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 metamodels, 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 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 in addition to some configuration parameters 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 run-time solution to be applicable online 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,

This is a post-peer-review, pre-copyedit version of an article published in Lecture Notes in Computer Science Vol. 6002. The final authenticated version is available online at: https://doi.org/10.1007/978-3-642-12261-3_13.

the efforts for developing monitoring approaches should be reduced. 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 metamodels which ease the specification of translations between different model types. As argued in [4], these techniques could be used for run-time models and thus also ease the development of architectural monitoring. Applying such techniques at run-time is promising for monitoring activities, as run-time models can provide appropriate abstractions of a running system from different problem space perspectives [5]. Thus, a run-time model can target a certain self-management capability and the corresponding concerns.

In this paper we propose a model-driven approach that enables a comprehensive monitoring of a running system by using metamodels and model transformation techniques as sketched in [6], 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. These models can be independent from the platform of a monitored system, which supports the reusability of analysis algorithms working on these models. The engine employs our optimized model transformation technique [7,8] 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 evaluated with respect to development costs and performance in Section 4. After discussing related work in Section 5, the paper concludes with a discussion on research challenges and future work.

2 Approach

To monitor the architecture and parameters of a running software system, our approach employs *Model-Driven Engineering* (MDE) techniques, which handle the monitoring and analysis of a system at the higher level of models rather than at the level of *Application Programming Interfaces* (APIs). Using MDE techniques, different models describing certain concerns or certain views on a running system required for different self-management capabilities are derived and maintained at run-time. Thus, models of a managed system and of its architecture essentially build the interface for monitoring the system. In the following, we present the generic architecture and the implementation of our approach.

2.1 Generic Architecture

The generic architecture of our monitoring approach is derived from [6] and depicted in Figure 1. A *Managed System* provides *Sensors* that are used to observe

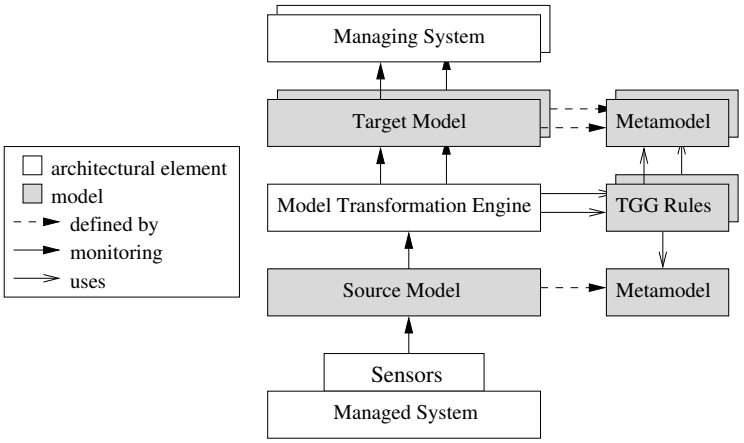

Fig. 1. Generic Architecture (cf. [6])

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 administration tools used by humans or even autonomic managers in case of a control loop architecture as proposed, among others, in [2].

Since it is difficult to obtain an architectural view on a managed system by using sensors at such a low level of abstraction, our approach provides a run-time model of the system in the form of a *Source Model*. This model enables a model-based access to sensors and it is maintained and updated at run-time if changes occur in the managed system. Though having a model-based view on a managed system, a source model represents all functionalities of the sensors and, therefore, it is usually related to the solution space of a managed system. Consequently, a source model might be quite complex and specific to the platform of a managed system, 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 *Metamodel*, 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 with respect to the source model and it provides a specific view on a managed system required for a certain self-management capability and the corresponding concern. Thus, in contrast to a source model that usually relates to the solution space of a managed system, target models tend to provide views related to problem spaces of different self-management capabilities and to abstract from the underlying platform of a managed system. This supports the reusability of managing systems that focus on problem spaces shared by different managed systems. For example, a target model might represent the security conditions or the resource utilization and performance state of a managed system to address *self-protection* or *self-optimization*, respectively. Moreover, a managing system being concerned with *self-optimization* will use only those target models that are relevant for optimizing a managed system, but does not have to consider

concerns or views that are covered by other capabilities such as *self-protection*. This also reduces the complexity for managing systems in coping with run-time models, though different target models may provide overlapping views. Consequently, several managing systems work concurrently on possibly different target models, as depicted in Figure 1.

