Model-Driven Self-Management of Legacy Applications

Markus Debusmann, Markus Schmid, Reinhold Kroeger

Fachhochschule Wiesbaden - University of Applied Sciences
Department of Computer Science
Distributed Systems Lab
Kurt-Schumacher-Ring 18, 65197 Wiesbaden, Germany
{debusmann|schmid|kroeger}@informatik.fh-wiesbaden.de

Abstract. Increasing complexity of todays applications and services leads to the emerging trend of self-managing systems. Legacy applications often offer interfaces for manual management and very rarely do provide self-management features. Therefore it is very important to reuse existing management interfaces for achieving self-manageability.
This paper presents our model-driven self-management approach of legacy applications. We introduce a framework for model-driven service level management which transforms abstract SLAs defined in UML into concrete SLA descriptions and deploys them for management. These SLAs are used to define the goals for our self-management agent which is responsible for providing feedback control. Its management knowledge is transformed from UML models using also a model-driven approach.

1 Introduction

The ongoing economic pressure on enterprises increases the need for outsourcing and purchasing services from external service providers. Service Level Agreements (SLAs) [1, 2] represent agreed upon contracts between service providers and service customers. SLAs are an important element within the outsourcing business and define Quality of Service (QoS) parameters of the provided service. Typically, these SLA parameters are related to performance and availability, e.g., the mean value for the response time of service functions. Service Level Objectives (SLOs) define reference values (e.g. thresholds and value sets) for SLA parameters. In order to detect violations, the SLA parameters have to be monitored and compared to the SLOs at runtime.

Over the past years, the complexity of distributed applications has significantly increased, accompanied by a rapid change of emerging middleware technologies, such as CORBA [3], J2EE [4], and Web Services [5]. These frequent technology changes as well as the overall complexity significantly complicate the management of today's distributed applications. Therefore self-managing functionality of applications is becoming more and more crucial. As a consequence, IBM started its *Autonomic Computing Initiative* [6] in 2001 whose goal is to develop self-managing systems covering one or more of the following aspects: self-configuration, self-optimisation, self-healing, and self-protection [7]. IBM's competitor HP also introduced a self-managing approach called *Adaptive Enterprise* [8].

The development of self-managing systems is a very complex task since such a system has to incorporate the knowledge that is usually hidden in the brains of human administrators. This leads to a number of critical research questions, like:

– How can management knowledge be formally represented in an effective way?
– How does a self-managing system handle situations that were not foreseen during the systems design?
– How can the design of self-managing systems be supported?

In the area of software development, the Model Driven Architecture (MDA) represents an approach for effectively designing complex systems as abstract models, which, ideally, are then automatically transformed into application code.

We claim that the principles of the MDA can tremendously benefit the development of self-managing systems. In this paper, we concentrate on achieving self-manageability for traditionally managed legacy systems. This is very important, since enterprises made large investments in existing software systems which are not easily replaceable. Usually legacy applications offer a set of interfaces for their manual management which should also be used for self-management. In addition, a lot of experience on managing existing applications has been gained during their operation. Our approach has the following advantages:

– existing management interfaces are reused, i.e., the application being managed does not need to be modified,
– existing management know-how is leveraged by formalisation.

Within our approach, the use of MDA for obtaining self-management for legacy applications is twofold:

1. We propose the definition of abstract SLA patterns, that can then be transformed and bound to concrete management platforms. These SLA patterns provide the basis for setting up a management platform for monitoring SLA compliance, and for configuring a self-management agent for carrying out corrective actions on the system under management based on the compliance information.
2. The knowledge base of a self-management agent can be set up by using a model driven approach, i.e., abstract management knowledge is transformed into concrete management directives by using appropriate model transformations.

The MDA-based management approach provides the necessary level of abstraction needed for dynamic on-demand services in large-scale heterogeneous environments. Furthermore, the proposed abstract SLA templates provide added value by being reusable and long-lasting.

