**(3) (2007) 583–597
6. Garlan, D., Schmerl, B., Chang, J.: Using Gauges for Architecture-Based Monitoring and Adaptation. In: Proc. of the Working Conference on Complex and Dynamic Systems Architecture. (2001)
7. Caporuscio, M., Marco, A.D., Inverardi, P.: Model-based system reconfiguration for dynamic performance management. Journal of Systems and Software **80**(4) (2007) 455 – 473
8. Akkerman, A., Totok, A., Karamcheti, V.: Infrastructure for Automatic Dynamic Deployment of J2EE Applications in Distributed Environments. In: Proc. of the 3rd Intl. Working Conference on Cmponent Deployment, Springer (2005) 17–32
9. Hnetynka, P.: A model-driven environment for component deployment. In: 3rd ACIS Intl. Conference on Software Engineering Research, Management and Applications. (2005) 6–13
10. Hein, C., Ritter, T., Wagner, M.: System Monitoring using Constraint Checking as part of Model Based System Management. In: Proc. of the 2nd Intl. Workshop on Models@run.time. (2007)
11. Morin, B., Barais, O., Jézéquel, J.M.: K@RT: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines. In: Proc. of the 3rd Intl. Workshop on Models@run.time. (2008) 127–136
12. Gacek, C., Giese, H., Hadar, E.: Friends or Foes? – A Conceptual Analysis of Self-Adaptation and IT Change Management. In: Proc. of the Workshop on Software Engineering for Adaptive and Self-Managing Systems, ACM (2008)
13. Bruhn, J., Niklaus, C., Vogel, T., Wirtz, G.: Comprehensive support for management of Enterprise Applications. In: Proc. of the 6th ACS/IEEE International Conference on Computer Systems and Applications, IEEE (2008) 755–762
14. 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, ACM (2009) 67–68
15. Robert, S., et al.: Deliverable d5.1a: Model based system management state of the art. Technical report, ModelPlex: Modeling solution for complex software systems, https://www.modelplex.org (2007)

Design for an Adaptive Object-Model Framework

An Overview

Hugo Sereno Ferreira[1,2], Filipe Figueiredo Correia[2], and Ademar Aguiar[1,2]

[1] INESC Porto

[2] Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
{hugo.sereno, filipe.correia, ademar.aguiar}@fe.up.pt

Abstract. The Adaptive Object-Model (AOM) architectural pattern has been significantly documented in literature, but there is not yet enough documentation explaining how to design and build a full AOM-based system. A AOM framework would need to address an additional number of issues that go well beyond individual software patterns. In this paper, we propose a design for a AOM framework that addresses several issues of building AOM-based systems, namely: integrity, run-time co-evolution, persistency, user-interface generation, communication and concurrency. We borrow concepts from distributed version-control systems. We show how applications based on a concrete realization of this framework, called Oghma, helps to avoid a traditional two-level domain classification, reduces accidental complexity, and directly exposes confined model evolution to the end-user.

Key words: Adaptive Object-Models; Frameworks; Software Design

1 Introduction

The industrialization of software development has been increasingly faced with the growth of software complexity. A considerable effort in development is also repeatedly applied to the same tasks, despite all the effort in research of reuse techniques and good practices, thus suggesting that reuse may need to be performed even more and at higher-levels. Hiding these inherent complexities of technological concerns by creating abstractions as been a recurrent reaction, at the cost of widening the existing gap between specification and implementation artifacts [9]. To make these abstractions useful not only for modeling, documentation, analytical and reasoning purposes [5, 12], models have to be made executable, by systematic transformation [15] or interpretation [14] of problem-level abstractions (i.e. specifications) into implementations (i.e. algorithms).

1.1 System Variability and Evolution

The difficulty of acquiring, inferring, capturing and formalizing software requirements is a recurrent problem in software engineering. This is because not only those processes are dependent upon the stakeholders' perspectives, but also because requirements often change faster than implementations. Since evolving software requires a considerable effort, as the implementation progresses, a strong resistance against changing the requirements is developed. What seems to be dismissed is that not only some business domains rely on constant adaptation of their own processes, but also that new knowledge is incrementally acquired as development unfolds, leading to new insights and expectations from software. If it is known *a priori* that the systems being developed are incomplete by design, won't there be benefits in designing for incompleteness? [11] Even so, current practices often focus on quick functional change, disregarding conceptual design, leading to a Big Ball of Mud, and eventually facing total reconstruction with a significant impact in economy [8]. Successful software needs to increase its resilience to change [13].

1.2 Framework for Adaptive Object-Models

Approaches to the use of models have traditionally been generative, automatically refining models into code artifacts during development. However, such techniques are based upon two premises: (a) that changes are always introduced by developers, within the development environment, and (b) that a full compile cycle (e.g. shutting down the system) is affordable. When these premises fail to hold, generative approaches may reveal insufficient, thus leading to the use of runtime domain models [14]. The systematic search for higher levels of abstractions — both to improve analysis and increase reuse — associated with the pervasive adoption of object-oriented models, converged to a common architectural style called the Adaptive Object-Model (AOM) [19, 18], which is founded on a growing collection of AOM-related patterns [7, 17, 16]. As patterns, they usually occur not as reusable components, but as perceived abstractions within the design of each particular system. In this paper, whenever referring to a pattern we use a SmallCase typographical style.

Frameworks are both reusable designs and implementations, that orchestrate the collaboration between core entities of a system. While they establish part of the system's behavior, they are deliberately open to specialization by providing hooks and specialization points. The framework dictates the architecture of the underlying system, defining its overall structure, key responsibilities of each component, and the main thread of control. It captures design decisions common to its application domain, thus emphasizing design reuse over code reuse. Patterns differ from frameworks because (a) are more abstract, (b) have smaller architectural elements, and (c) are less specialized [10].

Section 2 will provide a general overview of a framework which supports the development of AOM-based systems, and then proceed to detail each concern independently. Section 3 focus on its use within industry. Section 4, will draw some conclusions and present remaining issues to be addressed in the future.

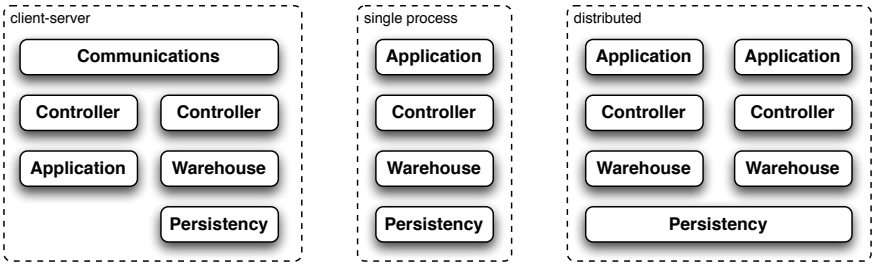


Fig. 1. Three possible component configurations of the framework: (a) client-server, where several processes are controlled by a centralized server, (b) single-process, only allowing a single running application, and (c) distributed, which takes advantage of the data-replication mechanisms from the underlying persistency engines.

2 Oghma: Architecture and Design

Oghma, which components are depicted in Fig. 1, is a framework to develop AOM-based systems, that balances adaptability and reuse. It supports the creation of models resembling MOF [1] and UML [2], and aims at covering the entire cycle of system creation and evolution. It also allows the introduction of changes to the system during runtime, thus providing a particular kind of confined end-user development.

Furthermore, the framework leverages the infrastructure used to support system evolution to provide additional features, such as auditing over the system's usage, and *time-traveling* to an arbitrary point along its evolution (i.e. to set the system in a past state).

Oghma includes a set of interchangeable components designed to have an high degree of flexibility — it supports several types of persistency engines, including relational, object-oriented, key-value and document-oriented, and architecture styles, such as single-process, client-server, and distributed.

2.1 Core — Structural

Fig. 2 depicts the design of the structural core of Oghma, resembling the TypeSquare [19] pattern: (a) `ObjectType`, which is refined into `Entities` (that represent classes) and `Interfaces`, (b) `Instance`, which complies to a given `Entity`, (c) `PropertyType`, which is refined into `RelationNodes` and `Attribute-Types`, and (d) `Property` which complies to its `PropertyType`. Each `Relation-Node`, besides specifying cardinality, navigability and role, must be connected to another node, thus establishing a `RelationType`. In order for a `RelationType` to have properties (similar to the *Associative Class* in UML) it can relate to an `Entity`. `Entities` can also inherit from other `Entities` and/or multiple `Interfaces`. Model-defined `Entities` and `Instances` can be made *Types* and *Objects* of the underlying programming language through the use of Plugins.

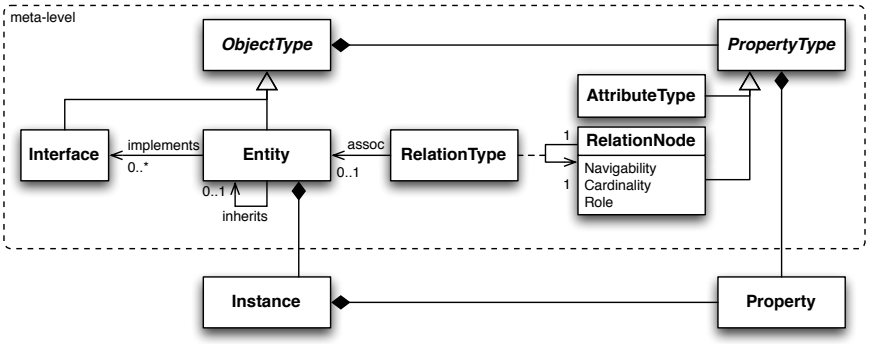


Fig. 2. Core design of the structural meta-model.

2.2 Core — Behavioral

Fig. 3 depicts `Expression` as the central concept of the behavioral core, extending the Rule Object pattern [17]. Expressions are stated in a *Domain Specific Language* and may be evaluated using an Interpreter [10] or a *Virtual Machine*. They're widely used to define: (a) `ObjectType` invariants, (b) derivation rules in `PropertyTypes` and `Views`, (c) body of `Methods`, (d) guard-conditions of `Operations`, etc. As such, they play an important role in assuring semantic integrity during model evolution (see Section 2.7). Structural rules, such as the cardinality and uniqueness of a `PropertyType` are translated to `ObjectType` invariants. `Methods`, which are used-defined `Batches` of `Operations`, may be invoked manually, or triggered by `Events`, thus allowing the specification of *State Machines*.

2.3 Controller

As seen in Fig. 1, the Controller serves as a entry layer for the GUI and Communications components, and it's key responsibility is to orchestrate the several other components in the framework by establishing a thread of control. It bootstraps the system by loading the meta-model, and the necessary versions of the domain-model from the Warehouse. It manages data requests by interacting with the Warehouse. It also provides several *hooks* to the framework through Chains of Responsibility and Plugins (e.g. interoperability with third-party systems by allowing subscribers to intercept requests).

2.4 Warehousing and Persistency

Because of the evolutive nature of the model, mapping to a classic relational database through the use of ORMs complexifies co-evolution. Warehousing (Fig. 1) hides the details of persistency from the Controller, exposing and consuming data and meta-data (i.e. `Things`), and managing versioning (i.e. through

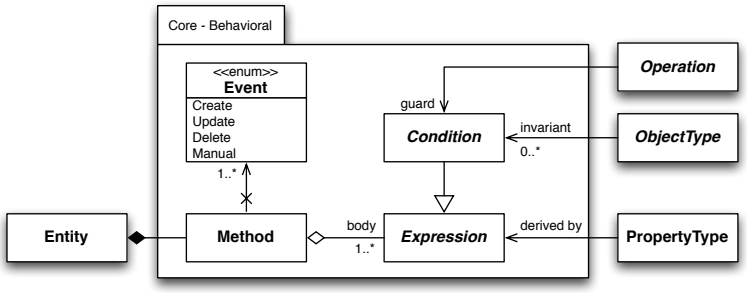


Fig. 3. Core design of the behavioral meta-model.

`Versions` and `States`). Its behavior can be extended and modified through inheritance and composition, as by the DECORATOR [10] pattern. Transient memory-only, direct data-base access, lazy and journaling strategies (e.g. using CACHES [10]) are just a few examples of existing (and sometimes simultaneous) configurations. Also, `Things` are always regarded as opaque, key-valued objects.

2.5 Communications

This design allows to assemble several types of communication stacks. If a client-server HTTP-based stack is chosen, Oghma currently provides a RESTful API for communication between the `Server Controller` and `Client Controller` through a pair of `HTTP Bridge` and `Server Dispatcher` acting as PROXIES [10]. Every `Thing` is addressable by its unique identifier as a resource. The contents of `States` and `Changesets` are serialized in XML. Simple queries can be expressed directly in the URL; those more complex require POST methods.

By specialization of the communication layer, other types of technology can be used for bridging the controllers (e.g .NET Remoting). For example, in the case of a single-user stack, the `Client Controller` would interact directly with the `Server Controller`. A different approach from the client-server architecture is to use distributed key-value databases (e.g. CouchDB), to handle both persistency and communication. Here, every application would assume direct access to the Controller, delegating the responsibility of disseminating contents to the underlying data warehouse.

2.6 Integrity

The **Structural Integrity** of the run-time model is asserted through rules stated in the meta-model. For example, `Instances` should conform to their specified `Entity` (e.g. they should only hold `Properties` which `PropertyType` belong to its `Entity`). Nonetheless, evolving the model may corrupt structural integrity, such as when moving a mandatory `PropertyType` to its superclass (e.g. if it doesn't have a default value, it can render some `Instances` non-compliant).

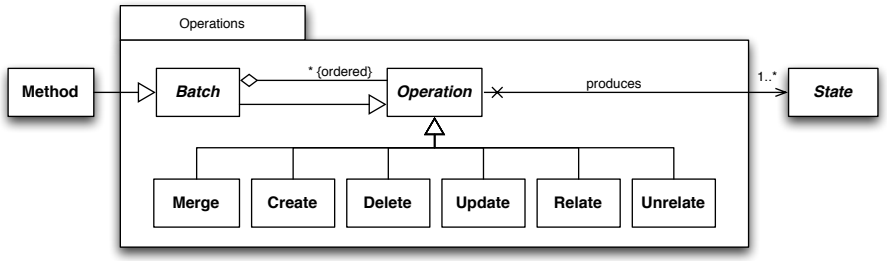


Fig. 4. Data and meta-data are manipulated through `Operations`, similarly to the COMMAND [10] pattern, which produces new `States` of `Things`.

Some model evolutions can be solved by foreseeing integrity violations and applying prior steps to avoid them (e.g. one could first introduce a default value before moving the `PropertyType` to its superclass).

Some steps of a particular evolution may also violate model integrity, although the end result would be valid. For example if a `PropertyType` is mandatory, one cannot delete its `Properties` without deleting itself and vice-versa. This problem is solved by the use of `Changesets`, and only enforcing integrity at the end.