The different target models are maintained by our *Model Transformation Engine*, which is based on *Triple Graph Grammars* (TGGs) [7,8]. *TGG Rules* specify declaratively at the level of metamodels how two models, a source and a target model of the corresponding metamodels, can be transformed and synchronized with each other. Thus, source and target models have to conform to user-defined metamodels (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 with respect to execution time as it avoids any recurring transformations from scratch. This enables the application of our technique at run-time. Therefore, for each target metamodel, 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 a *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 and it does not process unchanged model elements. The notifications contain all relevant information to identify and locate the changes in the source model 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.

Finally, our model-driven approach can be easily extended with additional target models. As target models can be platform-independent, the kinds of target models that are used in our approach are rather defined by managing systems than by our transformation engine or the underlying infrastructure. For example, to integrate an existing managing system or an analysis algorithm that uses a certain target model, only the corresponding target metamodel and TGG rules, which specify the synchronization of the source model with the corresponding target model, have to be provided. Thus, our approach fosters the reusability of managing systems or of analysis algorithms instead of having to re-engineer them to fit into our approach.

2.2 Implementation

The implementation is based on the autonomic computing infrastructure *mKernel* [9], which enables the management of software systems being realized with *Enterprise Java Beans 3.0* (EJB) [10] technology for the *GlassFish v2*[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 metamodel for the EJB domain that captures the capabilities of the API. This metamodel defines the source model in our example. A simplified version of it is depicted in Figure 2 and it is described in detail in the following section. Though our techniques are based on EMF, the whole infrastructure can run decoupled from the *Eclipse* workbench.

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 within a managed system, 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 metamodel for the EJB domain that specifies the source model is depicted in a simplified version in Figure 2.

The metamodel is conceptually divided into three levels. The top level considers the types of constituting elements of EJB-based systems, which are the

[1] https://glassfish.dev.java.net/ (Nov 4, 2009)
[2] http://www.eclipse.org/modeling/emf/ (Nov 4, 2009)

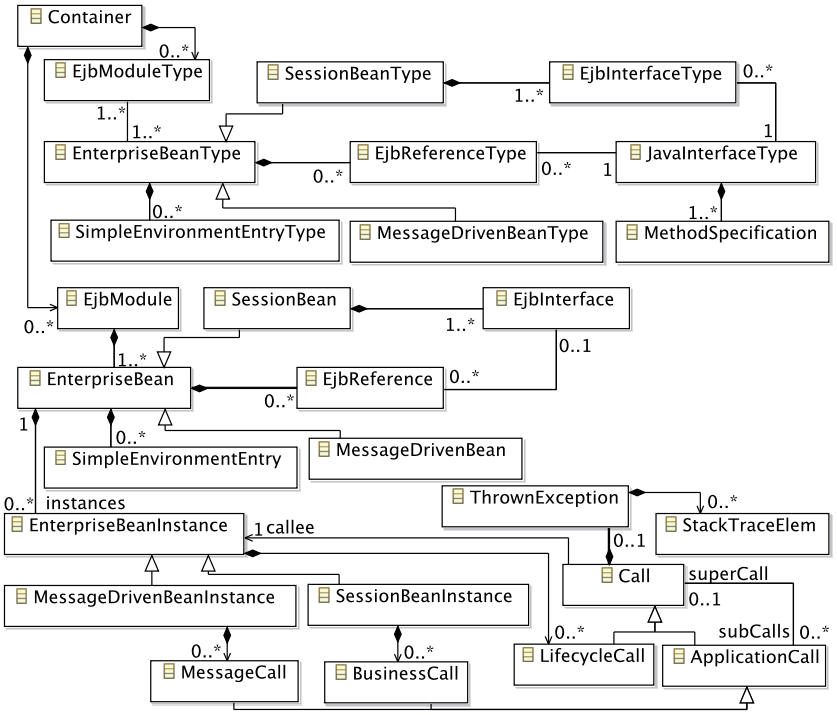

Fig. 2. Simplified Source Metamodel

results of system development. The middle level covers concrete configurations of EJB-based systems being deployed in a server. Finally, the lower level addresses concrete instances of enterprise beans and interactions by means of calls among them. The metamodel depicted in Figure 2 is simplified as it does not show, among others, attributes, enumerations, and several associations to navigate between the levels. For brevity, we refer to [9,10] to get details on the EJB component model and on the three levels.