The remainder of this paper is structured as follows: section 2 presents the foundations of the Model Driven Architecture and the extension mechanisms of the Unified Modeling Language, while section 3 presents our architecture for model-driven self-management in detail. After this, section 4 presents the application of the implemented prototype in a sample e-business scenario. The paper concludes with a summary and an outlook of our future work in section 5.

2 Model Driven Architecture & UML

The design and implementation of modern software systems is complex and costly due to rapid and continuous technology changes. These require tremendous efforts in application integration, as well as support for developing adaptive and open applications that can keep track with the steady technological progress. The Model Driven Architecture (MDA) [9], defined by the Object Management Group (OMG), aims at solving integration and development problems. It defines a model-based methodology which separates the application problem domain from the technological specifics of the underlying computing infrastructure [10].

The MDA distinguishes between three models: the *Platform-Independent Model (PIM)*, the *Platform-Specific Model (PSM)*, and the *Platform-Specific Code (PSC)*. The models can be converted into each other by a process called *model transformation*. Typically, an MDA tool (or a chain of tools) supports the definition and transformation of these models.

The PIM is the core ingredient of the MDA. It specifies a system independent of any platform-specific technology that is later used to implement the system. The PIM focuses on the business functionality and behaviour including the relationships between the components involved. MDA's vision is the specification of computationally complete PIMs. Ideally, all development is carried out at the modelling level, and these models can be exploited for testing and verification purposes.

The PSM is obtained by transforming the PIM for a given target platform, e.g., CORBA, J2EE, or Web Services. The goal is to capture as much of the platform know-how as possible in the PIM-PSM transformation and to generate as much of the PSM as possible automatically. Thus, the developer can concentrate on the application logic in the PIM and is relieved from the complexities of platform-specific code.

The last step is the generation of the (platform-specific) code from the PSM. Ideally, a MDA tool generates all necessary application code (and associated files) for the chosen programming language and deployment platform.

For the specification of PIM and PSM, the MDA proposes the Unified Modeling Language (UML) [11] which is also an OMG standard. UML supports various meta-level mechanisms to extend its specification elements: stereotypes, tagged values, and constraints. A profile represents a set of extension elements and is used for adapting UML to the needs of certain problem domains. For the exchange of UML models, e.g. between UML tools, the OMG has defined an XML-based interchange format called XML Metadata Interchange (XMI) [12].

3 Architecture for Model-Driven Self-Management

This section introduces our architecture for model-driven self-management. Section 3.1 gives an introductory overview while section 3.2 discusses the model-driven service level management in detail. After this, section 3.3 describes the structure of our self-management agent and the set up of its knowledge base, using a model-driven approach.

3.1 Overview

Figure 1 depicts a high-level view upon our architecture for model-driven self-management. The architecture comprises two subsystems:

- the *SLM system* for deploying and monitoring SLAs, and
- the *self-management agent* for accomplishing corrective actions.

A prerequisite for an effective self-management is the formal and unambiguous specification of management objectives. This is the task of a service level agreement which specifies all management-relevant SLA parameters of the system under management and the SLOs, the self-management system is trying to fulfil. The service level management (SLM) system is responsible for observing the conformance level of an SLA. Therefore, SLAs are being deployed by the SLA deployment system and then monitored by the SLA monitoring system. The system retrieves raw data and metrics from the system under management and computes the management-relevant parameters which are then compared to the defined SLOs.

In order to achieve self-manageability the self-management agent has to decide whether an adaptation of the system under management is required or not. Its decisions are based on metrics directly retrieved from the system under management and on metrics and events retrieved from the SLM system. These events may signal that a certain SLO is about to be violated and thus a proactive corrective action may be required. For issuing feedback control the self-management agent uses existing management interfaces of the system under management.