Semantic Integrity, on the other hand, is much harder to ensure since it's not encoded as rules in the meta-model. One cannot just look to the results of an arbitrary evolution and infer the steps which have lead to it. Consider the scenario where an `AttributeType age` is renamed to `date-of-birth`, recalculated according to the current date, and moved to it's superclass `Person`. Would we rely on the direct comparison of the initial and final models, a possible solution would be to delete the attribute `age` in `Employee` and create the attribute `date-of-birth` in `Person`. However, the original meaning of the intended evolution (e.g. that we wanted to store birth-dates instead of ages) would be missed. To solve this problem, Oghma makes use of MIGRATIONS [7], providing and storing sequences of model-level operations that cascade into instance-level changes.

2.7 Evolution

Allowing collaborative co-evolution of model and data by the end-user introduces a new set of concerns not usually found in classic systems. They are (a) how to preserve model and data integrity, (b) how to reproduce previously introduced changes, (c) how to access the state of the system at any arbitrary point in the past, and (d) how to allow concurrent changes. These concerns can be summarized into traceability, reproducibility, auditability, disagreement and safety, and are commonly found on version-control systems.

Typically, *evolution* is understood as the introduction of changes to the model. Yet, the presented design doesn't establish a difference between changing data or meta-data; both are regarded as *evolutions* of `Things`, expressed as

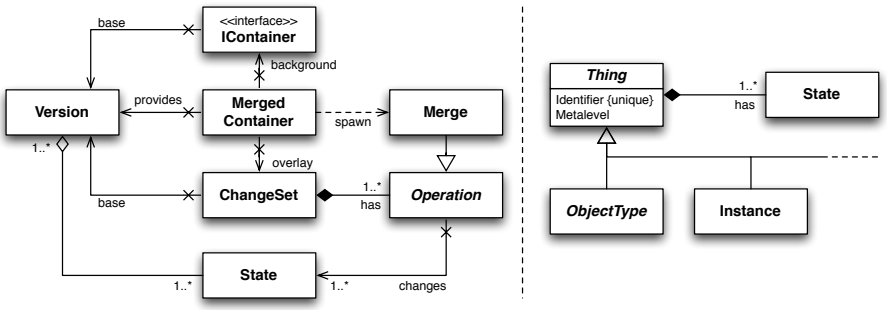


Fig. 5. (a) Merging mechanism used to validate and apply operations stored in a `Changeset`. (b) Data and meta-data are both viewed as `Things` and `States`.

`Operations` over `States`, and performed by the same underlying mechanisms as depicted in Fig. 4. To provide enough expressivity such that semantic integrity can be preserved during co-evolution, model-level `Batches` operate simultaneously over data and meta-data.

Sequences of `Operations` are encapsulated as `ChangeSets`, following the History of Operations pattern [7], along with meta-information such as user, date and time, base version, textual observations, and data hashes. Whenever the framework validates or commits a `ChangeSet`, the Controller uses the merge mechanism depicted in Fig. 5 (similarly to the System Memento pattern [7]), which dynamically overlays the modifications onto the base version by orderly applying each `Operation`, allowing for behavioral rules to be evaluated, and finally resulting in a new version.

If all changes to the data and model are preserved, one can easily recover past information. This not only solves the aforementioned issues, but it also brings to information systems the same notions of versioning that changed the scenario of collaboration in *wikis* and software development.

2.8 User-Interface

Oghma provides a run-time adaptive UI, by inspection and interpretation of the model and using a set of pre-defined heuristics and patterns. While detailing every heuristic applied is outside the scope of this paper, an overview is provided.

A set of grouped entry-points are presented to the user through the GUI. Groups correspond to `Packages` and entry-points to selected `ObjectTypes`. When choosing an entry-point, a list of the associated instances is presented, showing several details in distinct columns, inferred from special annotations made in the model; along with generic search mechanisms. `ObjectTypes` have two default views: *edition* and *visualization*. During edition, Oghma uses heuristics to render each `Property` by inspecting the cardinality, navigability and role of both nodes of a `RelationType`. The result is a different input panel, according to the

property in question: text-fields, text-areas, combo-boxes, tree-views, lists, embedded forms, etc.. Visualization is defined using `Views` — virtual `Instances` where every `Property` is derived — and are subsequently transformed through templates before being presented to the user. Mechanisms as clipboard (using object identifiers), undo and redo (using `Operations`) are orthogonally supported. Custom panels, either for special types (e.g. dates) or model-chunks (e.g. user administration), can also be loaded as PLUGINS.

The user workflow resembles those when using version-control systems. User changes are not immediately applied; instead, `Operations` are stored into the user `Changeset`, and sent to the `Server Controller`, allowing it to assert integrity and provide feedback on behavioral rules. When a user chooses to, it can *commit* its work to the server, by reviewing the list of `Operations` and additionally submitting a descriptive text about his work.

Awareness is also addressed through several feedback techniques: graphics that show the history of changes either in a particular `ObjectType`, by user or globally; alerts to the user for simultaneous pendent changes in the same objects, and presenting reconciliation screens when conflicts are detected; etc.

3 Oghma in the Industry

Oghma has been implemented in C# using the Microsoft .NET Framework v3.5, although the design here presented doesn't depend on a specific technology.

3.1 Use Cases

Oghma was already used to create production-level applications: (a) *Locvs*, an Information System for Management of Architectonic and Archaeological Heritage, and (b) *Zephyr*, a tool for document records management [3, 4]. Of particular interest is *Locvs*, whose domain model currently consists of more than 300 classes, and has gone through more than 1000 model versions, 12k instance-level commits, and 200k `Things`, throughout approximately 2 years of usage and evolution. It is deployed in dozens of machines using a client-server architecture. Performance tests have lead it to currently use SQLite as a storage and Full-Text Search engine. A custom-made DSL for specifying behavior was also implemented. This application will be a valuable asset to further research the role of AOMs in the development of large-scale information systems.

3.2 Lessons Learned

The development and usage of Oghma targeting adaptive applications allows us to elicit some lessons. First, the skills needed to deal with this type of architecture aren't trivial to find, and developers are not necessarily at ease to work at these levels of abstraction. From a framework standpoint, there's also a thin balance between a framework that makes the creation of new systems a quick and easy process, and one that is flexible enough to cover a wide scope of systems.

Because it's very tempting to make the framework address all use-cases using an adaptive and model-driven approach, there is a risk of the final models becoming as elaborate and complex as a full-blown programming language. In this sense, hooks are a key issue, as they are not always easy to foresee, but they establish the border line between what should be regarded as part of the framework and what is particular behavior of a specific instantiation.

Nonetheless, the conduction of small model experiences seem to show it's easy to quickly build a functional prototype which can be shown to the customer, thus providing very early feedback before refining it into a production-level application. Not only the costumer involvement in this process is also increased due to the end-user development capabilities offered by the framework, but it also reduces the burden of up-front design by allowing an incremental approach to formalization of the underlying business model.

4 Conclusions

Adaptive Object-Models and application frameworks are both solutions for a common problem: to increase software reuse. They try to minimize the effort of developing and evolving a software system. A AOM is a meta-architecture for domain variability; frameworks focus on providing code and design reuse. In this paper, they are combined and presented as a conceptual framework design which allows the creation of AOM-based systems, along with some details of a particular implementation being used within an industrial environment: *Oghma*.

Leveraging the concept of adaptability, end-users are empowered to introduce (confined) changes to the model at run-time. This choice raised several issues, such as traceability, reproducibility, auditability, disagreement and safety, which were addressed in the framework by borrowing concepts from distributed version-control systems. One of the side-effects of unifying data and meta-data evolution was that the classical two-level domain classification, where *types* are static entities, is diluted, thus reducing accidental complexity of applications [6]. For example, a user can edit an *enumeration*, or add a new *specialization* of a class, by directly editing the model, hence preserving the classification levels.

Yet, several open-issues remain to be addressed. While automatic run-time generated user-interfaces may not be on pair with custom-made ones regarding usability, they seem to be consistent and based on a strict set of metaphors, supporting a quick learning process by users. Nonetheless, which mechanisms should the framework provide to improve usability and customization of GUIs, while retaining the capability of automatically generating them?

Furthermore, no studies on the performance, robustness, usability, evolvability, maintainability, consistency, composability, scalability and several other software quality attributes regarding Adaptive Object-Models, in comparison to classical systems, were published yet, and even less regarding AOM frameworks.

In the design proposed in this paper, we've addressed data and meta-data evolution through an unified architecture. However, the whole framework depends on the definition of a well-known meta-model. Operations that support

model evolution are thus dependent on this definition. How often would the meta-model change, and how can we easily cope with its evolution? Should the abstraction be raised yet another level, or is a self-compliant meta-model enough to provide mechanisms for its own evolution?

Finally, this paper provides only an overview of the framework. There are several issues that should be taken into account when implementing it. Such details are outside of the scope of the work developed so far, but they are expected to be addressed in future work.

5 Acknowledgments

We would like to thank both *FCT* and *ParadigmaXis, S.A.* for sponsoring this research through the grant SFRH / BDE / 33298 / 2008.

References

1. MOF version 2.0. http://www.omg.org/spec/MOF/2.0/, Accessed on 2009/08/06.
2. UML version 2.2. http://www.omg.org/spec/UML/2.2/, Accessed on 2009/08/06.
3. Locvs. Technical report, ParadigmaXis, S.A. produced to CMP, 2009.
4. Zephyr. Technical report, ParadigmaXis, S.A. produced to CMP, 2009.
5. J. Arlow, W. Emmerich, and J. Quinn. Literate modelling—capturing business knowledge with the uml. *The Unified Modeling Language: UML'98*, 1999.
6. C. Atkinson and T. Kühne. Reducing accidental complexity in domain models. 2008.
7. H. Ferreira, F. Correia, and L. Welicki. Patterns for data and metadata evolution in adaptive object-models. *Proceedings of the 15th Conference on PLoP*, 2008.
8. B. Foote and J. Yoder. Big ball of mud. *PLoP'00*, 2000.
9. R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. *International Conference on Software Engineering*, Jan 2007.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.
11. R. Garud, S. Jain, and P. Tuertscher. Incomplete by design and designing for incompleteness. *Organization Studies*, Jan 2008.
12. J. Krogstie, A. Opdahl, and G. Sindre. Advanced information systems engineering: 19th international conference. Jan 2007.
13. D. Riehle and E. Dubach. Why a bank needs dynamic object models. *OOPSLA Workshop on Metadata and Active Object Models*, 1998.
14. D. Riehle, S. Fraleigh, D. Bucka-Lassen, and N. Omorogbe. The architecture of a uml virtual machine. Jan 2001.
15. M. Voelter. A catalog of patterns for program generation. 2003.
16. L. Welicki, J. Yoder, and R. Wirfs-Brock. A pattern language for adaptive object models: Part i-rendering patterns. *hillside.net*.
17. L. Welicki, J. Yoder, R. Wirfs-Brock, and R. Johnson. Towards a pattern language for adaptive object models. *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on OO programming systems and applications companion*, Oct 2007.
18. J. Yoder, F. Balaguer, and R. Johnson. Adaptive object-models for implementing business rules. *Urbana*.
19. J. Yoder, F. Balaguer, and R. Johnson. Architecture and design of adaptive object-models. *ACM SIGPLAN Notices*, Jan 2001.

Management of Runtime Models and Meta-Models in the Meta-ORB Reflective Middleware Architecture *

Lucas Luiz Provensi, Fábio Moreira Costa, and Vagner Sacramento

Institute of Computing, Federal University of Goiás
Campus Samambaia, UFG, 74690-815, Goiânia-GO, Brazil
{lucas,fmc,vagner}@inf.ufg.br
http://www.inf.ufg.br

Abstract. In the Meta-ORB reflective middleware architecture, runtime models provide the necessary meta-information to instantiate specialized platform configurations and to construct the reflective self-representation of base-level systems. Other kinds of useful meta-information may also be provided by the system's runtime model, such as policies that allow the middleware to adapt itself automatically. Evolving the middleware meta-model and extending its infrastructure to handle new kinds of model-based constructs may be considerably complex and would require re-implementation of several parts of the middleware. In this paper we present an approach for the management of runtime models and meta-models, aiming to simplify the evolution of the middleware so that it can support new kinds of constructs defined in its meta-model.

Key words: runtime models, meta-modeling, reflective middleware

1 Introduction

Distributed computing systems, especially those used for real-time applications, often suffer from dynamic changes in their execution environment and requirements, affecting, e.g., available bandwidth and network loss rates. Reflective middleware has been proposed as a way to handle such changes without disrupting the system or its applications. Such platforms are capable of inspecting and adapting their internal structure and behavior at runtime, providing an independent software layer which is ideal to implement the self-management and self-adaptation capabilities needed to handle dynamic requirements [1].

In this paper we review the Meta-ORB reflective middleware platform [2] and discuss how its evolution can be facilitated by the management of models and meta-models at runtime. Autonomic self-adaptation is proposed as a way to enable transparent middleware evolution, a feature that was not fully supported

* This work was partly funded by FAPEG, The Research Support Foundation of the State of Goiás, Brazil (Call 02/2007; Process number: 200810267000147), and by the HP Technology for Teaching Higher Education Grant.

in previous versions of the platform. This feature was introduced in the current version of the platform, called MetaORB.NET [3]. It enables new functionality and constructs to be dynamically added to the platform, as well as existing ones to be replaced or removed. The approach is based on the reflective manipulation of model and meta-model elements.

The paper is structured as follows. Section 2 discusses the Meta-ORB meta-model and the use of run-time models to instantiate specialized middleware configurations, as well as their use to build the reflective self-representation of base-level entities. Section 3 discusses middleware evolution in terms of its three-level architecture: meta-model, model and system entities. Section 4 discusses related work and Section 5 presents concluding remarks and future work.

2 Meta-ORB Model and Meta-model

The Meta-ORB architecture combines meta-information management techniques, which enable the definition of specialized platform configurations, with computational reflection, which enables dynamic adaptation. Meta-information, in the scope of our work, describes the structure and semantics of entities in a computational system. This description is used for static configuration of the middleware (by instantiating its components at load time) and for its dynamic adaptation (via runtime component-based reconfiguration).

In the Meta-ORB architecture, meta-information is specified in a model, according to an explicit meta-model. This explicit meta-model represents the platform's type system and is maintained in a repository that can be used for the definition, storage and retrieval of models that represent specialized configurations of the middleware and its applications. Once the definition of an entity (a component and its interfaces, for instance) is obtained from the type repository, it may be used to build a runtime model of the entity, allowing its dynamic instantiation by specialized factories and, if necessary, the construction of its reflective self-representation, used for dynamic introspection and reconfiguration.

The main constructs defined in the meta-model are components, that encapsulate functionality and can interact (and be composed) through well defined interfaces, and binding objects, that encapsulate interaction behavior, and can be defined in terms of internal components and other binding objects (a complete description of all meta-model constructs can be found in [2]). Figure 1 shows the creation of a component, where its definition is obtained from the type repository and parsed into a model that is used by the component factory as a blueprint to instantiate the component and its interfaces. The same process is performed by binding factories to instantiate binding objects.

