Model-driven Development of Self-managing Software Systems

Matthias Rohr, Marko Boskovic, Simon Giesecke, Wilhelm Hasselbring
{rohr|marko.boskovic|giesecke|hasselbring}@informatik.uni-oldenburg.de
Graduate School TrustSoft
Software Engineering Group
University of Oldenburg
26111 Oldenburg, Germany

Abstract— **The promise of self-management is to increase the dependability of complex software systems and its quality-of-service. However, self-management is a very complex task if implemented manually at code level. It introduces high risks to the system's maintainability and dependability. Model-driven development of self-management at the architectural level is a promising alternative to manual low-level approaches.**

This paper outlines a model-driven approach for the model-driven realisation of self-management. The core of the approach are meta-models to specify constraints (based on architectural views), monitoring, and reconfiguration operations. These models can be used to generate self-management consisting of (1.) the monitoring instrumentation, (2.) the runtime model that reflects the current state of the system in causal connection to architectural entities, (3.) the automatic checking of the conformance of the current runtime model to the given constraints, and (4.) the mapping to the reconfiguration operations that are provided by the employed middleware platforms.

I. Introduction

Self-management is a concept to cope with the increasing complexity and dependability requirements of software systems. For example, a self-managing system could improve the availability by automatically diagnosing failures and performing the reconfiguration that is required for the repair of the fault. Although self-management and other self-* terms (e.g., self-healing, self-configuration, self-protection) are buzzwords of the last years, the concept is not new [1]. For instance, already the SIFT system [2] had a fault tolerance capability that can be called "self-healing", as it can automatically perform failure detection and reconfiguration.

The pure *technical* problems of self-management have been largely solved, as standard middleware already provides such self-management functionality (e.g., late binding, architectural reconfiguration, or service monitoring). However, the major challenge in developing self-management of complex systems is to develop an efficient design method for self-managing systems, i.e. to put the technical self-management solution fragments together in a systematic way.

Former approaches to designing self-managing systems (such as SIFT mentioned above) were specific to single applications. Such specific solutions are usually very expensive, lack in maintainability, and introduce new risks to dependability, as many aspects have to be realised from scratch. Although some special purpose systems have successfully demonstrated promising healing abilities, fundamental research and experience with general self-management concepts are still missing, so that conceptual progress is difficult and risks arise from applying self-management techniques. Reusable architectural concepts are a more promising way to achieve higher dependability and trustworthiness through self-management with low risks.

Runtime Models as a Key Aspect of Systematic Self-management

Runtime models provide a view on (primarily static) parts of software architecture in combination with *operational data* observed from the running system. For instance, a particular runtime model could be a view on the internal service architecture of a system and connect it to current response times. Using runtime models as the basis for systematic self-management has two benefits:

1) it *isolates* the system aspects that are relevant to the self-management functionality from the primary system,
2) it provides a reusable, application-independent *interface* for self-management functions such as reconfiguration, monitoring, and diagnosis.
3) it enables *model-driven* development and tool support for the integration of self-management functionality into software systems.

This work is supported by the German Research Foundation (DFG), grant GRK 1076/1

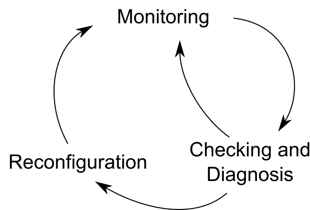

Fig. 1. The self-management cycle

Overview

We describe the general design elements of self-managing software systems in Section II. Section III describes the generation of runtime models from the specifications of constraints, monitoring, and reconfiguration operations. Section IV proposes a partitioning of the overall design of a self-managing software system into different sub-models, and sketches an approach for generating self-management from these models.

II. Design Elements of Self-managing Software Systems

Self-management comprises the activities monitoring, checking and diagnosis, and reconfiguration (see Figure 1). This first activity is monitoring, which observes the system under management, which we denote as primary system. The monitoring is performed via sensors which collect relevant operational characteristics. Next, the monitoring data is aggregated and checked. Under certain conditions, the diagnosis will identify the need to trigger reconfiguration operations to change the system.

A. Monitoring

Monitoring is a process of collecting runtime data (operational data) through measurement. This includes the computation of data from primary metrics (e.g., time stamps) to the level of secondary metrics (e.g., average response times). A complete documentation of software monitoring (e.g., to enable model-driven instrumentation) requires a *monitoring model* that specifies (1.) what has to be measured in terms of detailed metrics, (2.) where to measure (connection to software architecture), and (3.) when to measure.