Based on this metamodel, 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 concerns, we developed a metamodel 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 at higher level of abstractions aiming at run-time monitoring and analysis of architectural constraints, performance and failure states.

3.1 Architectural Constraints

Analyzing architectural constraints requires the monitoring of the architecture of a running system. Therefore, we developed a metamodel that is depicted in

Figure 3 and whose instances reflect simplified run-time architectures of EJB-based systems. It abstracts from the source metamodel 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. Moreover, it abstracts from concrete bean instances and calls among them.

To analyze architectural constraints, the *Object Constraint Language* (OCL) and checkers like EMF OCL[3] are used to define and check constraints that are attached to metamodel elements, as 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.

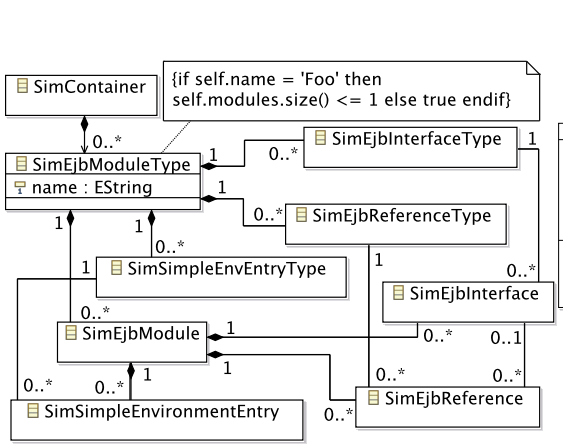

Fig. 3. Simplified Architectural Metamodel

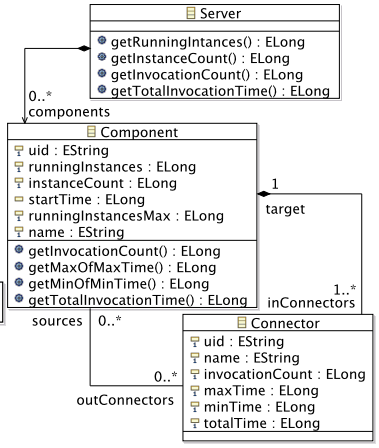

Fig. 4. Performance Metamodel

3.2 Performance Monitoring

Like the architectural target metamodel, the metamodel for target models being used to monitor the performance state of EJB-based systems also abstracts from the source metamodel. Moreover, it is independent from the EJB platform as it provides a view based on generic components and connectors. Figure 4 shows the corresponding metamodel.

The metamodel 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, the number of currently running instances, the maximum number of instances that have run concurrently, or the number of instances that have been created entirely are represented by the attributes *runningInstances*, *runningInstancesMax* 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

[3] http://www.eclipse.org/modeling/mdt/?project=ocl (Nov 4, 2009)

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 information about all connectors provided by the component (*inConnectors*), and a *Server* provides aggregated information about its hosted components. Overall, a model-based view is provided on the performance states of enterprise beans, which is comparable to parts of the *Performance Data Framework* defined by the *Java 2 Platform, Enterprise Edition Management Specification* [11].

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 to decide about relocating components, whose instances cause heavy loads, to other servers or improving the resource configuration.

The four TGG rules that are required to synchronize the source model with the performance target model are depicted in a simplified version in Figure 5. For all of them, nodes on the left refer to the source model, nodes on the right to the