The same meta-information contained in component and binding definitions can also be used to build the self-representation maintained by meta-objects that reify components and bindings. Figure 2 shows the construction of the self-representation of a binding object. First, the meta-object obtains the definition of the binding from the type repository. This definition contains meta-information that describes the internal configuration of the binding in terms of internal com-

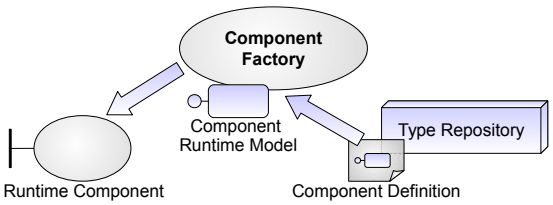


Fig. 1. Creating a component using a runtime model as the blueprint.

ponents and nested bindings. The meta-information is then combined with information maintained by the middleware at runtime, such as the location of each of the endpoints that participate in the binding. The result is compiled into a graph maintained by the meta-object, which contains information about the internal configuration of the binding in each of its endpoints. Once the self-representation is built, meta-objects can be used, via their meta-interfaces, to inspect and adapt the corresponding entities of the base-level system.

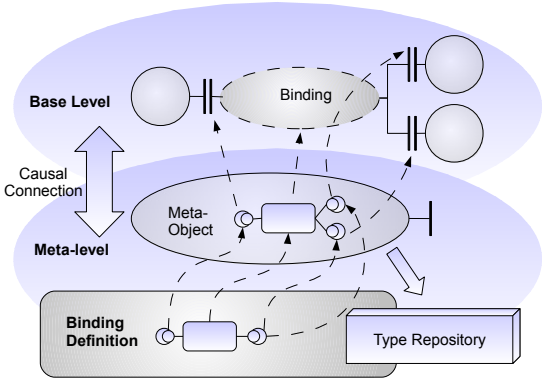


Fig. 2. Building the self-representation for a binding object.

The meta-information used by factories and meta-objects basically refers to the structure of the base-level entities: required and provided interfaces of components and the internal configuration of components and bindings. However, the platform's meta-model allows the addition of other kinds of meta-information to the definition of entities in order to describe their runtime behavior. One example is the definition of QoS constraints, which can be added as annotations to the definition of interfaces. Such annotations may indicate, for instance, the maximum acceptable delay for a media flow passing through the interface. This information in turn can be used in the negotiation process that will determine the type of binding that will be used to connect two or more interfaces.

More recently, the concept of adaptation policy was introduced in the meta-model [3]. As well as QoS constraints, policies describe desirable behavior for the entities they refer to. Policies provide the middleware with meta-information to determine, at runtime, when and how the platform should adapt itself in response to changes in the execution environment or requirements. The introduction of this and other behavioral concepts results in changes to the middleware that affect meta-model, model and base-level entities likewise. However, as such changes are not related to structure, they do not alter the main constructs of the meta-model (components, interfaces and bindings). The management of middleware evolution thus becomes easier, making it possible to apply meta-model changes without disrupting execution, as will be discussed in the next section.

3 Middleware Evolution

Adding new behavioral features, such as self-adaptation, requires several changes to the middleware, comprising three of the four levels described by the Meta-Object Facility (MOF) [4]. First, the middleware meta-model (Level 2) should be extended with constructs to describe the new features. Then, these new features should be incorporated into new and existing middleware models (Level 1). Finally, runtime entities (Level 0) may need to be modified (or created) to interpret the new features that are now part of the runtime model.

As an example, Figure 3 illustrates the changes made to Meta-ORB in order to add self-adaptation support. Firstly, package *policies* was added to the meta-model, with constructs for the definition of adaptation policies and their associations with other meta-model elements. Then, existing models were modified according to the new meta-model, e.g., each binding definition can now be associated with adaptation policies. Finally, the binding factory (which is at level 0) was modified to interpret binding definitions associated with policies, and a new component, the *Adaptation Manager*, was created to manage the application of the policies. Each of these steps will be discussed in more detail next.

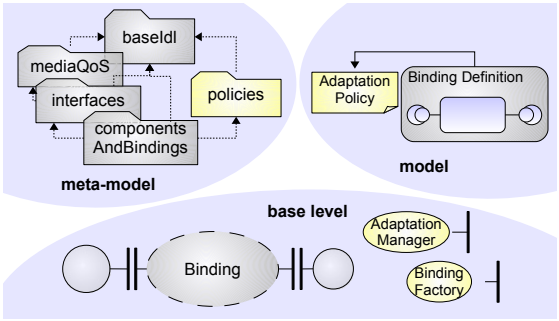


Fig. 3. Middleware extensions to support adaptation policies.

3.1 Meta-model Extension

Extending the meta-model (level 2) is probably the less painful work for the designer of new features, although such extensions usually affect other levels (levels 1 and 0), where the treatment is more complex. The meta-model itself is described using a meta-meta-model (level 3), allowing the meta-modeling of new concepts in accordance with a common language used to describe meta-models (known as the MOF model). Thus, the designer can use a modeling tool and a well-known language to make the necessary changes to the meta-model, making this task as easy as modeling any level 1 system.

In its current implementation [5], the meta-model of Meta-ORB was defined using EMF (Eclipse Modeling Framework) [6]. EMF is a plug-in for the Eclipse platform that facilitates the construction of tools and applications based on structured data models that are defined using a subset of the MOF model. The framework allows the manipulation of the middleware meta-model using graphical plug-ins for Eclipse or other compatible modeling tools. It also allows the generation of the core implementation of the type repository. This core implementation consists of Java classes that represent each of the meta-model constructs.

3.2 Model Transformation

Changes to the meta-model should be reflected in the middleware model. This means that the type repository must be (partly) re-generated to recognize the new features, enabling the definition, modification and access to meta-information elements that represent them.

With this extended repository, it is possible to define new types either programmatically or through the generated Eclipse graphical plug-in. Types created in the repository can be serialized into XMI, a programming language-independent format [7], and stored in local files. When necessary, the middleware model can be reloaded from these files into memory.

To retrieve the types, remote instances of the platform can access the type repository using a Web service, as shown in Figure 4. The service takes a type, which is a runtime instance of an EMF *EObject* (1), serializes it into XMI (2) and returns it to the remote platform (3). The remote platform is free to take the type and convert it into local objects using the appropriate programming language constructs in order to build a runtime model that provides useful meta-information to the middleware (4). For instance, factories can use the model to instantiate entities at runtime (5).

Changes to the meta-model do not affect the way the repository is accessed (via the Web service). However, with the new implementation of the repository (generated after the meta-model extension), model elements defined using the old meta-model (which were serialized and stored in local files) may no longer be reloadable due to incompatibilities between the meta-models. To avoid redefining the middleware model completely, the designer can edit the model manually (using the serialized XMI files) or use some model transformation technique [8]

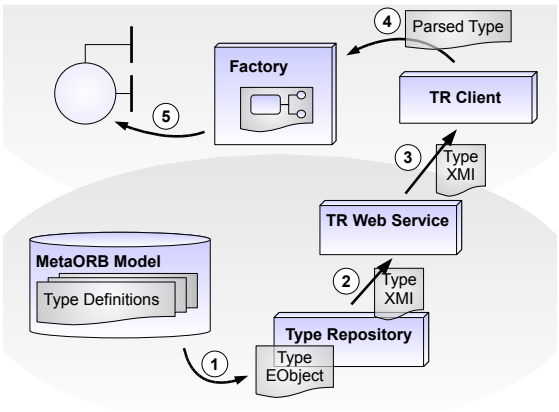


Fig. 4. Type Repository Web Service.

(and a tool compatible with EMF models) to adjust the model to the new meta-model. Adjusting the model manually can be a daunting task, depending on the extent of the changes made to the meta-model. It is generally less difficult to define rules to transform the model described in accordance with the previous meta-model into a model that conforms to the new meta-model.

3.3 Updating Runtime Entities

As seen in Section 2, the middleware makes extensive use of meta-information contained in the type repository. Special entities of the middleware core, such as component and binding factories, are capable of interpreting this meta-information at runtime. Thus, changes to the meta-model that affect existing types, as well as the introduction of new types, require modification of these entities or the creation of new ones to handle the new types at runtime.

The creation of new core middleware entities follows the normal procedure defined by the Meta-ORB platform, which consists in defining the type of the entity in the repository and implementing its primitive code in the host programming language (in the case of primitive components and bindings). The new entity can then be dynamically instantiated using an appropriate factory. An entity whose type has changed, however, may not be immediately replaced by a new entity, as the system may be running and the entity may still be in use and referenced by other entities of the middleware.

To this end, the repository uses type versioning: a change in a type implies the creation of a new version of this type. Operation *UpdateToVersion()* was thus added to the specialized factories of the platform, allowing the dynamic update of entities that have new type versions. This operation gets the new version of the type from the repository and uses the reflective framework of the platform to dynamically adapt the entity in accordance with its new version. In the case of primitive components or bindings, which cannot be adapted through structural

reflection, the factory re-instantiates its internal implementation preserving its existing interfaces and adding new ones if necessary.

4 Related Work

In Meta-ORB, meta-modeling and computational reflection are key aspects, which means that middleware evolution is subject to the evolution of its meta-model. At this point, our work relates to other works focused on meta-model evolution, such as [9], which presents an approach for automatic and gradual evolution of generic meta-models. Our approach, however, is more focused on the evolution of a meta-model to provide new sources of meta-information for the middleware. It does not address aspects such as the characterization of the relationships between meta-models and the automatic co-adaptation of models.

Our work is also related to other middleware approaches that use runtime models as a source of meta-information, such as [10] and [11]. Both use architectural runtime models to automatically drive middleware adaptation. However, these works do not define an explicit meta-model as in Meta-ORB. We believe that an explicit meta-model is important for middleware evolution and, with the help of meta-modeling tools such as EMF, it is possible to modify this meta-model as easily as modifying any (level 1) model. In the Meta-ORB platform, the meta-model is also used to generate the type repository implementation, which is an important tool for the management of meta-information.

5 Concluding Remarks and Future Work

In this paper, we presented our approach for managing models and meta-models in the Meta-ORB reflective middleware platform. We also discussed the way it can facilitate middleware evolution. The paper shows how changes made to the meta-model can be reflected in the middleware model and consequently affect the entities that form a (base-level) runtime middleware configuration.

One of the main limitations of this work refers to the possibility of modifying only meta-types that do not structurally affect the middleware programming model. This means that it is not possible to redefine the meta-model concepts of component, interface and binding, as this would imply a complete re-implementation of the middleware core based on these modified concepts. Thus, only changes aimed at providing new types of behavioral meta-information are considered, not affecting the structural part of its programming model. The ability to evolve the middleware programming model still needs further study.

Another important aspect that was not addressed in this work is the creation of an integrated toolkit for defining and manipulating models and meta-models, which can even include a model transformation tool. There are works, such as [12], which propose frameworks for in-place model transformation based on EMF and can be employed for this purpose. In this way, the middleware designer would be able to use the same tool to modify the meta-model, create transformation rules and automatically adapt existing models.

As seen in Section 4, several proposals suggest the use of runtime models to provide information that is important for the autonomy of middleware platforms. The models can hold useful information about the middleware architecture, as well as information about how it must operate. In this paper, we argue that maintaining an explicit and unified meta-model, combined with an infrastructure to support the modeling and meta-modeling of middleware configurations, is an important improvement. This offers a structured and more natural way to evolve the concepts employed in the runtime models used by the middleware.

Finally, as reported in [3], we have successfully used this approach to add autonomic adaptation capabilities to improve the support for a class of multimedia applications in Meta-ORB. We are currently investigating the use of meta-model extension to enhance Meta-ORB's support for other classes of distributed applications, especially as part of a larger grid and cloud computing infrastructure.

References

1. Blair, G.S., Coulson, G., Blair, L., Duran-Limon, H., Grace, P., Moreira, R., Parlavantzas, N.: Reflection, Self-awareness and Self-healing in Open ORB. In: WOSS '02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, ACM (2002) 9–14
2. Costa, F.M.: Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware. PhD thesis, University of Lancaster (2001)
3. Provensi, L.L., Costa, F.M., Sacramento, V.: Self-adaptive Middleware for Digital Ink Based Applications. In: ARM '08: Proceedings of the 7th workshop on Reflective and adaptive middleware, New York, NY, USA, ACM (2008) 29–34
4. Object Management Group: Meta Object Facility (MOF) Core Specification. "`http://www.omg.org/spec/MOF/2.0/`", accessed in September, 2009 (2006)
5. Costa, F., Provensi, L., Vaz, F.: Using Runtime Models to Unify and Structure the Handling of Meta-information in Reflective Middleware. LECTURE NOTES IN COMPUTER SCIENCE **4364** (2007) 232
6. The Eclipse Foundation: Eclipse Modeling Framework Project (EMF). "`http://www.eclipse.org/modeling/emf/`", accessed in September, 2009 (2009)
7. Object Management Group: XML Metadata Interchange (XMI). "`http://www.omg.org/spec/XMI/2.1.1/`", accessed in September, 2009 (2007)
8. Bézivin, J.: From object composition to model transformation with the MDA. In: Proceedings of TOOLS'USA. (2001) 350–354
9. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. Lecture Notes in Computer Science **4609** (2007) 600
10. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-adaptation with Reusable Infrastructure. IEEE Computer **37**(10) (Oct. 2004) 46–54
11. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E., ICT, S., Trondheim, N.: Using architecture models for runtime adaptability. IEEE software **23**(2) (2006) 62–70
12. Biermann, E., Ehrig, K., Kohler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical definition of in-place transformations in the eclipse modeling framework. Lecture Notes in Computer Science **4199** (2006) 425

Modeling Context and Dynamic Adaptations with Feature Models

Mathieu Acher[1], Philippe Collet[1], Franck Fleurey[2], Philippe Lahire[1], Sabine Moisan[3], and Jean-Paul Rigault[3]

[1] University of Nice Sophia Antipolis, I3S Laboratory (CNRS UMR 6070), France
{acher,collet,lahire}@i3s.unice.fr
[2] SINTEF, Oslo, Norway
franck.fleurey@sintef.no
[3] INRIA Sophia Antipolis Mediterranée, France
{moisan,jpr}@sophia.inria.fr

Abstract. Self-adaptive and dynamic systems adapt their behavior according to the context of execution. The contextual information exhibits multiple variability factors which induce many possible configurations of the software system at runtime. The challenge is to specify the adaptation rules that can link the dynamic variability of the context with the possible variants of the system. Our work investigates the systematic use of feature models for modeling the context and the software variants, together with their inter relations, as a way to configure the adaptive system with respect to a particular context. A case study in the domain of video surveillance systems is used to illustrate the approach.