B. Checking and Diagnosis

Checking and verification of the system behaviour requires a specification that describes acceptable and unacceptable behaviour.

There are several architectural viewpoints, commonly used in software engineering, which can be used to organise the system architecture in different views (compare [3]):

- data view (i.e., current values variables or databases),
- structural view (i.e., class, object, component, or process configuration models),
- dynamic view (e.g., interaction models such as (timed) protocol specifications), and
- deployment view that maps architectural entities to resources (deployment contexts).

These views are defined in an abstract sense, the information from the views is manifested in different models. A model may cover multiple views, which is indeed necessary to relate the views to each other. For example, UML Sequence Diagrams lay a strong emphasis on the dynamic view, but need to refer to classes and objects which belong to the structural view.

Constraints can be defined for any model, and thus they can refer to any view. The UML, e.g., provides the Object Constraint Language (OCL) to specify constraints that refer to the entities defined in the UML Metamodel and their attributes and operations. A typical way to specify operational goals or requirements is to define constraints in languages based on first order predicate logics, e.g., $\mathrm{employee.salary} < 40$; $2 \leq \mathrm{number_of_peers_per_super_peer} \leq 5$; $\mathrm{maximum_end_to_end_response_time} < 500\mathrm{ms}$; $\mathrm{max_clients_per_server} \leq 500$.

C. Reconfiguration

Checking and diagnosis is followed by the selection and execution of reconfiguration operations. The strategy for selecting a particular operation will depend on the system property that is under self-management and might involve the use of a prediction method in order to select a suitable reconfiguration operation and an optimal reconfiguration time [4]. For load balancing, queueing models could be used to change routing strategies, for example.

III. Runtime models as abstraction layer for self-management

A runtime model is a *causally connected* [5] representation of a software system under operation, i.e. changes in the system are propagated to the model and vice versa (see Figure 2). Therefore, there have to be mechanisms for maintaining (weak) consistency between the system and its representation. Changes in the (state, structure, or behaviour) of primary system are automatically propagated via monitoring into the runtime

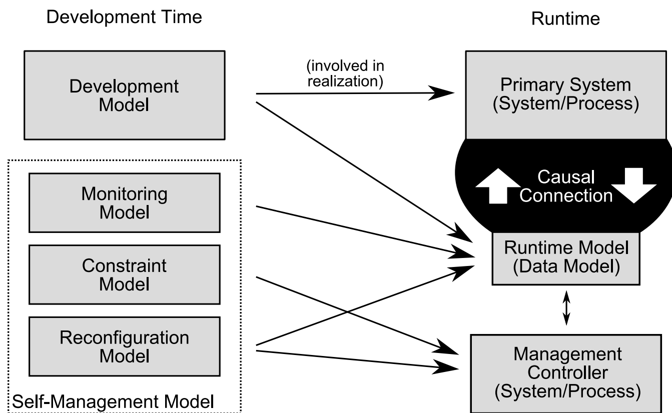

Fig. 2. The runtime model reflects both the current state of the system and provides reconfiguration functionality by a causal connection to architectural entities. The structure of the runtime model is given by the parts of the development model that are instrumented by monitoring, and on the parts of the development model that are involved in reconfiguration operations. The values in the runtime model are provided by monitoring.

model, and changes to the runtime model, i.e. reconfigurations via the self-management controller, enact the corresponding changes in the primary system.

The characterisation conforms to recent work on dynamic and self-adaptive software architectures. Oreizy et al. [6] have used a runtime model, that reflects the current structural state of the system architecture. An architectural reconfiguration operation (e.g., adding or removing software components) is first preformed on the runtime model, and afterwards this change is propagated into the primary system. The aim was to allow system maintenance without the need to restart the system. Garlan et al. [7] extended Oreizy et al.'s approach by using the runtime model as a simplified (role-based) abstraction of the system, which is used for constraint checking, diagnosis, and reconfiguration. They model mainly the structural view for representing the primary system. Failure detection is implemented by verification of constraints defined in the ADL Armani [8].

The general advantages of using a simplified runtime model for self-management are (1.) that it simplifies constraint checking, diagnosis, and the selection of repair operations, and (2.) the decoupling of self-management from the primary software system allows to develop reusable self-management strategies independently from a concrete software architecture.

IV. An approach for the specification and generation of self-management functionality

A. Specification of self-management

A *constraint model* is a specification of constraints (using the different views). As pointed out above, the different architectural views provide the subjects to constrain. The constraint model has to reference a *monitoring model* to establish the connection to system characteristics such as response times. The constraint model and the monitoring model share the references to architectural entities.