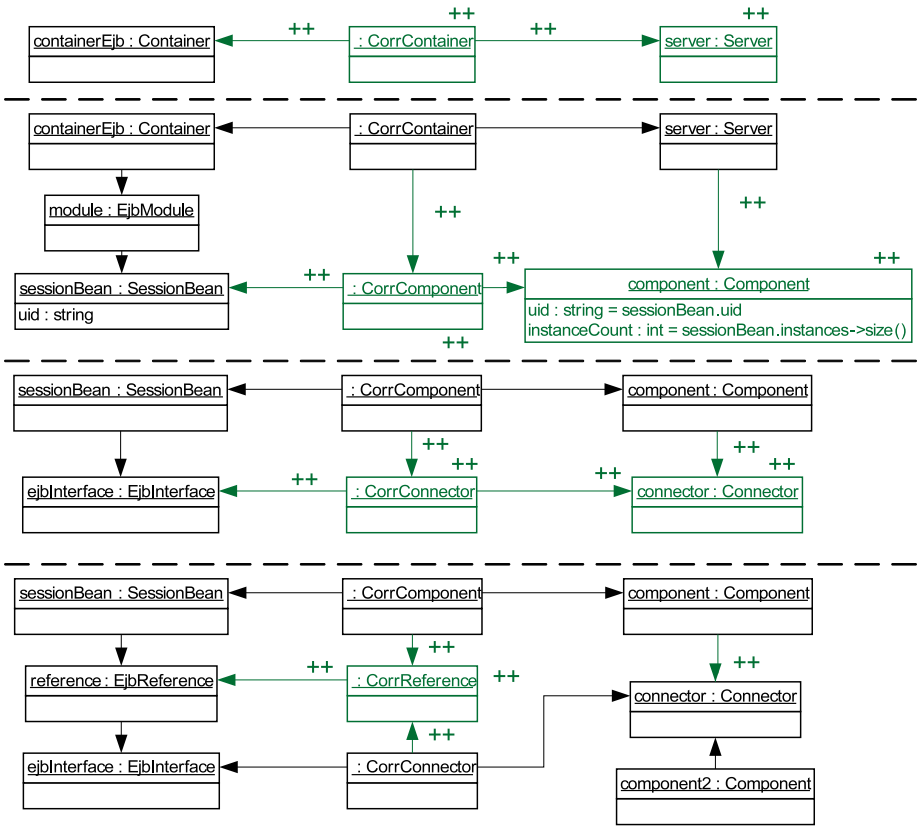

Fig. 5. Simplified TGG rules for performance target model

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 not drawn black and marked with *++* are created by applying the rule.

The first rule in Figure 5 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 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 similar to the second one. It maps an *EjbInterface* provided by a *SessionBean* to a *Connector* for the corresponding *Component*. The *CorrConnector* maintains the relation between the *EjbInterface* and the *Connector*.

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.

3.3 Failure Monitoring

The last target model is intended for monitoring failures within managed systems. The corresponding metamodel is shown in a simplified version in Figure 6.

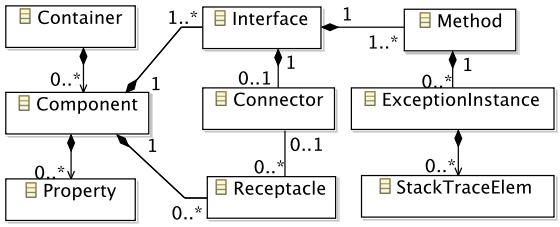

Fig. 6. Simplified Failure Metamodel

It defines a platform-independent architectural view on a managed system that is enriched with information about occurred failures. *Components* representing session beans are running in a *Container*. They can be configured through

Properties that correspond to simple environment entries, which are configuration parameters for enterprise beans. A component provides *Interfaces* and potential requirements of functionality from other components are represented by *Receptacles*. Receptacles can be connected to interfaces via *Connectors*. An interface specifies at least one *Method* whose usage at run-time might result in occurrences of exceptions (*ExceptionInstance*). Finally, each exception instance usually contains a stack trace (*StackTraceElem*).

Thus, the failure model provides a basic view on the architecture and configuration of an EJB-based system that can be related to exceptions that have occurred. If a certain method of an interface throws many exceptions, it can be analyzed whether the configuration of the component providing the corresponding interface is faulty because some property values of the component are not set appropriately, the implementation of the component is faulty, or other components using the corresponding interface are not using the interface appropriately.

4 Evaluation

In this section we evaluate our approach 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)**: For each target model, this approach retrieves the current run-time state of a managed system, i.e. a system snapshot, by extracting required 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*, for each target model, 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.

4.1 Development Costs

Having implemented our approach and the *NIA*, we are able to give concrete values indicating development costs, which are depicted in Table 1. Using our

Table 1. TGG Rules and Lines of Code (LOC)

	Model-Driven Approach			**NIA**
Target Model	Rules	Nodes/Rules	LOC	LOC
Simpl. Architectural Model	9	7,44	15259	357
Performance Model	4	6,25	5979	253
Failure Model	7	7,14	12133	292
Sum	20		33371	902

Table 2. Performance measurement [ms]

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

approach, we had to specify 20 TGG rules to define the transformation and synchronization between the source and all three target models 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, an 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 2.2), that however does not depend on any target metamodel and therefore is generic and reusable.