1 Introduction

Dynamic Adaptive Systems (DAS) are software systems which have to dynamically adapt their behavior in order to cope with a changing environment. A DAS needs to be able to sense its environment, to autonomously select an appropriate configuration and to efficiently migrate to this configuration. Handling these issues at the programing level proves to be challenging due to both the large number of contexts and the large number of software configurations which have to be considered. The use of modeling and the exploitation of models at runtime provide solutions to cope with the complexity and the dynamic nature of DAS [1].

DAS have similarities with Software Product Lines (SPLs). The basic idea of SPL engineering is to design and implement a products family from which individual products can be systematically derived [2]. An SPL is typically specified as a set of variation points together with their alternatives. The products are derived by selecting different sets of alternatives associated to the variation points. SPLs are usually modeled using feature models which provide an intuitive way for stakeholders to express the variation points, alternatives and constraints between these alternatives. Depending on how the feature models are linked to the SPL artefacts (e.g. models), the product derivation can be more or less automated.

Just like an SPL, an important characteristic of a DAS is its variability. SPL techniques can be applied to model the variability in a DAS but do not provide

any solution to model *when* the system and *how* the appropriate configuration should be chosen according to the environment. To tackle this issue, an emerging approach is to use *Dynamic* SPLs [3] which try to achieve the self-modification of a system by dynamically (re)binding variation points at runtime. Modeling a DAS not only requires the modeling of variability in the system. It also needs models of its environment and some adaptation rules that specify which configuration of the DAS should run in each specific context.

The contribution of this paper is to propose an approach based on SPL techniques not only for specifying the variability in the DAS but also the variability in its environment. The idea is to model the DAS and its environment as two different SPLs and then to link them in order to capture adaptation. In practice, the DAS and its environment are modeled using two independent feature models. They are then connected by dependency constraints that specify how the DAS should adapt to changes in its environment. The use of feature models allows one to present homogeneously the system, its environment and the associated constraints. The feasibility of the approach has been evaluated through a case study of a digital video processing application. To ensure its usability, the proposed approach has been built on top of the Domain Specific Modeling Language (DSML) for adaptive systems proposed in [4].

The paper is structured as follows. Section 2 first introduces the DSML-based approach on top of which the contribution is built, then recalls the basics of feature models. It also introduces our case study and finally details how the proposed approach leverages feature models to revisit the DSML-based approach. Section 3 details how feature models are used to model the context, the adaptive system and the adaptation rules. Section 4 compares the approach to related work. Section 5 concludes and addresses future work.

2 From DSML to Feature Models

2.1 A DSML for Modeling Adaptation

In [5], the authors propose to combine the use of models at runtime and aspect-oriented modeling techniques to implement DAS. The proposed approach is based on an adaptation model connected to a set of alternatives implemented as aspects. At runtime, a reasoning engine processes the adaptation model and selects the functionality which matches the context. The adaptation of the system is implemented by weaving the corresponding aspects in a model kept causally connected with the running system. The adaptation model is made of four main elements:

Variants They make references to the available variability for the application. Depending on the complexity of the system, it can be a simple list of variants or a data structure like a hierarchy.

Constraints They specify constraints on variants to be used over a configuration. For example, the use of a particular functionality (variant model) might require or exclude others. These constraints reduce the total number of configurations by rejecting invalid configurations.

Context The context model is a minimal representation of the environment of the adaptive application to support the definition of adaptation rules. It only considers elements of the environment relevant for expressing adaptation rules. These elements are updated by sensors deployed on the running system.

Rules These rules specify how the system should adapt to its environment. In practice, these rules are relations between the values provided by the sensors and the variants that should be used.

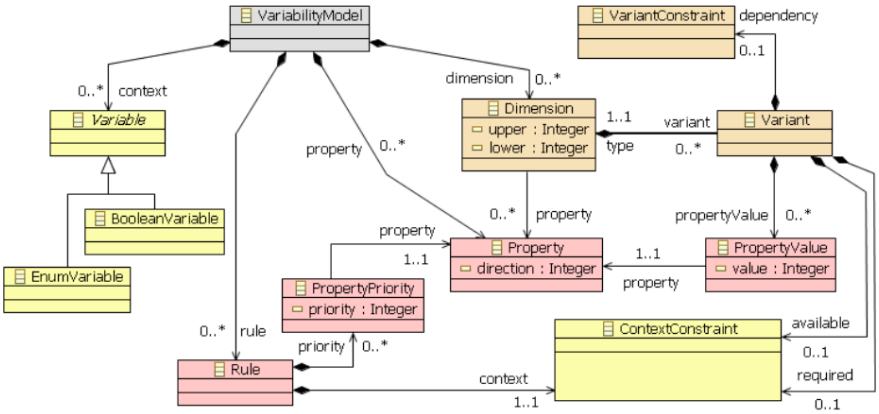


Fig. 1. Excerpt of the adaptation DSML

Until then, two different formalisms have been experimented for capturing the adaptation rules. In [5], the adaptation model is based on event-condition-action (ECA) rules. In [4], the adaptation model combines a set of hard-constraints and optimization rules. Both of these approaches rely on a model capturing the variability in the system and the variability in the context. Fig. 1 presents an excerpt of the adaptation DSML used in the last approach [4]. The root class for the DSML is *VariabilityModel*. On the left, it contains a set of variables which model the context of the DAS. On the right, it contains a set of *Dimensions* and *Variants* which models the variability in the system. The classes *VariantConstraint* and *ContextConstraint* are used to express hard-constraints between the system and its context. Finally, the classes *Rule* and *Property* are used to choose the best solution among the acceptable configurations in a particular context.

2.2 Feature Models

Feature models (FMs) are perhaps the most common formalism used to model SPL commonalities and variabilities [6,7,8]. A FM can capture different kinds of variability, ranging from high-level requirement variability to software variability. SPL variants are configured by selecting a set of features that satisfy FM constraints. Every *member* of an SPL is represented by a unique combination

of features and a FM compactly defines all features in an SPL and their valid combinations. It is basically an AND-OR graph with constraints which organizes hierarchically a set of features while making explicit the variability.

A *configuration* of a FM is a set of concrete features. An SPL is the set of all configurations that are valid for the FM which represents the SPL. A configuration is valid if the selection of all features contained in the configuration and the deselection of all other concrete features is allowed by FM. The semantics of FM is defined as follow: *i)* if a feature is selected, so should be its parent; *ii)* if a parent is selected, all the mandatory child features of its And group, exactly one of its Xor group(s), and at least one of its Or group(s) must be selected; *iii)* cross-tree constraints relating features (e.g. feature dependencies) must hold. For brevity's sake, we do not exemplify here FMs; the reader will find examples in Section 3.

2.3 Video Surveillance Case Study

Throughout the paper we propose to compare the systematic use of FMs with the DSML-based approach for modeling the adaptive system with a case study of Video Surveillance (VS) systems.

The purpose of VS is to analyze image sequences to detect interesting situations or events. The corresponding results may be stored for future processing or may raise alerts to human observers (detecting intrusion, counting objects or events, tracking people or vehicles, recognizing specific scenarios, etc.). At the implementation level, a typical VS processing chain starts with image acquisition, then segmentation of the acquired images, clustering to group image regions into blobs, classification of possible objects, and tracking these objects from one frame to the other. The final steps depend on the precise task (see Fig. 2).

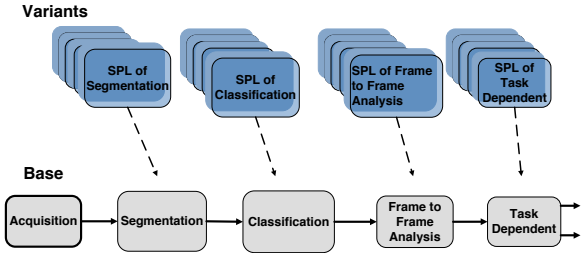


Fig. 2. A processing chain of the VS system and the SPLs

An important issue is that each kind of task has to be executed in a particular context. This context includes many different elements: information on the objects to recognize (size, color, texture, etc.), nature and position of the sensors (especially video cameras), etc. The number of different tasks, the complexity of contextual information, and the relationships among them induce many possible variants at the specification level, especially on the context side. The first activity of a VS application designer is to sort out these variants to precisely specify

the function to realize and its context. In our case study, the underlying software architecture is component-based. The processing chain consists of any number of components that transform data before passing it to other components. As a result, the designer has to map this specification to software components that implement the needed algorithms. The additional challenge is to manage the dynamic variability of the context to cope with possible runtime change of implementation triggered by context variations (e.g. lighting conditions, changes in the reference scene, etc.).

2.4 Revisiting the Approach with Feature Models

In comparison with the DSML approach previously described [4], we propose a more intuitive and more compact notation for the adaptation models. One of the major benefits of the DSML is to provide the ability to simulate and validate the adaptation model at design-time. Part of our objective is thus to keep a similar expressiveness so that existing simulation and validation techniques can be reused.

The key idea is to model the context and the software variants as two families (i.e. SPLs). The context model is represented as an SPL of context where each member of the SPL describes one valid state of the context. The software system is also an SPL and should adapt itself with respect to a contextual information. In SPL terminology, adaptation to context changes corresponds to product derivation or product configuration (i.e. the choice of a member of the SPL). In our case, we use a FM both for modeling the SPL of context and the set of possible variants. As previously described, the specification of constraints between features is possible. The constraints on variants can thus be directly expressed in the FM formalism. Moreover, the inter relations between context elements and software variants specify the adaptation rules. They are also represented with constraints between features of the context FM and features of the software variants FM. Using FMs, a configuration of the context corresponds to the actual contextual information where the software operates. At runtime, context changes mean that the context FM has a new configuration. A configuration of the software variants is an effective software system considering that all variants are then comprehensively integrated. The concept of configuration in FM formalism also helps to clarify the semantics relation between the two models: each configuration of the context should correspond to a configuration of the software system.

3 Modeling Context and Adaptation

3.1 Modeling Software Variants

All steps of the VS processing chain correspond to software components that the designer must correctly assemble to obtain a processing chain. A mandatory task is to acquire images. Then, for each step, many variants exist, along different dimensions (see Fig. 2). The first challenge is thus to model the software variants.

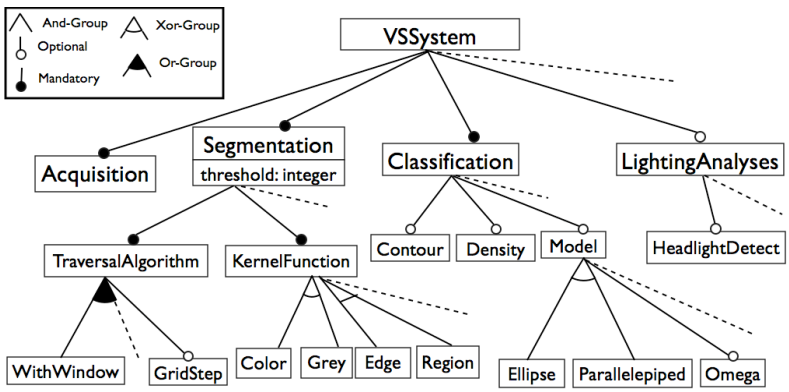


Fig. 3. SPL of the VS system

For instance, there are various Classification algorithms with different ranges of parameters, using different geometrical models of physical objects, with different strategies to identify relevant image blobs. Another example is depicted in Fig. 3 where the subtree of the FM whose root is Segmentation represents a family of segmentation algorithms. For each combination of subfeatures of Segmentation, we assume that there is a corresponding component. As an example, if TraversalAlgorithm, Grid Step, With Window, Kernel Function, Edge, Color features are selected, a fully parameterized component can be derived. Another component would be derived if the Region and Grey features are selected (instead of Edge and Color). Additional constraints are used to express dependencies between features:

$$\text{GridStep } \textbf{or } \text{WithWindow } \textbf{excludes } \text{Edge} \qquad (C_1)$$

$$\text{GridStep } \textbf{excludes } \text{Ellipse} \qquad (C_2)$$

$$\text{Edge } \textbf{excludes } \text{Density} \qquad (C_3)$$

The constraint C_1 means that if the features GridStep or WithWindow are selected, then it is not possible to select Edge. Note that the constraints do not necessary relate features of the same kind of algorithms (e.g. the constraint C_2 states that if the feature GridStep of a segmentation algorithm is selected, then the feature Ellipse of a classification algorithm is not selected). These constraints are expressed in propositional logics and correspond to the hard-constraints defined in the DSML (see Section 2.1). (A **excludes** B is a shortcut to express A **implies not** B).

Models at runtime may deal with values, which can be for example provided by sensors. To be able to handle them we need to use an extended formalism of basic FMs that propose adding extra-functional information to the feature using attributes [9]. A feature attribute has a a domain and possibly an assignment value when the feature is selected. For instance, *threshold* is an attribute of the feature Segmentation whose type is an integer in Fig. 3. Another example is the attribute of the feature TimeOfDay which can take the value *night* or *day* in Fig. 4. An attribute corresponds to the *Property* concept defined in the DSML.

3.2 Modeling Context

At present, there is a set of configurations available for each category of component in order to achieve each task of the processing chain. The contextual information is needed to reason at runtime on the effective choices of components. For instance, lighting changes can have an impact on the parametrization (i.e. configuration) of some components of the processing chain. Our approach is to represent the context model also as a FM. In Fig. 4, a contextual information that needs to be defined is the Objects of interest(s) to be detected, together with their properties.

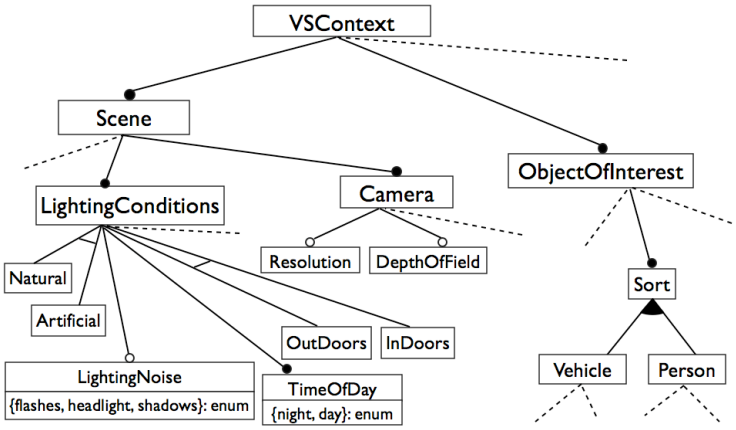
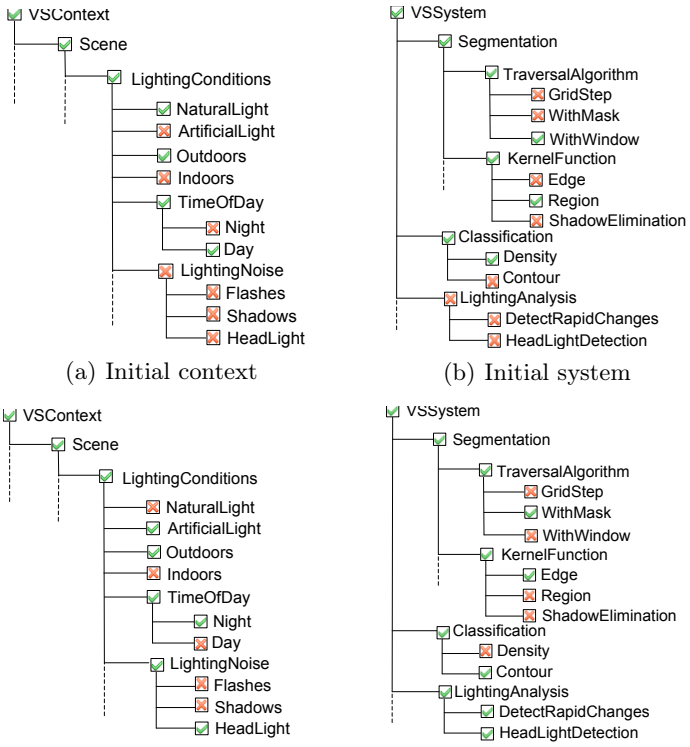


Fig. 4. SPL of the context

Then, Scene is the feature with the largest sub-tree; it describes the scene itself (its topography, the nature and location of cameras) and many other environmental properties (only some of them are shown on the figure). The elements of the FM may be related together with intra constraints which can reduce the configuration space.