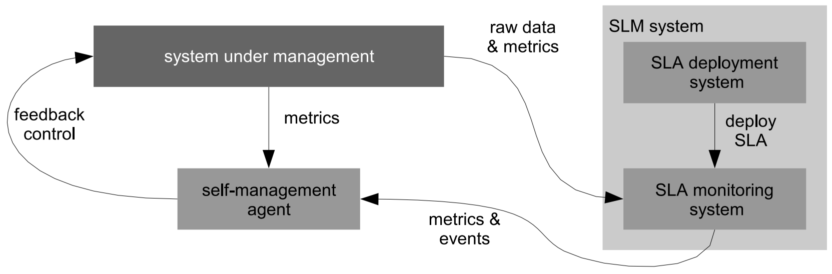

Fig. 1. Overview of the general architecture

The following two sections discuss the SLM system and the self-management agent in more detail.

3.2 Model-Driven Service Level Management

This section describes the concept of the model-driven management of service level agreements realised within the SLM system. The deployment of an SLA is a prerequisite for this.

The central idea of our approach is to apply the principles of the Model Driven Architecture to the problem domain of Service Level Management. Therefore, we need to define the meaning of the different MDA models (PIM, PSM, and PSC) for the area of SLM first. In our approach, the PIM represents an *SLA pattern* which defines an abstract SLA which can be bound to various appropriate configurations and environments. Thus, an SLA pattern only contains the types and their relations to various other component types and an abstract description how SLA parameters are derived.

In the next step, an SLA pattern ("the PIM") is transformed into a concrete SLA ("the PSM"), which we call *SLA instance*, by binding the pattern to concrete configuration information. This includes the binding of the abstract component types to concrete component instances as well as the identification of a concrete management framework, that is used for managing the SLA (e.g. WBEM/CIM). An SLA instance contains all information necessary for configuring the management infrastructure for the given SLA.

The third step is the actual deployment of the SLA instance ("the PSC"). The result of this step is the actual configuration of the management infrastructure enabling it to autonomously manage the conformance of the SLA.

Figure 2 depicts the overall architecture for the model-driven deployment of SLAs.

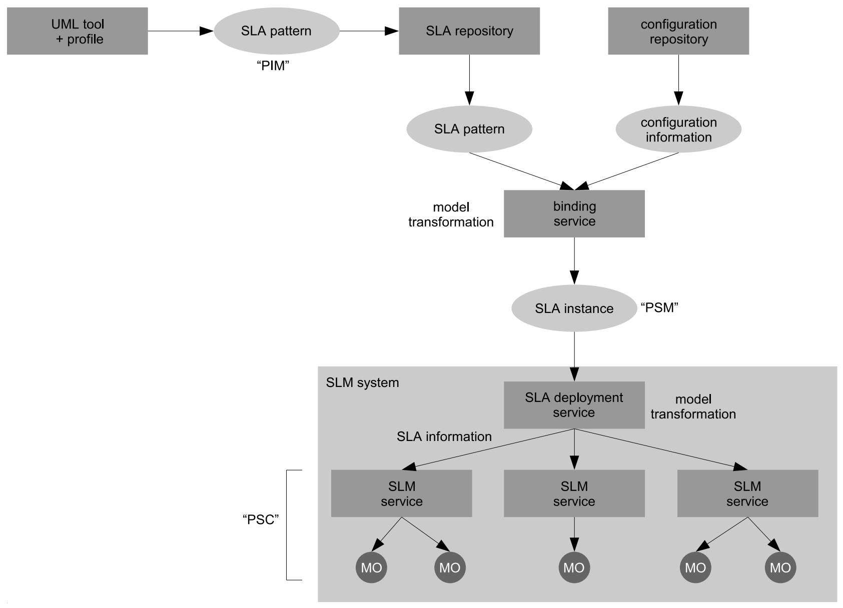

Fig. 2. Architecture for model-driven deployment of SLAs

Defining an SLA pattern The starting point for Service Level Management is the signing of a legal contract (SLA) between a service customer and a service provider. In our approach, the formalisation of the contract is done in UML by an administrator using a UML tool and an appropriate UML profile tailored for SLA definition (for details see [13]). The result of the formalisation is the SLA pattern which is stored in the SLA repository for further use. An SLA pattern specifies the relations and dependencies of service types in an abstract manner. For example, a service type for providing HTML content might depend on a web server, a web container, a database, some machines hosting the components, and network connections between them. The virtue of the approach is that for a certain type of service a pattern has to be defined only once and can then be re-used and instantiated multiple times.