Consequently, specifying an acceptable number of TGG rules declaratively seems to be less expensive and less 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 as the 20 TGG rules regarding the effort for development.

Moreover, our approach using model transformation techniques is easier to extend or to adapt for the case that new target models have to be integrated or metamodels of already used target models have changed, respectively. It requires only the provision of the new or changed metamodels and the creation or update of appropriate TGG rules, which is done in a declarative manner. Afterwards, code corresponding to the rules is generated automatically.

In contrast and under the assumption that the other two approaches do not apply MDE techniques, especially code generation, these approaches require code-based re-engineering, i.e., usually altering imperative code. This can be expected to be more time consuming and error-prone than our approach, when having the different characteristics of the approaches regarding development costs in mind and when comparing declarative and imperative approaches.

4.2 Performance

Finally, the approaches are discussed with respect to run-time performance. The results of some execution time measurements[4] are shown in Table 2. 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,

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

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. Changes can range from the modification of a single attribute value to several model elements if an EJB module together with the included beans, interfaces, references and simple environment entries is added or removed from the system.

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 columns show the consumed time in milliseconds (ms) to retrieve a system snapshot from sensors and to create the three target models. For example, having deployed ten beans, it took 9663 ms to obtain and reflect the structural aspects in the target models, while the behavioral aspects required 43054 ms. Overall, the sum of both (52717 ms) has been consumed.

For our *Model-Driven Approach*, structural aspects are obtained through events and behavioral aspects through pull-oriented sensors. For the structural monitoring, the fourth to ninth columns show the composite average times of two subsequent activities:

1. *Event Processing*: n events, that are emitted by sensors to notify about n structural changes of the managed system, are processed by reflecting these changes in the source model. Regarding structural aspects, this activity keeps the source model up-to-date and it is performed by the *EMF Adapter* (cf. Section 2.2).
2. *Model Synchronization*: This activity synchronizes changes of the source model, which are the result of processing n events, to the three target models incrementally by invoking *once* the model transformation engine.

For example, for $n = 2$ and having deployed ten beans, 272 ms are consumed on average for processing two events, which includes updates of the source model depending on these two events, and for synchronizing at once 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 synchronization 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 last column of Table 2 shows the average times for retrieving *one* system snapshot of behavioral aspects, i.e., observed interactions among bean instances, through pull-oriented sensors, for reflecting this snapshot in the source model, and finally for synchronizing the updated source model to the three target models by invoking *once* the transformation engine. The third and last columns of Table 2 indicate that for both approaches the behavioral monitoring is expensive. This is a general problem, when complete system behavior should be observed.

However, comparing both approaches regarding behavioral monitoring, our approach outperforms the *NIA*. Using our approach, the sensors are only accessed

once to obtain the required interaction information, which are then reflected in the source model. In contrast, the *NIA* accesses sensors twice to get interaction information, namely to create the failure target model and the performance target model. Both target models require interaction information, to obtain thrown exceptions or execution times of calls among bean instances, respectively.

Regarding structural monitoring, our approach clearly outperforms the *NIA* as it works incrementally, while the *NIA* does not.

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 listener our approach requires for the source model.

To conclude, our approach outperforms the considered 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 [12], who use a system representation based on an architecture description language, but no advanced model-driven techniques like model synchronization.

Model-driven approaches considering run-time models, in contrast to ours, do not work incrementally to maintain those models or they provide only one view on a managed system. In [13] 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 [14] 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 [15] whose run-time model is updated incrementally, but reflects also only a structural view. However, they use model weaving techniques to transform models specified by the same metamodel to obtain new structures for a managed system [16].

All these approaches do not apply advanced MDE techniques like model transformation [13,14] or they do not consider the transformation of models specified by different metamodels [15,16]. In this context, initial preliminary ideas exist, like [17], who apply a QVT-based [18] approach to transform models at run-time. They use *MediniQVT*[5] 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- [19] or QVT-based ones when transforming class and block diagrams. Moreover, for the case of synchronization, our engine outperforms the other engines [20].

[5] http://www.ikv.de/ (Nov 4, 2009)

Though the approach presented here uses different models, metamodels and therefore different transformation rules, similar results can be expected for the case study used in this paper.