3.3 Modeling Adaptation

The purpose here is to define the adaptation logics that defines the behaviour of the software system considering the context. In our approach and terminology, it means that the SPL system should change its configuration. As the available context information is also a configuration that affects the SPL configuration, we propose to inter relate the two FMs with adaptation rules. Adaptation rules are defined with the constraint language of FMs (i.e. propositional logic-based language). Abstract syntax rules consist of a Left hand side (LHS) and Right hand side (RHS). Both of them address features possibly connected with “and”, “or”, “not”. With this mechanism, simple ECA rules can be expressed. The LHS represents the condition part and is an expression based on the context information. The action is a change in the configuration of variants (RHS of the rules).



(a) Initial context (b) Initial system

(c) New context (d) SPL after reconfiguration

Fig. 5. Configurations of the VS system with respect to context changes

As an example, the following adaptation rule:

$$\text{Night } \textbf{and } \text{HeadLight } \textbf{implies } \text{HeadLightDetection} \qquad (AR_0)$$

states that if the contextual information describes that the Night and HeadLight are active, then the feature HeadLightDetection which corresponds to a component of the platform is integrated in the software system.

Some other adaptation rules are defined as follow:

$$\textbf{not } \text{LightingNoise } \textbf{implies } \text{Region} \qquad (AR_1)$$

$$\text{LightingNoise } \textbf{implies } \text{Edge} \qquad (AR_2)$$

$$\text{ArtificialLight } \textbf{implies } \text{DetectRapidChanges} \qquad (AR_3)$$

$$\text{Flashes } \textbf{or } \text{HeadLight } \textbf{implies } \text{Contour} \qquad (AR_4)$$

The Fig. 5(a) depicts an excerpt of the initial context configuration where the VS system operates. (The green icon in the box states that a feature is selected while the red cross means that the feature is deselected). One can notice that the system is executed in an Outdoor environment during the Day. The corresponding configuration of the VS system is represented in Fig. 5(b). As the feature LightingNoise is not selected in the context, the feature Region is selected in the VS system applying the rule AR_1.

An adaptation of the VS system is required with respect to context changes depicted in Fig. 5(c). Here, the system should run during the Night and the

light is now Artificial. Additionally, the HeadLights (e.g. of the vehicles) should be taken into account. Applying the different rules (AR_0, AR_2, AR_3 and AR_4), the adaptive system is then configured (see Fig. 5(d)).

4 Related Work

Feature Modeling. As in our running example, a few other approaches use multiple FMs during the SPL development. Kang et al. define four layers, each containing a number of FMs [6]. The paper discusses these layers and their FMs on a structural level and provides guidelines for building FMs. Some layers defined in their papers, like *operating environment* or *capability*, can be part of the contextual information. In [7], FMs are used to model decisions taken by different stakeholders at different stages of the software development. Hartmann et al. introduce context variability model (CVM) to represent context information of software products [10]. The combination of the CVM and another FM results in a so-called Multiple Product Line FM. All above-mentioned papers do not explicitly present their work as suited to self-adaptive or dynamic systems. The configuration of the FMs is rather static and does not evolve over time. In [11], Fernandes et al. propose to model context-aware systems. As in our approach, the authors use FMs to represent the context and the software system. A domain-specific language is also designed and allows the developer to specify context rules. Nevertheless, the formalism and notation used to represent FMs are not standard ; their work can be seen as an hybrid approach between the DSML-based and the FM-based approaches.

Adaptation Modeling. Beyond the approach presented in [4] and discussed in the beginning of the paper, a number of techniques have been proposed in the literature to capture and express adaptation. These techniques can be categorized under two families. The first one is the most commonly used and is based on event-guard-action type of rules [5, 12]. In these approaches the context and the configurations are related by a set of rules, which express how the evolution of the context should affect the running configuration of the application. The second family of approaches relies on the optimization of some utility functions [13, 3] associated to the system. In these approaches, changes in the environment trigger an optimization process that evaluates possible alternative configurations and adapt the system to maximize the utility of the running configurations. The FM notation proposed in this paper is very well suited to represent event-guard-action type of rules as these rules can be captured as constraints across the FMs. For optimization based-techniques, a possible solution would be the use of attributes in the FMs in order to represent the information needed by the optimization process. This will be investigated as future work.

5 Conclusion and Future Work

This paper addresses the reconfiguration of Dynamic Adaptive Systems and proposes to reason at the model level in order to compute the new configuration according to context changes. We have shown that the concepts in the DSML are expressible with the FM formalism. The concept of configuration is naturally present in FMs: the valid combination of variants or context elements is

defined by the semantics of the FMs. As a result, there is no need to define an ad-hoc semantics or constraint checking techniques. The context elements are no longer represented as Boolean variables and the user can structure hierarchically domain concepts. Besides, it is possible to express constraints between the elements of the context model, invariants and adaptation in the FM formalism. The uniform representation of the context model and the software system makes possible to express relations between the two models. The DSML-based approach and the FM-based approach can complement each other. On the one hand, the FM-based approach can take advantage of simulation and validation checking proposed in [5, 4]. On the other hand, the DSML-based approach can use the infrastructure, tools, techniques, etc. associated to FMs. As part of our future work, we intend *i)* to address the validation and simulation directly at the level of FMs and *ii)* to leverage the expressiveness of the FM-based approach (e.g. using attributes). We also plan on achieving a better separation of concerns thanks to a set of operators allowing to extract a subset of the context model; we believe that the context that may influence the runtime execution of the system is most of the time only a part of it. Finally, our long term goal is to connect state-of-the-art adaption engines to our models and provide an end-to-end software solution.

References

1. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J., Solberg, A., Dehlen, V., Blair, G.: An Aspect-Oriented and Model-Driven approach for managing dynamic variability. In: Model Driven Engineering Languages and Systems conference. (2008)
2. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag (2005)
3. Hallsteinsen, S., Stav, E., Solberg, A., Floch, J.: Using product line techniques to build adaptive systems. In: Software Product Line Conference. (2006)
4. Fleurey, F., Solberg, A.: A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In: Model Driven Engineering Languages and Systems conference. (2009)
5. Fleurey, F., Delhen, V., Bencomo, N., Morin, B., Jézéquel, J.M.: Modeling and validating dynamic adaptation. In: Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS'08, Toulouse, France (oct 2008)
6. Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: Form: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering **5**(1) (1998) 143–168
7. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration through Specialization and Multilevel Configuration of Feature Models. Software Process: Improvement and Practice **10**(2) (2005) 143–169
8. Batory, D.S.: Feature models, grammars, and propositional formulas. In Obbink, J.H., Pohl, K., eds.: SPLC. Volume 3714 of LNCS., Springer (2005) 7–20
9. Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Automated reasoning on feature models. LNCS, CAiSE 2005 **3520** (2005) 491–503
10. Hartmann, H., Trew, T.: Using feature diagrams with context variability to model multiple product lines for software supply chains. In: SPLC '08, IEEE (2008) 12–21
11. Fernandes, P., Werner, C.M.L.: Ubifex: Modeling context-aware software product lines. In Thiel, S., Pohl, K., eds.: SPLC (2), Limerick, Ireland (2008) 3–8
12. Zhang, J., Cheng, B.H.C.: Specifying adaptation semantics. In: WADS '05: Proceedings of the workshop on Architecting dependable systems, ACM (2005) 1–7
13. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjorven, E.: Using architecture models for runtime adaptability. IEEE Softw. **23**(2) (2006) 62–70

Statechart Interpretation on Resource Constrained Platforms: a Performance Analysis

Edzard Höfig[1], Peter H. Deussen[1], and Hakan Coşkun[2] *

[1] Fraunhofer Institute for Open Communication Systems, Berlin, Germany
{edzard.hoefig|peter.deussen}@fokus.fraunhofer.de
[2] Faculty IV, Department of Design and Testing of Communication-Based Systems, Technical University of Berlin, Germany coskun@cs.tu-berlin.de

Abstract The statechart formalism allows for the specification of behaviour models of complex, reactive systems. It is employed in the embedded systems domain to specify and verify applications at design time. By enabling the interpretation of formalised behaviour models one earns the favourable abilities of application behaviour inspection, control, and substitution at runtime. One of the major arguments against such an approach concerns poor interpretation performance and high-resource overhead. We are answering this argument by showing that it is feasible to implement a statechart interpreter on a resource-limited platform. We define the utilised statechart formalism and use it as a base for implementing a resource-efficient interpreter on a 8bit microcontroller with 2 kByte RAM. Performance overhead of key aspects of the interpretation engine is evaluated using suitable behaviour models and by comparison with compiled code.

Key words: Statechart, Interpretation, Behaviour Model, Performance Analysis

1 Introduction

Professionals are modelling reactive systems using statecharts for the purpose of system analysis and quality assurance at system design time. Recently there is an interest in using such formalised behaviour models to control and describe system behaviour at runtime. For runtime evolution of software and communication protocols such an approach has potential advantages over the direct generation of system code. Take for example the dynamic re-configuration of embedded systems without a firmware "flashing" procedure or shutdown, the possibility to trace a system state with only minimal performance overhead, or the application of formal validation methods at runtime to assure that the system is in a valid state. Although these properties are of interest for researchers and practitioners

* The authors would like to acknowledge the European Commission for funding the Integrated Project EFIPSANS "Exposing the Features in IP version Six protocols that can be exploited/extended for the purposes of designing/building Autonomic Networks and Services" within the 7th IST Framework Program.

in the autonomic communication and networking field, the resource usage of the interpretation mechanisms is critisised. We often heard the argument that the performance overhead renders the approach unsuitable for resource-constrained platforms. Contrary to this, we believe that such an approach is feasible even for embedded hardware platforms. As we found no hard numbers on interpretation performance, we decided to conduct a study on the performance of a statechart interpreter on a resource-constrained platform. Our challenge is two-fold: Firstly, we created a proof-of-concept implementation of such an interpreter. Secondly, we quantified the performance, enabling us to give a well-grounded answer to the performance argument.
The paper is structured as follows. We describe related work in section 2, and give a formal definition of statecharts in section 3. In section 4 we detail the mapping of our definition to a runtime mechanism. Subsequently, section 5 describes the performance characteristics obtained by measuring the implementation. We conclude with a discussion of our findings in section 6.

2 Related Work

Statecharts were invented more than 20 years ago by D. Harel [1] and are in widespread use as part of UML2 state diagrams. An example for a statechart-based behaviour model of a simplified car door and passenger room light is depicted in Figure 1.

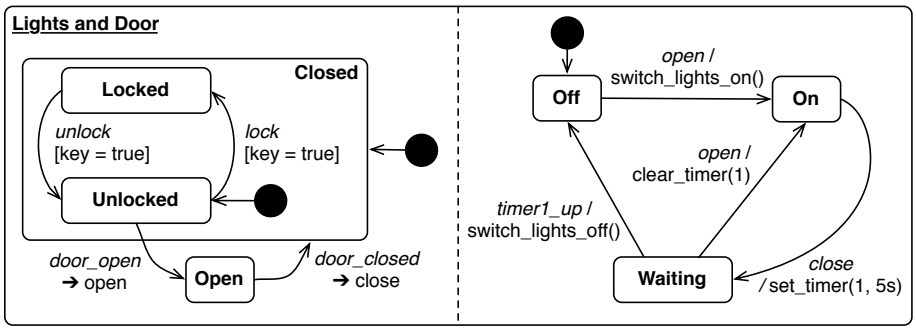


Figure 1. Example behaviour model "Lights & Door"

Employing statecharts for model checking and code generation is common practice [2,3] and the efficient interpretation of statecharts has also been researched by J. Ebert from a theoretical point of view [4]. First practical usage of interpreters for similar formalisms emerged in the last years out of the business process field [5]. An execution standard for statecharts is specified by the W3C under the name "State Chart XML"[6] with two implementations[3] available.

[3] A Java version from the Apache Software Foundation (`http://commons.apache.org/scxml`) and a C++ engine from QT Labs (`http://qt.gitorious.org/qt-labs/scxml`)

In our current work we use statecharts for network and system management [7] with the goal of equipping network routers with the ability to make autonomic decisions on an incoming packet stream by interpreting behaviour models [8]. We also have prior experience with optimisation of Extended Finite State Machines (EFSM) based automatons for analysing large XML data streams [9] and some of the optimisations that we are employing were discovered during previous work.

3 Formal Definition of Statecharts

Our formalism follows the definition specified in the annex of the original research paper [1]. We left out the "History Connector" definition, but apart from that we have a full-fledged statechart, including aggregate states and parallel components.

We define a *higraph* as a structure $H = \langle S, E, s_0, \theta, \sigma \rangle$, where S is a finite set of *symbolic states*, $E \subseteq S \times S$ is a set of *edges*, $\sigma : S \to 2^{2^S}$ is a *substate function*, and $s_0 \in S$ is a *root state*. We call each set of states $Q \in \sigma(s)$ a *component* of s. Distinguished components $Q_1, Q_2 \in \sigma(s)$ are called *parallel* to each other. A state $s \in S$ is called *atomic* if $\sigma(s) = \emptyset$ does hold, otherwise it is called *composed*. Moreover $\theta : \Theta \to S$ assigns a unique *start state* to each component of H, where $\Theta =_{\text{def}} \bigcup_{s \in S} \sigma(s)$ is the set of *components* of H. We assume that $\theta(Q) \in Q$ for each $Q \in \Theta$, i. e., the initial state of a component is a member of that component.

We stipulate a number of restrictions; to this end, let us define $\sigma^+(s)$ to be the smallest set (w. r. t. $\subseteq$) of symbolic states satisfying $\bigcup_{Q \in \sigma(s)} Q \subseteq \sigma^+(s)$ and $s' \in \sigma^+(s) \Rightarrow \bigcup_{Q \in \sigma(s')} Q \subseteq \sigma^+(s)$.