Generating an SLA instance For actually managing an SLA, a corresponding SLA pattern is retrieved from the SLA repository and bound to a concrete configuration. This is a rather complex task, because all real instances of the types identified in the SLA pattern have to be known. The necessary information is usually retrieved from a configuration repository which contains all relevant information concerning the dependencies between services and components. The transformation of the SLA pattern into an SLA instance is performed by the *binding service* (see figure 2). As explained above, such an SLA instance is constructed for a certain SLM system.

Deploying an SLA instance The final MDA step is the generation of platform-specific code. For Service Level Management this corresponds to the physical deployment of the SLA instance generated in the previous phase.

The SLA instance is passed to an *SLA deployment service* (see figure 2) which is specific for the target SLM system. The SLA deployment service configures the services of the SLM system that are responsible for managing the SLA conformance. For example, when using WSLA [14] as SLM system, the WSLA deployment service will configure WSLA's measurement service and condition evaluation service. After configuration, these services are set up to autonomously manage the deployed SLA.

Prototype implementation The SLA UML profile has been defined using the UML modelling tool Kase [15]. An example of an SLA pattern modelled using UML can be found in [13]. The models are exported as XMI [12] documents and further processed using the Xalan-J engine (http://xml.apache.org/) and an XSLT [16] style-sheet. The result of this simplification process is an SLA pattern represented in XML and containing only the information that is necessary for representing the SLA.

For instantiating SLA patterns we developed a Java GUI. We used the *Java Architecture for XML Binding* (JAXB, http://java.sun.com/xml/jaxb/) framework for efficiently implementing the access to the XML-represented SLA pattern. The GUI uses a set of plugins for different target platforms to generate the SLA instances. After choosing an SLA template and the desired target platform, the plugin prompts the administrator for the information which is necessary to bind the pattern. The generated instance is then handed over to the platform-specific deployment service for installation. Currently, the SLA patterns are simply read from the file system. In the future, the SLA repository will be implemented as a Web Service.

3.3 Model-Driven Generation of Management Knowledge

We will now describe the model-driven generation of management knowledge which is subsequently used by the self-management agent.

Figure 3 depicts the architecture for the model-driven configuration of the self-management agent which is tightly interrelated with the architecture for the model-driven deployment of SLAs described in the previous section.

Structure of the self-management agent Internally, the self-management agent consists of four major building blocks. Details on the architecture of the self-management agent are provided in [17]. Adaptors and converters are responsible for retrieving metrics and receiving events. They are also responsible for converting this data into the internal messaging format. The task of the internal messaging subsystem is the delivery of all internal information and events. The knowledge interpreter in combination with the management knowledge is the ’brain’ of the self-management agent. The management knowledge is evaluated on regular intervals and on the occurrence of certain events. Necessary corrective actions are carried out by application-specific *actors* which realise the feedback control. In a simple scenario actors could be implemented by operating system commands for killing and restarting processes. Actors could also provide a generic interface to interact with a certain class of legacy applications, e.g. an SNMP interface to communicate with SNMP-enabled applications.

Currently, we assume that the person modelling the management knowledge knows or anticipates the correlation between the system metrics and its control variables. Since we focus on achieving self-manageability for legacy applications the necessary knowledge should exist. For new applications partial knowledge can be already gained in the development process. This knowledge is then being gradually completed.