6 Conclusion, Discussion and Future Work

This paper presented our approach for the model-driven monitoring of software systems. It enables the efficient monitoring by using models based on metamodels and model synchronization techniques at run-time. The incremental synchronization between a run-time system and problem space oriented models for different concerns can be triggered when needed and multiple managing systems can operate concurrently. As target models can be platform-independent, our approach leverages the reusability of managing systems across different managed systems. The presented solution using TGGs outperforms feasible alternatives considering development costs and performance.

Since TGGs have the capability of bidirectional model synchronization, basically and as sketched in [6], this technique can also be used for adapting a managed system. Instead of performing architectural adaptations directly by changing the source model, we propose that changes are applied to target models and synchronized to the source model. However, several research challenges emerge in this context, which are discussed in the following.

Adapting the architecture of a managed system is complex, as a set of atomic changes might have to be performed. The order of changes performed on a target model might differ from the order of performing corresponding changes to the source model and therefore to the managed system. Not suitable orders might affect the consistency of a system. Moreover, in some cases the abstraction step between a source and a target model is too large. As a consequence, the relation between source and target models is only partial and need not to be injective. Therefore synchronizing target model changes to the source model requires additional information such as default values that depend on the concrete application. The same problem is discussed for round-trip engineering in [21] that emphasizes the difficulties of bidirectional model synchronization.

Even worse, when using multiple target models representing different concerns of a managed system, adaptations are more challenging, since changes can be applied concurrently to different target models. Conflicting changes can lead to an inconsistent source model and managed system. Consequently, coordination among managing systems is required, which can be done by restricting adaptations to one target model and controlling access to this model. However, we believe that adaptations can be specific for a certain concern and therefore for a certain target model. Thus, restricting adaptation to exactly one target model would not be appropriate, and a more generic solution addressing relationships, like dependencies, interactions, or trade-offs between different target models, respectively their concerns, would be required. Such relationships could be used, among others, to coordinate changes and to validate changes of one target model by analyzing the impact of this change to other target models.

As future work, we are currently investigating the usage of model synchronization techniques for architectural adaptations using one target model and then, to enable adaptations using multiple target models. Moreover, extending our approach to a distributed setting is considered.

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), 41–50 (2003)
3. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In:Proc. of the Workshop on Future of Software Engineering, pp. 259–268. IEEE, Los Alamitos (2007)
4. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: Proc. of the Workshop on Future of Software Engineering, pp. 37–54. IEEE, Los Alamitos (2007)
5. Blair, G., Bencomo, N., France, R.B.: Models@run.time. Computer 42(10), 22–27 (2009)
6. 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, pp. 67–68. ACM, New York (2009)
7. Giese, H., Wagner, R.: From Model Transformation to Incremental Bidirectional Model Synchronization. Software and Systems Modeling 8(1) (2009)
8. Giese, H., Hildebrandt, S.: Incremental Model Synchronization for Multiple Updates. In: Proc. of the 3rd Intl. Workshop on Graph and Model Transformation. ACM, New York (2008)
9. 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, pp. 755–762. IEEE, Los Alamitos (2008)
10. DeMichiel, L., Keith, M.: JSR 220: Enterprise JavaBeans, Version 3.0: EJB Core Contracts and Requirements (2006)
11. Hrasna, H.: JSR 77: Java 2 Platform, Enterprise Edition Management Specification (2006)
12. 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)
13. 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)
14. Dubus, J., Merle, P.: Applying OMG D&C Specification and ECA Rules for Autonomous Distributed Component-based Systems. In: Proc. of the 1st Intl. Workshop on Models@run.time (2006)
15. 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, pp. 127–136 (2008)

16. Morin, B., Barais, O., Jézéquel, J.M., Fleurey, F., Solberg, A.: Models@Run.time to Support Dynamic Adaptation. Computer 42(10), 44–51 (2009)
17. 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)
18. OMG: MOF QVT Final Adopted Specification, OMG Document ptc/05-11-01
19. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A Model Transformation Tool. Science of Computer Programming 72(1-2), 31–39 (2008)
20. Giese, H., Hildebrandt, S.: Efficient Model Synchronization of Large-Scale Models. Technical report, No. 28, Hasso Plattner Institute, University of Potsdam (2009)
21. Hettel, T., Lawley, M.J., Raymond, K.: Model Synchronisation: Definitions for Round-Trip Engineering. In Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 31–45. Springer, Heidelberg (2008)