Then we assume that:

1. $Q \in \sigma(s) \Rightarrow Q \neq \varnothing$, i. e., substates of s are non-empty sets (note that this does not imply that $\sigma(s) = \varnothing$, i. e., we allow atomic states).
2. σ is *non-cyclic*, i. e., $s \notin \sigma^+(s)$ for all $s \in S$.
3. The sets in $\sigma(s)$ are pairwise disjoint, i. e., $Q_1, Q_2 \in \sigma(s) \wedge Q_1 \cap Q_1 \neq \varnothing \Rightarrow Q_1 = Q_2$, for all $s \in S$ such that $Q_1, Q_2 \in \sigma(s)$;
4. that the whole higraph has a tree-like structure, i.e., $\sigma^*(s_1) \cap \sigma^*(s_1) \neq \varnothing \Rightarrow s_1 \in \sigma^*(s_2) \vee s_2 \in \sigma^*(s_1)$.
5. Finally, for the root state s_0 we assume that $\sigma^*(s_0) = S$, and s_0 is the only state with this property.

For the sake of notational simplicity we moreover define $\sigma^{-1}(s) =_{\text{def}} s'$ whenever $s \in \sigma(s')$ (note that the expression $\sigma^{-1}(s)$ is undefined for $s = s_0$).

Next, we define a *statechart* as a structure $C = \langle H, V, D, I, \iota, \omega, \gamma, \alpha \rangle$, such that $H = \langle S, E, s_0, \theta, \sigma \rangle$, is a higraph called the *skeleton* of C. V is a finite set of *variables*, D is a set of *data*, I is a set of *events* including the *empty event* ϵ, ι,ω : $E \to I$ are mappings assigning a *triggering event* $\iota(e)$ and an *output event* $\omega(e)$ to each edge of H, respectively. Moreover $\gamma : E \to (V^D \to \{0, 1\})$ assigns a predicate to each edge of H. Here, V^D denotes the set of total mappings from V to D, i. e. all assignments of values from D to variables from V. Finally

$\alpha : E \rightarrow (V^D \rightarrow V^D)$ defines the effect of executing an edge $e \in E$ to an assignment $\rho \in V^D$.

A statechart describes a set of concurrent processes, where parallel processes are syntactically distinguished as substates $Q \in \sigma(s)$ of some high-level state $s \in S$. Hence we first need to define what a run-time state of statechart is. An *aggregated state* of a statechart C is a minimum (w. r. t. set inclusion) set $R \subseteq S$ such that

1. $s \in R \;\; \& \;\; s \neq s_0 \Rightarrow \sigma^{-1}(s) \in R$, i. e., if a state s is a member of an aggregated state, then its corresponding high-level state $\sigma^{-1}(s)$ is also;
2. $s \in R \wedge \sigma(s) \neq \varnothing \Rightarrow (\forall Q \in \sigma(s))|R \cap Q| = 1$, i. e., if s is a member of R, then R contains exactly one state from each component of s.

Note that by definition we have $s_0 \in R$ for each aggregated state R. Moreover, there is a uniquely defined initial aggregated state for each statechart C, namely the aggregated state R_0 with $\theta(Q) \in R_0$ for all $s \in R_0$ and $Q \in \sigma(s)$.

In order to fully describe the run-time behaviour of a statechart, we further need to take into account its current variable vector. Hence, *run-time states* are tuples of the form $\langle R,\rho \rangle$, where R is an aggregated state and $\rho \in V^D$ is a variable assignment. Let us denote the set of run-time states of C by Σ.

Now we are ready to define the behaviour of a statecharts in terms of transitions leading from one run-time state to another. To this end, we define a partial transition relation $\xrightarrow{a,b} \subseteq \Sigma \times \Sigma$ for each pair of events $a, b \in I$:

$$\begin{aligned}
&\langle R_1, \rho_1 \rangle \xrightarrow{a,b} \langle R_2, \rho_2 \rangle \\
&\Leftrightarrow_{\text{def}} (\exists e = \langle s_1, s_2 \rangle \in E)\big[s_1 \in R_1 \wedge s_2 \in R_2 \\
&\qquad\qquad \wedge\, \iota(e) = a \wedge \omega(e) = b \wedge \gamma(e)(\rho_1) = 1 \wedge \alpha(e)(\rho_1) = \rho_2 \\
&\qquad\qquad \wedge\, (\forall s \in R_2 \setminus R_1)(\forall Q \in \sigma(s))\big\{Q \cap R_1 = \varnothing \Rightarrow \theta(Q) \in R_2\big\}\big]
\end{aligned}$$

This means, a transition from a run-time state $\langle R_1, \rho_1 \rangle$ to another run-time $\langle R_2, \rho_2 \rangle$ if R_1 and R_2 contain symbolic states s_1 and s_2, respectively, connected by an edge $e = \langle e_1, e_2 \rangle$ labelled with the input event a and the output event b. Moreover, the predicate $\gamma(e)$ applied to ρ_1 yields true, and ρ_2 is the result of applying the $\alpha(e)$ to ρ_1. Finally, the last line in the formula above ensures that if a component Q is newly introduced into R_2 by the transition, then R_2 contains its start state. Using these definitions we can now discuss the statechart interpreter implementation.

4 Mapping of the Formalism to a Runtime Mechanism

We implemented the interpreter using the C programming language on an Arduino Duemilanove test board with a 16MHz ATmega328P microcontroller. There are 32 KByte Flash and 1 KByte EEPROM non-volatile memory available, as well as 2 KByte of volatile SRAM. We are using the most simple mapping that still allows to show a working approach. Introduction of more complex features would greatly improve the usability of the devised mechanism, but add nothing substantial in terms of evaluating the runtime performance overhead.

4.1 The Behaviour Model

We abstained from defining a syntax for behaviour models and work directly with an Abstract Syntax Tree (AST) in-memory representation, which we suppose can be generated from any suitable representation format (e.g., UML2 state diagrams, or SCXML). For each model the complete AST data is allocated as a single chunk of memory and the AST structure is constructed with single-byte references to this data. Prior to interpretation, an additional *executor* structure is allocated that holds input and output event queues, as well as data structures for processing parallel components, and a reference to s_0 as the initial starting point for execution.

4.2 States and Data Space

We restrict S to contain up to 256 symbols encoded by the numbers 0..255. Each state is represented by a data structure containing fields that allow to bidirectionally navigate the substate tree spanned by σ. For performance reasons we separate the state data structure into a substate set, a set of parallel components, and an additional reference to a superstate. Additionally, the structure contains a set of references to outgoing edges and a so-called *flag* byte used to indicate state properties, e.g., θ is implemented as a single bit in the flag byte. Sets are generally implemented as byte arrays with an additional field that holds set size. For aggregated states, and states containing parallel components, it is necessary to evaluate θ to identify the start state of contained component(s), and to additionally create data structures that allow for pseudo-concurrent processing of parallel components.

The variables V are limited to a maximum of 246 read- and writeable entries per behaviour model and 10 additional global entries shared between all executing models. Variables are referenced by the numerical values 0..255, where the values 0..9 refer to global values and 10..255 refer to local ones. The data set D is limited to 8 bit integer numbers. There is no type system. When data values are evaluated within boolean expressions, we follow C conventions for assigning logical values: 0 corresponds to a logical "false", other values are "true".

4.3 Edges and Event Processing

E is implemented as a set of data structures, which contain a reference to a destination state, the triggering event assignment ι, and the output event assignment ω. There can be a maximum of 256 edges. Events are numbered from 0..255 and identified by their numerical value – 0 is the special "empty" event symbol ε. The edge structure also contains references to a guard condition predicate γ and an action mapping α. Due to parallel processing of edges it is possible for multiple events to be received during a single step of a model. Events are buffered for input and output in ring-buffers, limited to 10 elements.
The guard condition predicates γ need to be evaluated to decide if an edge should

be traversed (hence the name "guard condition"). They are specified within the model AST and can be constructed from variable or constant references (notationally depicted using a $ sign), boolean operators (!, $\wedge$, $\vee$), and comparison operators ($=$, $<$, $>$, $\leq$, $\geq$). Evaluation precedence is implicitly given through the AST hierarchy.
The action bindings α are implemented as code that is statically bound to the runtime mechanism before the interpretation of a behaviour model commences. An action binding is a conventional function call with an arbitrary number of input and output parameters, and represents fixed capabilities of a device that are orchestrated using statechart logic. It is implemented by a structure holding a function pointer plus an ordered set of variable references. Parameters need to be de-referenced inside of the action function and can be used to read or write the variable value. A specific set of actions considers timers. We created three timers that can be set with a delay value using `set_timer(id,delay)` to deliver the specific events 8..10 once the delay time passes. Timers can be cleared using `clear_timer(id)` which suppresses dispatching of the timer event.
Explaining the processing algorithm goes beyond the scope of this paper. For an understanding of the intricacies refer to the paper by J. Ebert [4]. In a nutshell: For each input event all active components in a statechart are evaluated for triggered edges. If a triggered edge has a matching guard condition, the assigned action is executed and an output event send. The state(s) are then changed and another evaluation iteration is run with the next input event. These steps are repeated until all active components reach end states. It is worth mentioning that all ε edges are traversed before the next input event is taken from the input queue. Also, specific handling functionality needs to be executed on entering and exiting parallel states to maintain data structures for active components.

5 Performance Analysis

We found that the experimental platform has sufficient resources for the statechart interpreter code, which uses less than 8 KBytes of Flash memory. In this section we describe the evaluation results using the experimental platform.
Latency measurements have been conducted using in-line timestamps, the slight delay that has been introduced by this is negligible for the overall result. The employed timestamping mechanism has an accuracy of approx. 4 μs. Stack memory measurements were conducted by dumping the stack pointer during runtime. Performance of these routines is uncritical as such experiments only measured memory consumption, not latency.

5.1 Memory Overhead

To analyse stack performance, we exercised the behaviour model shown in Fig. 1. We used the following sequence to measure the normalised[4] stack allocation as shown in Fig. 2: $key \leftarrow false$, *door_open*, *door_close*, *lock*, *door_open*,

[4] Showing only the additional bytes consumed during interpretation of the model

door_close, *key* ← *true*, *lock*, *door_open*, *key* ← *false*, *unlock*, *door_open*, *key* ← *true*, *unlock*, *door_open*, *door_close*, *wait for the lights to turn off*. The interpreter executes a single *step* method to iteratively advance the statechart. This method uses 25 bytes stack when processing input events and 23 bytes when processing ε events. Peaks in the stack usage are due to evaluation of the *key* guard conditions on the edges between the *Locked* and *Unlocked* states.

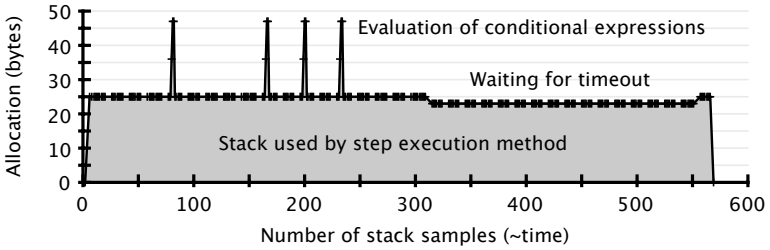


Figure 2. Stack usage during interpretation of Light & Door model

5.2 Conditional Expression Evaluation

The expression evaluator is implemented as a tree walker that recursively traverses a binary tree of statement tokens (variables, constants, and operators). We used three expressions to measure performance "$\$0 < 15$" (interpreted in 24 μs), "$(\$0 < 15) \wedge (\$1 = \$2)$" (56 μs), and "$((\$0 < 15) \wedge (\$1 = \$2)) \wedge ((30 > \$4) \vee (\$3 = \$5))$" (116 μs). This approach has a remarkable performance overhead: A hard-coded C version of any of these expressions executes in less than 4 μs. As seen in Fig. 2 the evaluation of conditional expressions is depicted as peaks in the stack usage. By sampling stack size during evaluation we found that the expression evaluator uses an additional 11 bytes per recursive iteration, e.g., Expression C uses a total of 44 bytes stack memory during evaluation.

5.3 Simple Edge Matching

We are measuring the time that our implementation needs to react with a single output event to a single input event using the traversal of a single edge. There can be more than one outgoing edge assigned to a single state, so we are also interested in the latency of the interpreter when processing multiple edges. We are using 30 behaviour models with an increasing number of edges for a single state. Each edge is triggered by a specific event 1..30 and sends a corresponding output event 101..130. Each model is then supplied with exactly one event, activating the edge that is triggered by the highest event. This is done to force the interpreter into exhibiting worst-case behaviour (it checks each edge before finding the edge that matches). The results, along with an illustration of the experimental models, can be seen in Fig. 3. To put the measurements into perspective, we also added the time that a conventional "switch" statement needs to deliver the same result.

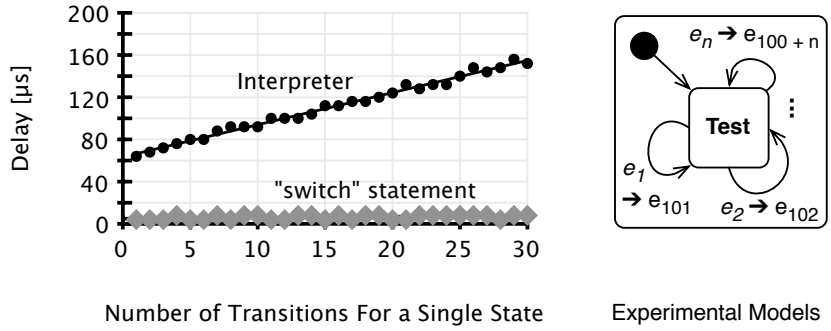


Figure 3. Delay of edge matching processes

The latency for a simple edge transition is approx. 64 μs, which includes event processing, timer handling, edge selection, and edge execution. It is approximately a factor 10 slower than a conventional switch statement which executes at around 6 μs. Latency increases linearly with approx. 3 μs for each edge up to 152 μs for 30 edges. The "switch" statement has a constant delay independent of the given event. The reason for the linear increase is the need to check each of the edges for a possible match.
The usage of dynamic action bindings instead of static function calls also has an impact on the latency of action execution due to the way parameters are passed to function code. We created models that trigger an action using a single edge from a single state and altered the number of parameters (0..10) passed to the action. The additional delay introduced amounts to an average of approx. 3 μs per additional parameter. For conventional function calls we believe that an additional delay exists as well, but we found that the measured latency differences are within the precision range of the employed timing mechanism for the number of arguments studied (delay differs < 4 μs).