Prototype implementation The self-management agent was implemented on AIX and Linux, using C++. The two main design goals for the agent are modularity and high availability. All functionality of the self-management agent (the adaptors & converters, the knowledge interpreter, and the actors) are implemented as shared libraries which can be dynamically loaded into the agent at runtime. The core of the agent is the *module manager* which provides a framework for concurrent communication between the modules.

For achieving high availability the self-management agent is realised as a self-checking process pair. The agent continuously self-diagnoses its modules in order to ensure proper operation. In order to be able to manage a wide range of applications the self-management agent supports a large number of adaptors and converters as well as actors. At the moment, we support SNMP including traps, log analysis, shell scripting, CORBA, WBEM/CIM, and Web Services.

Currently, the self-management agent only supports a simple policy language (defined as an XML schema) for describing the management knowledge, but the architecture is open to support other paradigms, such as finite state machines or neural networks.

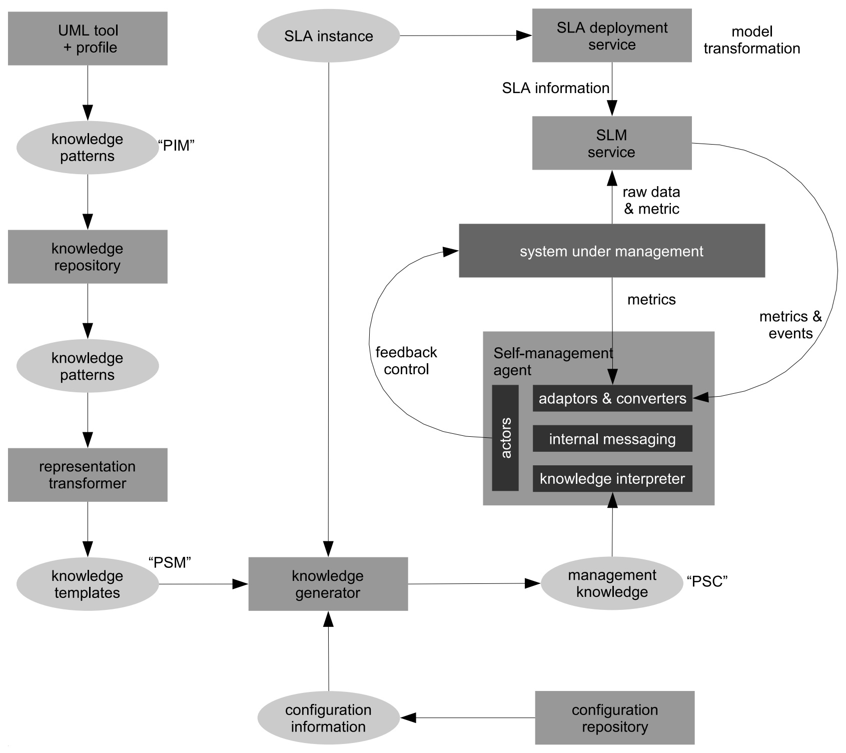

Fig. 3. Architecture for model-driven configuration of the Self-Management Agent

Modeling management knowledge Similar to the abstract nature of the SLA patterns, the management knowledge of the self-management agent is described in an abstract manner, called *knowledge patterns*. As a starting point, we use UML state-charts and UML activity diagrams in conjunction with an UML modelling tool and an appropriate profile for describing the knowledge patterns. After modelling, these patterns are stored in a knowledge repository for further use.

For instantiating management knowledge appropriate knowledge patterns have to be retrieved from the knowledge repository. In a first step, the patterns (the "PIM") are transformed by the *representation transformer* into a desired knowledge representation. From our point of view there are certain limitations related to possible target formats of knowledge. We expect to be able to successfully transform our knowledge patterns into policy languages, logical expressions or certain finite state machine models. For different representation formats, such as AI rules, an alternative modelling approach is required.

