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.

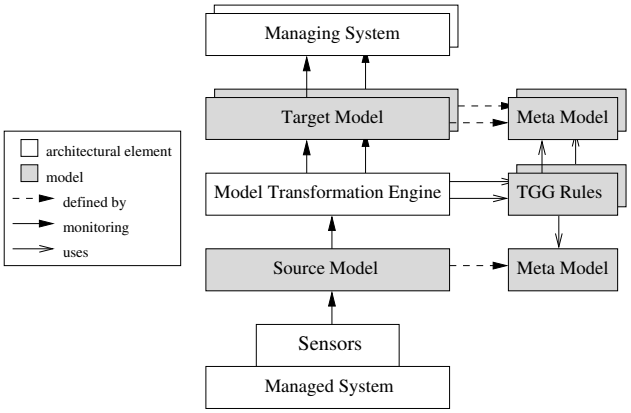

Fig. 1. Generic Architecture (cf. [5])

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/

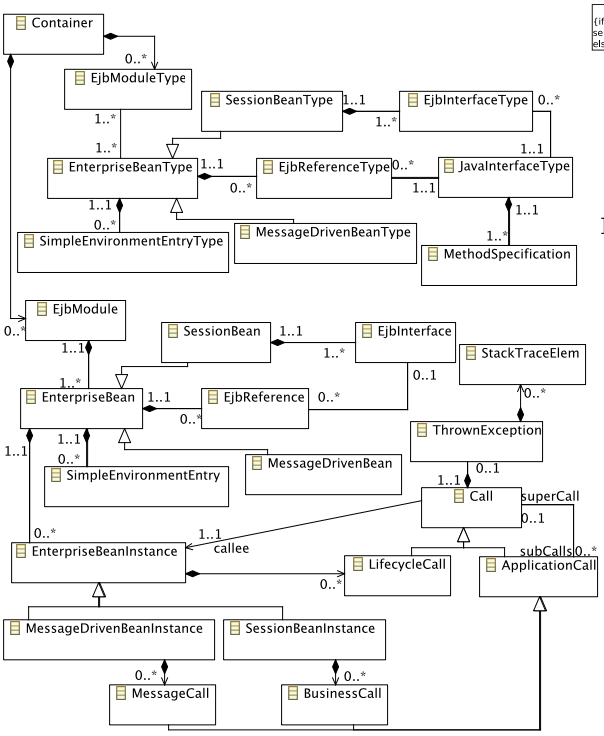

Fig. 2. Simplified Source Meta Model

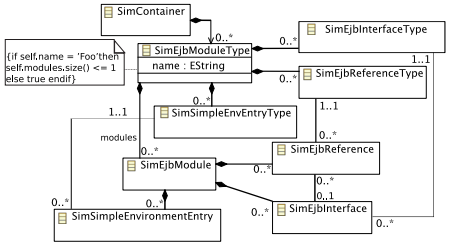

Fig. 3. Simplified Architectural Meta Model

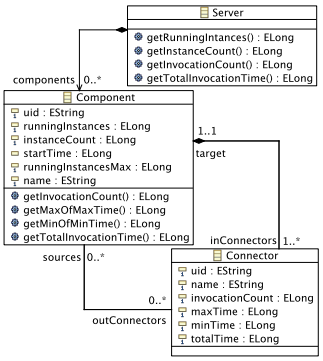

Fig. 4. Performance Meta Model

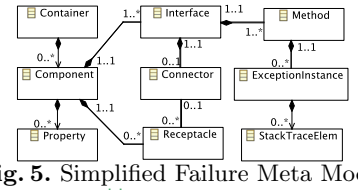

Fig. 5. Simplified Failure Meta Model

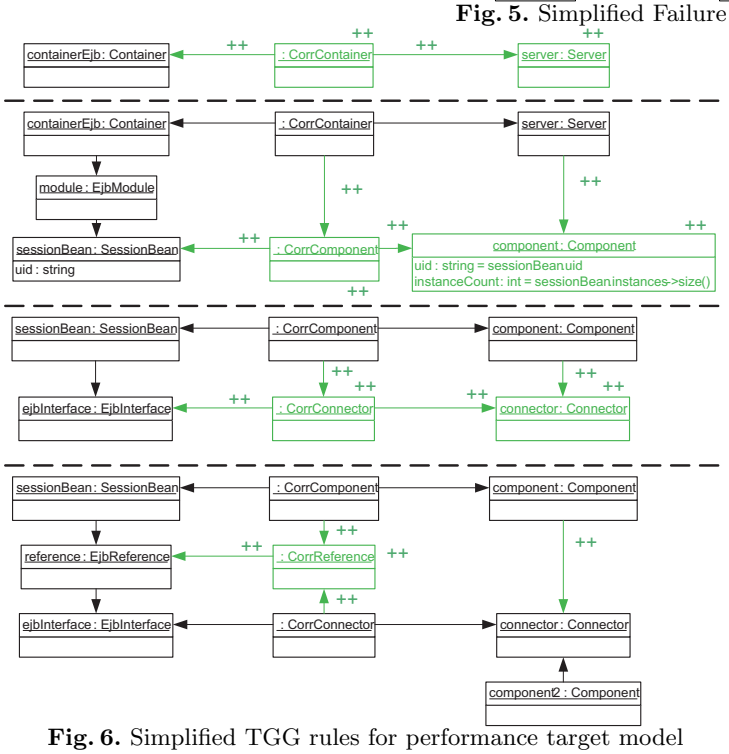

Fig. 6. Simplified TGG rules for performance target model

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)