5.4 Processing of Aggregates and Parallel Components

The two major features that differentiate statecharts from EFSM are aggregation and the ability to specify parallel components. To measure performance of aggregation handling we used a series of models with an increasing number of nested states (from a single state to an aggregate with a nesting depth of 30) where the most deeply nested state had an edge that matched on a given input event. The parallel component processing was analysed using 30 models which contained a superstate with an increasing number of parallel components, each triggering on the same input event. The results are displayed along with the experimental models in Fig. 4. We found it necessary to differentiate between the first input event and subsequent events[5] processed in the same state. This is due to additional functionality executed when entering an aggregated state or a state that contains parallel components. Fig. 4(A) shows an average delay of 12.5

[5] in the diagram labelled as 2^{nd} event, representative for all subsequent events

μs per additional nested state for an event that triggers entering the aggregate. Once the aggregate has been entered, the delay for processing subsequent events is independent of the nesting level. This is different for parallel components, as shown in Fig. 4(B). Entering a state with parallel components has an average latency of approx. 52 μs per parallel component. There is an average overhead of approx. 26 μs per active component for each subsequent event. To compare the latency overhead with conventional constructs, we also show the delay of a "for-loop" sequentially processing the input event. In this case, the overhead is at approx. 2 μs per additional iteration.

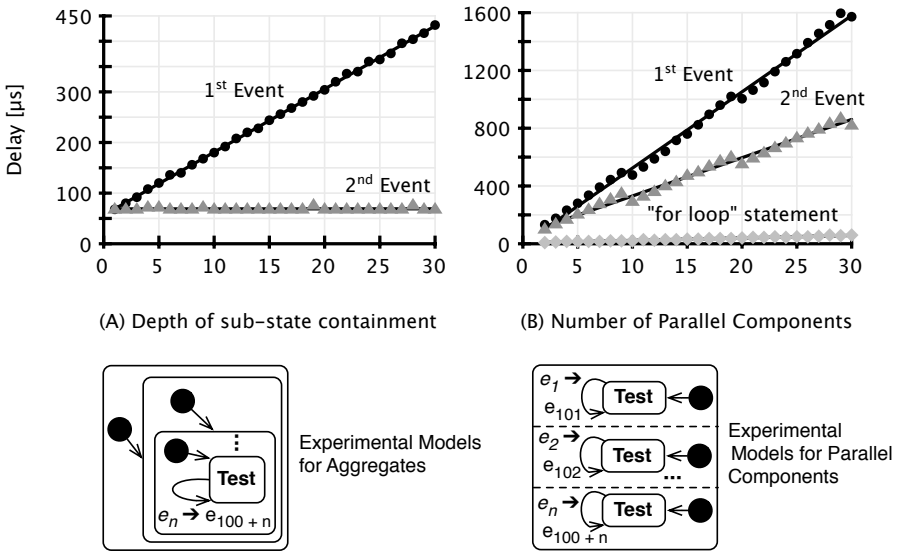


Figure 4. Event delay characteristics for processing of aggregated states (A) and parallel components (B) with the employed experimental models

6 Conclusion

The results confirm our initial assumption: It is possible to implement a statechart interpreter on a severely resource-constrained platform. We had no problem fitting the interpreter into non-volatile memory, though the available heap memory establishes clear constraints on the complexity of the behaviour models. Heap memory was large enough to hold any of the experimental models we applied for performance assessment, but we found that models with about 100 states are the limit. Stack memory is unlikely to be exhausted: Models would need to use a very deep conditional expression token tree.
On the performance side, we found that the interpreter clearly adds a processing overhead. In the best case execution latency is about a factor 10 longer than with compiled code. Performance depends largely on the structure of the interpreted

models, main factors are: the number of edges leaving a state, the nesting depth for aggregates, the number of parallel components, and the usage of guard conditions. Also, the ratio between the time the interpreter spends in action functions and the time spent in statechart interpretation plays an important role: If action functions are sufficiently complex, the overhead caused by the statechart engine is much smaller. On the other hand, if system behaviour is completely modelled using a statechart, the interpretation overhead becomes very large.

The interpreter performance can still be improved, mainly by using a better expression evaluator, but also by optimisation of the edge evaluation code (e.g., grouping triggering events, combining guard conditions). Even with ingenious optimisations, some overhead cannot be purged: In the worst case, any statechart interpreter needs to evaluate all outgoing edges for a single state, including the outgoing edges of parent states, and there will always be an overhead for event processing and handling aggregates, as well as parallel components. Therefore we conclude that our approach is adequate for reactive systems, which are idle most of the time. It does not seem to be suitable for systems that need the fastest possible reaction time due to the introduced interpretation delay, which can easily amount to 1 ms. Such a value is unacceptable for most real-time applications. Regarding high-throughput systems, successful applications should be possible but will depend on the underlying platform performance and the utilised behaviour model complexity.

References

1. Harel, D.: On visual formalisms. Communications of the ACM **31**(5) (May 1988)
2. Gnesi, S., Mazzanti, F.: On the fly model checking of communicating uml state machines. ACIS Int. Conf. on Software Engineering Research, Management and Applications (2004) 331–338
3. Raghunathan, B., Hartrum, T.: The automated transformation of statecharts from a formal specification to object-oriented software. 48th Midwest Symposium on Circuits and Systems (2005) 319– 322
4. Ebert, J.: Efficient interpretation of state charts. Fundamentals of Computation Theory **710** (Jan 1993) 212–221
5. Sánchez, M., Barrero, I., Villalobos, J., Deridder, D.: An execution platform for extensible runtime models. Proc. models@run.time workshop (2008) 107–116
6. W3C: State Chart XML (SCXML): State machine notation for control abstraction. http://www.w3.org/TR/scxml/
7. Höfig, E., P.H.Deussen: Document-based network and system management. Proc. 2nd International Conference on Autonomic Computing and Communication Systems (Jun 2008)
8. Höfig, E., Coskun, H.: Intrinsic monitoring using behaviour models in ipv6 networks. to be presented at the IEEE Modelling Autonomic Communication Environments (MACE) Workshop (October 2009)
9. Hinnerichs, A., Höfig, E.: An efficient mechanism for matching multiple patterns on xml streams. Proc. IASTED Int. Conf. on Software Engineering (2007) 164–170

Using Specification Models for RunTime Adaptations*

Sébastien Saudrais[1], Athanasios Staikopoulos[2], and Siobhán Clarke[2]

[1] Embedded Systems Laboratory, ESTACA, France sebastien.saudrais@estaca.fr
[2] DSG, Trinity College of Dublin, Ireland
{athanasios.staikopoulos,siobhan.clarke}@cs.tcd.ie

Abstract. For a myriad of reasons, modern applications face constant change to their requirements and working environment, requiring them to adapt accordingly. Increasingly, such adaptation is even required during runtime. In Model-Driven Engineering (MDE) approaches, models are first-class entities in the development of applications, though they have not, to date, been sufficiently taken advantage of in runtime adaptation specification. In many existing approaches, designers are required to consider the execution model when specifying any runtime adaptation, forcing them to understand the different formalisms of both the execution model and the specification model. The focus of this paper is to show how runtime models to monitor an application's execution can be derived efficiently from the specification, and how they support the designer in considering the application's execution in the same formalism as the specification.

1 Introduction

Model-Driven Engineering (MDE) promotes the use of models throughout the development of software. The underlying idea is to promote models as the primary artefacts of software development, making executable code a pure derivative of those models. Models containing adaptation specifications are an increasingly important and frequently encountered part of the development process. This is especially true for modern applications that need to adapt at runtime to cope with constant changes to their requirements and operating environments. Such changes have to be considered at the specification phase, and the models validated before they are transformed to real code. However, despite the importance of specification models, they have, to date, been ignored during the execution of the software. Once the code is generated, the specification models are no longer used with the potential loss of information that would be especially valuable during adaptation specification.

A further difficulty emerges during the process of adapting the execution. While the adaptation may be based on a specification model, the actual adaptation is necessarily performed either by hand on the application's code, or requires a complete regeneration of the system since it is unlikely to match the old specifications. Working directly with the application's code means a move to a different formalism from that of the specification. This change between formalisms has a number of disadvantages. The first is that the adaptations performed in the new formalism must be validated against those in the specification model. An automatic generation of the entire module that is to be adapted can ease this checking, but this needs to be coupled with a reverse-engineering technique to reproduce the specification model. The second disadvantage is that while the software architect knows the specification formalism, he is not always familiar with the implementation's one(s). If there

* This work has been carried out within the FP7 project ALIVE IST-215890, which is funded by the European Community. The authors would like to acknowledge the contributions of their colleagues from ALIVE Consortium (http://www.ist-alive.eu)

are adaptation problems at runtime, he has to be able to understand the second formalism in order to solve the errors, or work closely with an implementation team member. Either approach is likely to be difficult where an application's execution context often changes, requiring manual adaptation at runtime. It would be easier for the architect to visualise a snapshot of the actual execution in the specification formalism.

The approach proposed in this paper takes advantage of the specification models during the execution. A runtime model is generated from the specifications, which supports the monitoring of the execution required to supply sufficient information to apply adaptations directly on the specification models of the software. The runtime model contains the information needed to trigger the adaptation and is created based on the adaptations defined at the specification. At runtime, when an adaptation needs to be performed, the specification models are updated to correspond to the actual execution and the adaptation is performed on the up-to-date specification models. The new configuration of the application is, finally, generated from this new specification models. The approach allows a designer to use a single language (the specification language) to design the software and to interact with it during the execution. Our approach is applied within the ALIVE project[1], which is funded by the EU under Framework 7. ALIVE's objective is to enrich service-oriented architectures with coordination and organisation mechanisms often seen in human and other societies. The remainder of the paper is organised as follows: Section 2 presents the different metamodels and how runtime models are generated. Section 3 illustrates the approach with the ALIVE Crisis Management scenario. Section 4 compares our approach with related work and discusses the advantages of runtime models. Finally Section 5 concludes.

2 From Specification to Runtime

Our approach uses the adaptation rules defined during the specification directly when needed at runtime. For the purposes of this paper, we assume these adaptation rules have been proven during the specification of the application and are understandable by the architect. We use the adaptation rules to generate runtime models that will monitor the application and launch an adaptation when needed. Only a subset of the information contained in the adaptation rules is required to produce the runtime models. This subset is composed of the model elements that need to be monitored to trigger the adaptation and those that need to be read to perform the adaptation. An overview of the approach is presented in Figure 1. The runtime models are generated from the specification models and the enabling conditions of the adaptation rules. During execution, the runtime models monitor the application's code. When an adaptation is triggered, the adaptation rules and the runtime models are used to provide a snapshot of the application containing only the part involved in the targeted adaptation. The adaptation is then performed on the specification models obtained from this snapshot. A new configuration of the code is obtained from the new version of the specification models using the same code generation techniques used in the initial generation of the software. In the next section, we define a metamodel for runtime based on adaptation rules. An algorithm is then presented to automatically generate the runtime models. Finally we explain how the adaptation can be performed.

2.1 Adaptation Rules

Adaptations specify the appropriate reaction to changes that can occur at runtime and that have an impact on the software. An adaptation rule is composed of a

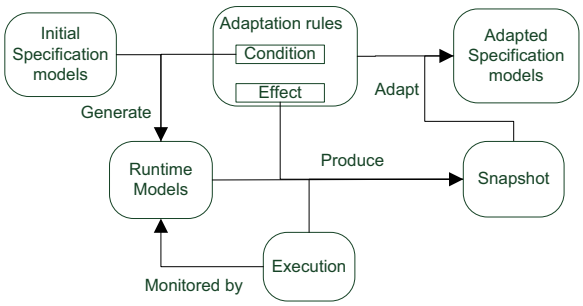


Fig. 1. Approach overview.

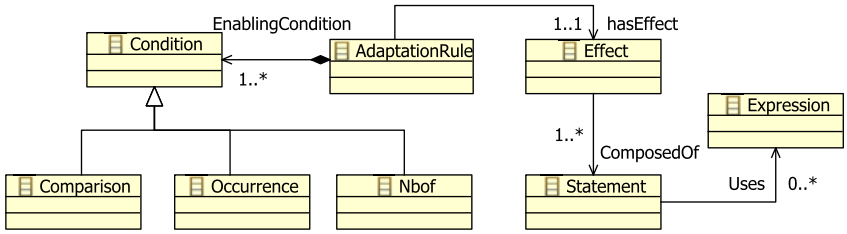


Fig. 2. Adaptation rules concepts.

condition and an effect. The condition contains the information triggering the adaptation: for example, an occurrence/absence of an event, a comparison of an object with a value or a number of occurrences of an event. The effect explains how the adaptation is performed and is written in the model transformation language used to specify the adaptation. It explains how the adaptation is applied and which part of the application is involved in the adaptation. Figure 2 presents the (simplified) metamodel of the adaptation rules in our approach. The condition is a superset of the possible conditions and can be extended by other types. The effect part only contains the expressions, i.e. the elements of the specification models involved in the adaptation. These elements will be manipulated and updated by the adaptation.

2.2 Runtime Models

Runtime models contain the information needed to support an adaptation when it must be performed. They link the implementation, the specification models and the adaptation rules. We have defined a generic metamodel to represent the different relations between these three elements. The runtime models have the same objective as the condition part of the adaptation rules: triggering the adaptation. As illustrated in Figure 3, the runtime metamodel has as base the metamodel of the adaptation rules relating to the conditions and is extended with information about the platform to monitor the software. The left part of the metamodel corresponds to the enabling condition and the right part to the link with the platform. Each *adaptation rule* has different *triggers* of the same type as the enabing condition and so can be extended with other types of conditions. The class *Element* references the elements of the specification model. For each element to be monitored, the corresponding implementation is obtained through an *AccessPoint*. The access point provides the means to access the value of the element in the implementation, for example, via a method to access the value or an exchange of messages. Only some of the possible types of access points are presented is the metamodel, *method* and

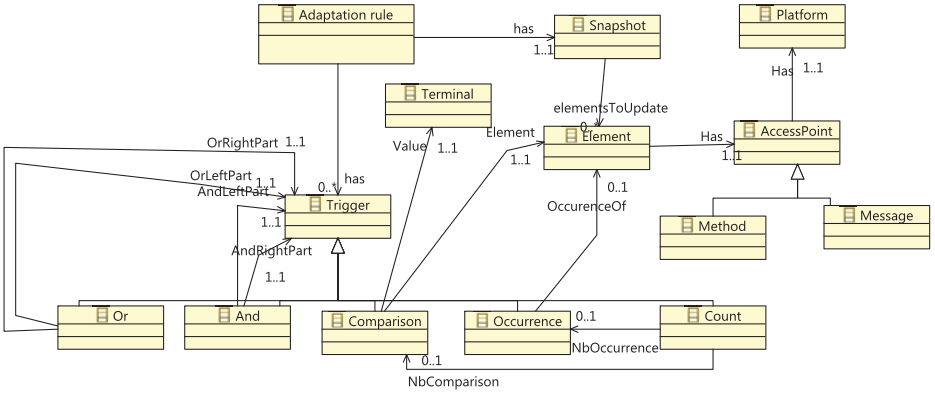


Fig. 3. Runtime metamodel.

message, but extensions can be easily made depending on the requirements of the software.