The result of the transformation process are *knowledge templates* (the "PSM") which still represent abstract knowledge but already for a target representation format. Finally, the *knowledge generator* generates concrete *management knowledge* (the "PSC"). The generator instantiates the knowledge templates using the information contained in the SLA instance generated by the SLM system (see section 3.2). Possibly, additional configuration information retrieved from a repository has to be included as well. The generated management knowledge is necessarily application-specific, but may in large parts be similar for certain application types, e.g. Web Servers. It is interpreted by the self-management agent for deciding whether feedback into the system under management is required and which (application-specific) actors are to be invoked in this case.

3.4 Related Approaches

[18] describes an approach which successfully utilises MDA for modelling QoS aspects when designing distributed applications.

IBM carries out significant research work on self-management, but the design of the controllers presented is highly application-specific: [19] describes the application of control theory to a DB/2 DBMS, [20] shows ways to optimise queueing in a Lotus Notes server. In principle, such complex controller-approaches could complement our architecture, however a possible interaction with SLAs and generated management knowledge requires further analysis.

4 Sample Application Scenario

The prototype has been evaluated in our e-business application infrastructure which represents the system under management (see figure 4). This infrastructure consists of the Squid Web proxy on the client side and the Apache Web server, the Tomcat Web container, the JBoss application server, the ORBacus CORBA platformand the relational database MySQLon the server side. Our environment is fully instrumented using the OpenGroup *Application Response Measurement (ARM)* API [21] for determining the response and processing times of individual components within our e-business environment.

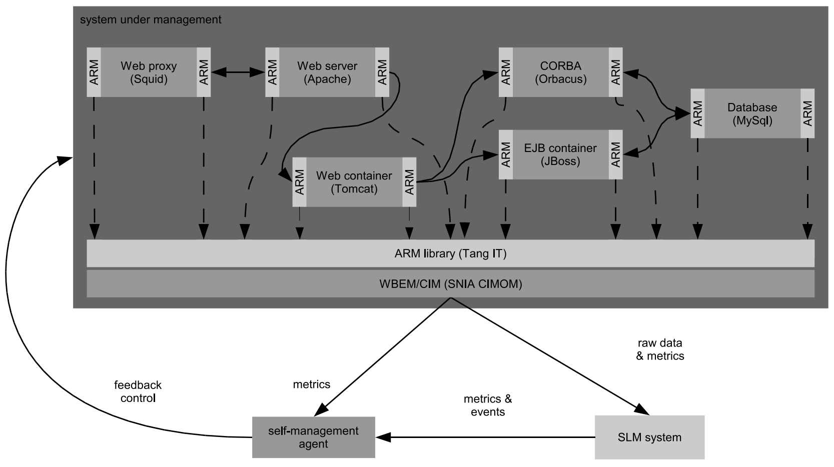

Fig. 4. Managing an e-business environment

The collected performance measurement data is used by the SLM system for validating SLOs (see [22] for details). In our sample scenario we defined SLOs concerning response time and availability. The actual response times are monitored by the SLM system. In case a response time SLO is violated an appropriate event is sent to the self-management agent in order to trigger a corrective action. The self-management agent tries to fulfil the response time goals by starting additional JBoss instances within a cluster. The EJB client within Tomcat uses the JBoss SmartProxy mechanism for achieving load balancing. If the response times decreases below a certain limit, the self-management agent again gradually removes JBoss instances from the cluster in order to save resources.

The availability of the components is observed by the self-management agent itself via its adaptors and converters. In case a component crashes it is restarted automatically by the self-management agent.

Figure 5 depicts the knowledge pattern for managing the number of JBoss instances. The pattern is modelled as a state machine consisting of three states. The automaton initially starts in the `Operating` state and remains there as long as the response time requirements are satisfied. If the response time rises above a certain limit (`enter_critical`) and the maximum number of parallel JBoss instances has not been reached, a new instance is started and the state machine enters the `Heavy_load` state. As long as the response time remains above `enter_critical` the automaton continues to start new instances until the maximum number is reached. If the response time increases further and the maximum number of instances is already running, the state machine sends the `event_overload` notification and enters the `Overload` state. The notification indicates that additional intervention is required by a superior self-management agent or by a human administrator. If the response time drops again below `leave_critical`, the state machine will send an `event_no_overload` notification and will switch back to the `Heavy_load` state. If the response time drops further below `leave_critical`, the automaton will remove instances in order to save resources and will reenter the `Operating` state. If the response time falls below `low_load`, the state machine will continue removing instances.

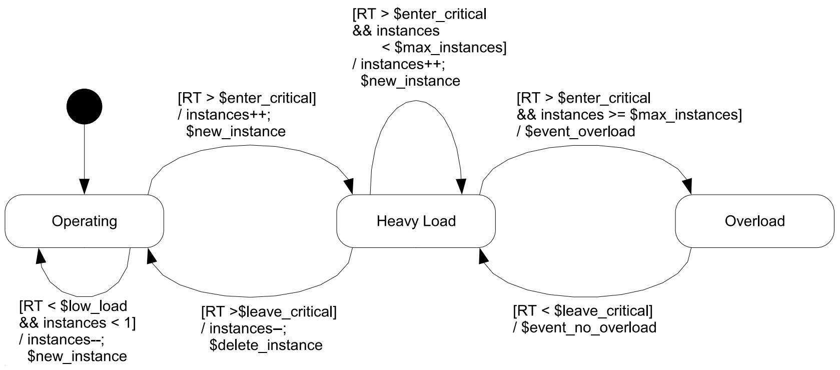

Fig. 5. Knowledge pattern for managing JBoss instances

Elements starting with $ represent placeholders that are replaced by the knowledge generator during the transformation process when creating the management knowledge. As can be seen, the presented knowledge pattern is not limited to managing JBoss instances but can be reused for response time management of any component type where multiple instances reduce the resulting response time.

5 Conclusions and Future Work

The constantly increasing complexity of distributed applications and services requires a higher degree of self-management in order to efficiently fulfil rising quality requirements. Traditionally, most applications solely provide interfaces for manual management. In this paper, we propose an approach for achieving self-manageability of legacy applications by leveraging their existing management interfaces. Our approach uses principles of the Model Driven Architecture for obtaining self-management. The overall architecture consists of two basic subsystems: the SLM system for deploying and monitoring SLAs and the self-management agent for realising feedback control.

For effectively deploying SLAs in order to monitor them with an SLM system we propose a model-driven approach. SLAs are modelled in an abstract manner using UML and are called SLA patterns. They can be reused by binding them to various appropriate configurations. A bound SLA pattern is referred to as SLA instance which contains all necessary information for monitoring the SLA. After the deployment of the SLA instance the SLA is automatically monitored. The SLA determines the reference values that should be satisfied by the self-management.

The self-management agent is responsible for achieving conformance with the deployed SLAs. The modular agent consists of adaptors & converters for retrieving metrics and events, an internal messaging system, a knowledge interpreter for the self-management algorithms, and actors for realising feedback control.

The management knowledge is transformed using the model-driven approach as well. First of all, abstract knowledge patterns are modelled as UML state-charts and activity diagrams using an UML tool plus an appropriate UML extension. The knowledge patterns can then be transformed into knowledge templates that represent a desired target representation of the knowledge patterns. Finally, the knowledge templates are instantiated using SLA instances and additional configuration information. The generated management knowledge represents the self-management algorithms that are executed by the knowledge interpreter.

We successfully implemented the model-driven approach for SLA deployment and monitoring and the self-management agent. The implementation of model-driven generation of the management knowledge is ongoing work. Our future work will concentrate on completing the implementation and gaining experience in different modelling formats for knowledge patterns and their transformation into various target formats in order to realise different types of feedback algorithms.