The runtime metamodel is also used for the snapshot through the class *snapshot*. The purpose of a snapshot model is to give an updated view of the software and links elements of the specification to the platform. It contains only a set of elements involved in the effect part of the adaptation corresponding to the right part of the Figure 3: the *Element* and *AccessPoint*.

2.3 Generation of the Runtime Models

Our approach includes an automated process to apply the adaptation rules on the specification models during the execution. The architect may also add new adaptation rules during the execution that will need to be incorporated in the runtime model without human intervention. The runtime model is automatically generated from the specification, the adaptation rules and the platform specifications. The generation algorithm has two steps. The first step is to select the different classes from the specification that are used by the adaptation rules. For each enabling condition, the set of elements required for monitoring is identified. The trigger is created using the enabling condition of each adaptation rules. The set of elements is then reduced to avoid duplicate elements. This step is designed to ensure that the runtime model contains only the elements required to support adaptation, and is therefore smaller and more efficient to process than would be a runtime model of the complete specification.

The second step is to identify the access point in the implementation. This step will use information from the specification and platform specifications. The access point is attached to the element in the runtime model and needs code to be generated before it can access the implementation. As software modules do not have a single implementation language, the different access points can be implemented in different languages. The runtime model is updated with values obtained through the access point during the monitoring process. The actual implementation of the runtime model is done using Kermeta [2]. Kermeta offers calls to Java classes with interfaces to other languages. For each access point, a Kermeta method is created with the intermediate code in Java, if needed, to make the link with the implementation. This access point can be regenerated at runtime if the access point is changing during the execution.

2.4 Adaptation at Runtime

Once the runtime model is generated, its monitoring capabilities are executed and the runtime models are automatically updated. When an adaptation is triggered, the specification models are updated with the actual values contained in the runtime model and a snapshot is created. The process of creating the snapshot is based on the same algorithm as the generation of the runtime models but where only the current adaptation's effect's expressions are considered. Once the snapshot of all useful information is created, the adaptation can be performed on the specification models using the adaptation rules. Once the adaptation is performed, the new implementation is generated using the same method as for the first generation of the implementation.

The architect can also use the snapshot process to create a visualisation of the actual execution. This visualisation may consider only a subset of the application and some adaptation rules. The snapshot process is used in this case to support the architect adding new adaptations that take account of the actual execution of the software. A new runtime model is then generated to incorporate the new enabling conditions of the added adaptation rules.

3 Evaluation: Crisis Management Case Study

In this section we show how runtime models are exploited in a use case from the ALIVE project that describes a crisis management scenario defined by Thales[3]. We first present a high-level summary of the specification used in ALIVE applications. We then apply our approach on the example.

3.1 ALIVE's Specification

Three metamodels describe the ALIVE layered architecture: organisation, coordination and services. Each one has a different level of abstraction and its own adaptation rules. Model transformations are defined from the metamodels and are bi-directional between the different layers.

The organisation level provides context for the two other levels, supporting an explicit representation of the organisational structure of the application. It presents the roles involved in the organisation and their inter-relations. Each role has a set of objectives for which it is responsible. The coordination level uses the organisation level as a starting point, and provides coordination plans to achieve the objectives of the organisation. As agents can play different roles in an organisation, the coordination metamodel has also the concept of actors capturing the goals of an agent playing a specific role. The coordination plans describes the interaction between the actors. For example, a payment objective will be refined by cash, paper payment or electronic payment. The service level supports the semantic description of services and the selection of the most appropriate service for a given task. It connects the executing environment and the two other levels, which are input to the service level. It contains agents and the different services. The agents are connected to the actors of the coordination. The services are refinements of the coordination actions, for example, the electronic payments become different services from each bank that offers an electronic payment.

The adaptation rules of the different levels are based on the occurrence of specific events or properties. An adaptation is triggered if certain conditions are verified. Properties from all three levels may trigger an adaptation to an ALIVE application. Depending on the level where the adaptation trigger occurs, the adaptation will have a different impact on the application. Adaptations affecting the service level

will be performed without impacting the two others. An adaptation that impacts the coordination level is also likely to impact the service level. An adaptation at the organisation level is likely to impact all three levels. The same language is used by the three levels to express the adaptations.

3.2 Initial Specification

The use case describes a system to handle emergency situations.The organisation includes a police station, first-aid station, emergency centre and fire station. The main objective of the fire station is to evacuate people. Other objectives of the different roles are to identify the emergency location, to provide an ambulance service and to regulate traffic. These objectives are delegated through the arrows to the other roles as depicted on the top part of Figure 4. The coordination level describes a plan to achieve the evacuation objective in different steps: selection of the transport vehicle, provision of an itinerary to the accident location, collection of injured people, provision of an itinerary to the hospital. This plan is a generic one that can be used and refined by the service level. The middle part of the Figure 4 shows the coordination level.

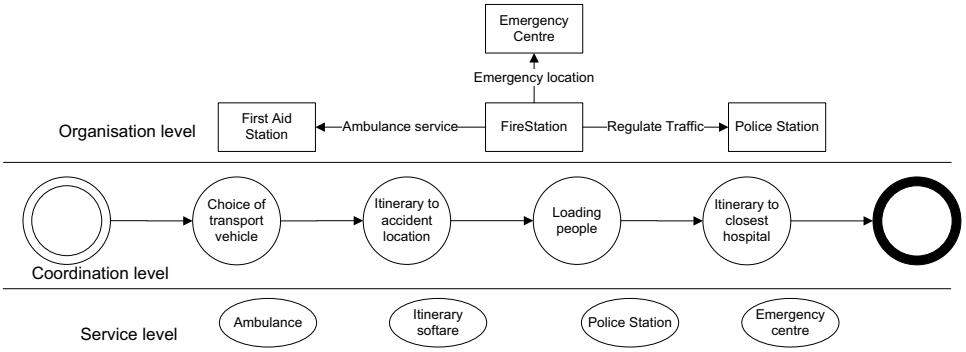


Fig. 4. Initial specification of the crisis management scenario.

During an accident, the fire station makes decisions relating to the evacuation of people. The evacuation plan is called at the service level. Specific services are used: an ambulance, the emergency centre, itinerary software and the police station. The bottom part of the Figure 4 shows the different services in play.

Adaptation rules are defined to handle common failures that can happen to this type of application: traffic jams, engine failure, escalation of the danger level. For example, a first adaptation may concern engine failure. Depending on the position of the ambulance and on the level of risk for rescued people, different choices can be made: ambulance change, people transfer or ambulance repair. This adaptation concerns only the service level. A second adaptation may concern a failure relating to difficulties encountered by the rescue personnel in achieving their objectives. The ambulance has a problem and no other terrestrial vehicle, as needed by the plan, is available. Alternative transport has to be considered, either by air or by sea and a new plan has to be given to the service level. This adaptation concerns both the coordination and the service levels. A last adaptation may be triggered when the coordination level is unable to find a new plan when the ambulance fails. The organisation level needs to adapt to the situation and may incorporate new roles. In this case, private companies can be added, like private helicopters, to evacuate people. While this adaptation will impact the three levels, some parts of each level can be reused, like the abstract plan and different services.

3.3 Runtime Models

The runtime model obtained from the specifications to support the second adaptation presented above is depicted on Figure 5. The enabling condition from the adaptation has the occurrence of the message *ambulance_blocked* and the occurrence of the properties *no_repairable* and *no_terrestrial_vehicule_available*. The trigger is added to the runtime model. The next step in the creation of the runtime model is to link with the implementation. For the purpose of the evaluation, we are using the Thales simulation workbench to simulate the different services. For each of the three elements, the corresponding access point is provided according to the platform specification. The ambulance provides its status and position through the methods *Ambulance_position* and *Ambulance_status*. The emergency centre provides the transports' availability through *Transport_Availability*. The methods are implemented in Java and interact with the workbench.

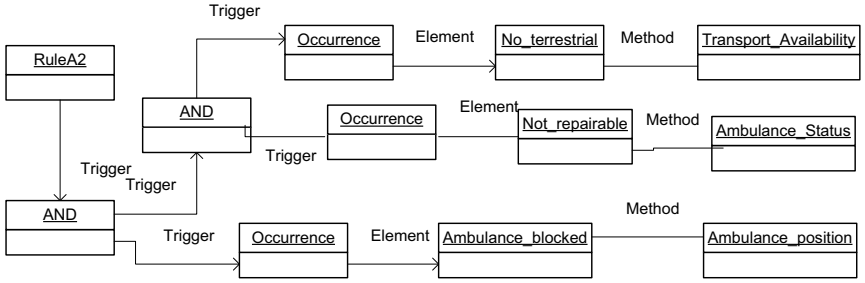


Fig. 5. Runtime model.

Once the adaptation is triggered, a snapshot of the part of the application of interest to the adaptation is made. The *create_plan_aerial_evecuation* call needs nothing at the coordination level as a new plan is created. Once the evacuation plan is created, the status of different aerial transport is needed to select one available to execute the plan. The snapshot contains two elements *helicopter* and their access point. The specification models are updated using both the runtime model and the snapshot model, and the adaptation is performed.

The new configuration is then produced from the adapted specification models as shown on Figure 6. The plan is modified and the services *helicopter1* and *helicopter2* are added.

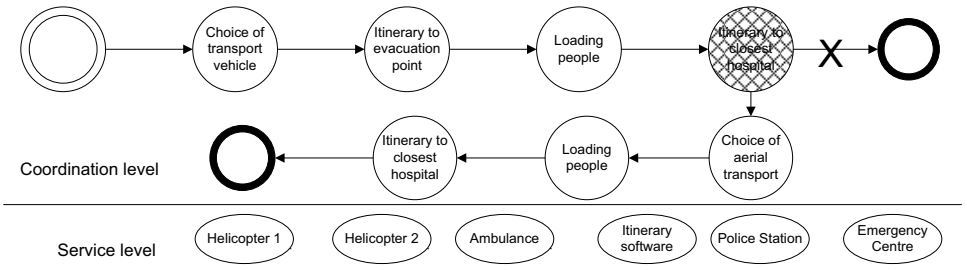


Fig. 6. New specification.

4 Discussion and Related Work

Discussion Our hypothesis related to the efficiency of this approach is based on an assumption that only a subset of the application is subject to adaptation. The approach generates a runtime model based on only those elements required to support adaptation, thereby reducing its size relative to the full application, making it more efficient to work with. Given this, our approach is therefore well-suited for applications where a big part of the specification is static (in other words, not expected to require adaptation over the execution of the application) and mainly used to understand the objectives of the application. A good example of this is ALIVE's organisation level. The static part of ALIVE applications do not, therefore, require permanent monitoring at runtime. In applications where adaptation rules cover a bigger part of the specification, the runtime model will be a correspondingly bigger proportion of the full specification, reducing the extent of the efficiencies. Further experiments are needed to identify the maximum coverage percentage that will still result in efficiency benefits in the monitoring process. The evaluation runtime model contains 10 elements to monitor when the specification models contain 50 elements. The snapshot models need an average of 10 elements to update.

A second potential limitation is the feasibility of performing the adaptation on the models at runtime. If the application is centralised, different transformation languages can be used but as modern applications are often distributed, including ALIVE applications, the adaptation may also be distributed. Few transformation languages focus on ensuring a light execution footprint, which may be problematic in a distributed setting. The current version of our runtime models is implemented using Kermeta but it requires at least a Java virtual machine. A more optimal approach would be a transformation language than can be interfaced with multiple implementation languages but without any constraints on the execution platform.

Related Work Many approaches adapt applications using a different formalism than the specification. In such approaches, the adaptation module can be seen as a runtime model because it has its own representation of the execution. However, the gap between the specification and the execution requires a re-test of the adaptation even though it has already been proven at the specification phase. For example, Pickering et al [4] propose an approach to manage complex systems with runtime models. The systems management is defined in specification models that are transformed to runtime models in a specific infrastructure, IBM WebSphere and so are expressed in a different language than the specification. Rainbow [5] provides an adaptation framework based on an abstract architectural model to monitor runtime properties to accommodate resource variability, system faults, etc. In our approach, runtime model is built on dynamic parts of the specification models and not on an abstract model to apply adaptations.

Other approaches are in a position to use the specification models at runtime because of the specific platform they provide. For example, Fractal [6] monitors the execution and performs the adaptation using the reflexivity of its own language. The ALIVE approach uses standard languages, and therefore assumes different languages at the implementation level. The Diva [7] approach considers both design and runtime phases of development. At design time, an application is modelled using a base model (containing the common/core functionalities), a set of variant models (capturing the adaptive application variability) and an adaptation model (specifying which variants should be used according the rules and current context of the executing system). At runtime, the models are processed by model composers that produce the system's configuration. The application is fully monitored and is based on the reflexivity of the underlying language.

5 Conclusion

In this paper, we presented an approach to using specification models to derive efficient runtime models that support runtime adaptation. We defined a metamodel for runtime models based on adaptation rules. Runtime models are automatically generated from the specification. Adaptation is performed at runtime using the specification models. The approach is designed to address two main objectives. This first is to use the same formalism for adaptation both at design and runtime. This reduces the potential for introduction of errors, by avoiding the transformation to another formalism, and aids the architect's understanding of the execution without requiring him to learn additional languages. The second objective is to optimise the efficiency of the runtime models. This is achieved as the runtime models monitor only the parts of the application that are involved in adaptation. A snapshot is taken of only those elements of interest to the adaptation. A full snapshot is available when the architect wants to have an overview of the system or wants to introduce new adaptation rules. The automation of the generation of runtime models supports this addition of new adaptation rules. We illustrated an evaluation of the approach through application on a case study.

References

1. ALIVE: Coordination, organisation and model driven approaches for dynamic, flexible, robust software and services engineering, http://www.ist-alive.eu/
2. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In L. Briand, S.K., ed.: Proceedings of MODELS/UML'2005. Volume 3713 of LNCS., Montego Bay, Jamaica, Springer (October 2005) 264–278
3. Aldewereld, H., Dignum, F., Penserini, L., Dignum, V.: Norm dynamics in adaptive organisations. In Boella, G., Pigozzi, G., Singh, M.P., Verhagen, H., eds.: NORMAS. (2008) 1–15
4. Brian Pickering, Sylvain Robert, S.M., Mengusoglu, E.: Model-driven management of complex systems. In: Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS'08, Toulouse, France (oct 2008)
5. Huang, A.C., Garlan, D., Schmerl, B.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. In: ICAC '04: Proceedings of the First International Conference on Autonomic Computing, Washington, DC, USA, IEEE Computer Society (2004) 276–277
6. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in java. In Crnkovic, I., Stafford, J.A., Schmidt, H.W., Wallnau, K.C., eds.: CBSE. Volume 3054 of Lecture Notes in Computer Science., Springer (2004) 7–22
7. Fleurey, F., Delhen, V., Bencomo, N., Morin, B., Jezequel, J.M.: Modeling and validating dynamic adaptation. In: Proceedings of the 3rd International Workshop on Models@Runtime, at MoDELS'08, Toulouse, France (oct 2008)