References

1. Lewis, L.: Managing Business and Service Networks. Kluwer Academic Publishers (2001)

2. Sturm, R., Morris, W., Jander, M.: Foundations of Service Level Management. SAMS Publishing (2000)
3. Object Management Group: Common Object Request Broker Architecture: Core Specification. (2004) Version 3.0.3 - Editorial changes, OMG document formal/04-03-01.
4. Sun Microsystems, Inc.: Java 2 Platform Enterprise Edition Specification, v1.4. (2003) Final Release, http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf.
5. World Wide Web Consortium: Web Services Architecture. (2004) W3C Working Group Note, http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.
6. Horn, P.: Autonomic Computing — IBM's Perspective on the State of Information Technology. IBM Corporation. (2001) http://www.research.ibm.com/autonomic/.
7. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer Magazine (2003) 41–50
8. Hewlett-Packard: Building an adaptive enterprise — Linking business and IT. (2003) http://h30046.www3.hp.com/solutions/utilitydata.html.
9. Object Management Group: Model Driven Architecture (MDA). (2001) OMG document ormsc/2001-07-01.
10. Object Management Group: Model Driven Architecture (MDA). (2001) document number: ormsc/2001-07-01.
11. Object Management Group: OMG Unified Modeling Language Specification. (2003) Version 1.5, OMG document formal/03-03-01.
12. Object Management Group: XML Metadata Interchange (XMI) Specification. (2003) Version 2.0, OMG document formal/03-05-02.
13. Debusmann, M., Geihs, K., Kroeger, R.: Unifying Service Level Management using an MDA-based Approach. In Boutaba, R., Kim, S.B., eds.: Proceedings of the 9^{th} International IFIP/IEEE Network Operations and Management Symposium (NOMS 2004), IEEE (2004) 801–814 Seoul, South Korea.
14. Keller, A., Ludwig, H.: The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Journal of Network and Systems Management, Special Issue on "E-Business Management" **11** (2003)
15. Weis, T.: Kase UML Tool. (2003) QCCS Project, http://www.qccs.org/KaseTool.shtml.
16. World Wide Web Consortium: XSL Transformations (XSLT). (1999) Version 1.0.
17. Debusmann, M., Kroeger, R.: Widening Traditional Management Platforms for Managing CORBA Applications. In Zieliński, K., Geihs, K., Laurentowski, A., eds.: Proceedings of the 3^{rd} IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS'2001), IFIP, Kluwer Academic Publishers (2001) 245–256 Krakow, Poland.
18. Weis, T., Ulbrich, A., Geihs, K.: Modellierung und Zusicherung nicht-funktionaler Eigenschaften bei Entwurf, Implementierung und zur Laufzeit verteilter Anwendungen. In Irmscher, K., Faehnrich, K.P., eds.: Kommunikation in Verteilten Systemen (KiVS), GI/ITG, Springer (2003) 119–130 (in German).
19. Diao, Y., Eskesen, F., Froehlich, S., Hellerstein, J.L., Spainhower, L.F., Surendra, M.: Generic Online Optimization of Multiple configuration Parameters With Application to a Database Server. In: Proceedings of the fourteenth IFIP/IEEE Workshop on Distributed systems: Operations and Management (DSOM 2003). (2003)
20. Parekh, S., Gandhi, N., Hellerstein, J., Tilbury, D., Jayram, T., Bigus, J.: Using Control Theory to Achieve Service Level Objectives In Performance Management. In: Proceedings of the Seventh International Symposium on Integrated Network Management (IM). (2001)
21. The Open Group: Systems Management: Application Response Measurement (ARM). (1998) Open Group Technical Standard, Document Number: C807.
22. Debusmann, M., Schmid, M., Schmidt, M., Kroeger, R.: Unified Service Level Monitoring using CIM. (2003) EDOC 2003, Brisbane, Australia.