The monitoring model can be used to generate the instrumentation for collecting (and aggregating) data about the system behaviour. The constraint model could be directly used to check the collected data for conformance. No code generation is required for this, as the logic required for checking can be encapsulated into a component that is independent from the concrete system.

Reconfiguration is the last step of self-management. A *reconfiguration model* specifies *what* should happen under certain conditions. This could be expressed in terms of if-then rules, where the "if"-clause would often references to constraints from the constraint model. If further runtime information about the primary system would be required, then this should be done by references to the monitoring model. For the "then" clause, implementation-independent reconfiguration operations are specified.

B. A model-driven development approach for self-management and runtime models

The specifications of the constraints, monitoring, and reconfiguration completely define platform-independent self-management. To make self-management operational, several elements have to be realised: (1.) monitoring instrumentation, (2.) constraint checking functionality, (3.) a runtime model, (4.) management instrumentation. However, all these elements can be either be generated or can be reused for different applications (as indicated by the horizontal arrows in Figure 2).

Current research on model-driven monitoring (e.g., [9]) suggests that it is feasible to efficiently generated the monitoring instrumentation.

The constraint checking functionality is not application-dependent and could be provided by reusable middleware components. The constraint checking functionality connects constraints formulated as described before using a predicate logic based on the four architectural view, in reference to the monitoring model. So it is the challenge to define a suitable

meta-model for monitoring and constraint models. Existing constraint languages such as Armani [8] or the Object Constraint Language (OCL) [10] provide a promising basis, but lack support for timed constraints, for example.

There is much existing research that can already be used to realise (and generate) a runtime model as a runtime artifact. For instance, using object oriented data-structures, existing reflection mechanisms from languages such as Java, or data-structures that base on existing ADLs such as Acme [11] (as used by [7]). A remaining challenge will be to define a meta-model for runtime models for specific concerns to reduce the complexity for managing them during runtime.

Many different kinds of reconfiguration operations can be distinguished and some might just involve the change of single values of the business data, while others affect the structure or behaviour of the system. A large class of adaptation operations (e.g., adding and removing software components) might be already provided by the component platform (see [4]), so that only a mapping from the platform-independent reconfiguration-model to the platform-specific functionality would have to be generated. A remaining challenge is to enable additional reconfiguration operations by the middleware, such that model-driven development of self-management functionality will be limited to a high level mapping.

V. Summary and Outlook

The technical building blocks for self-managing software systems are already available. Anyhow, the last decades have shown that the manual development of such systems is very complex. We outlined an approach for the model-driven development of a self-managing software system that can simplify the realisation task by the specification of self-management on the model level. A key aspect of the approach are runtime models, which reflect the system aspects that are relevant for self-management. We have discussed that the structure of the runtime models can be generated from the self-management models and the development model of the primary system.

The complete realisation of our (or a similar) approach includes the open challenge to design (and generate) suitable runtime models based on the needs given by specifications of monitoring, constraint checking and diagnosis, and (for enabling self-management) reconfiguration operations. The concrete relations, and properties of the languages for expressing these specifications are largely unexplored and a major topic for future research, with the potential to build a conceptual bridge between development models and runtime models.

References

[1] Rohr, M., Giesecke, S., Hasselbring, W., Hiel, M., van den Heuvel, W.J., Weigand, H.: A classification scheme for self-adaptation research. In: International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS06) Poster Session, Erfurt, Germany. (2006)

[2] Wensley, J.H., Lamport, L., Goldberg, J., Green, M.W., Levitt, K.N., Melliar-Smith, P.M., Shostak, R.E., Weinstock, C.B.: SIFT: Design and analysis of a fault-tolerant computer for aircraft control. Proceedings of the IEEE **66**(10) (1978) 1240–1255

[3] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architecture. Addison-Wesley, Boston (2002)

[4] Matevska-Meyer, J., Olliges, S., Hasselbring, W.: Runtime reconfiguration of J2EE applications. In: Proceedings of DECOR'04 - 1st French Conference on Software Deployment and (Re)Configuration, Grenoble, France, University of Grenoble (2004) 77–84

[5] Maes, P.: Concepts and experiments in computational reflection. In: Conference proceedings on Object-oriented programming systems, languages and applications (OOPSLA '87), New York, NY, USA, ACM Press (1987) 147–155

[6] Oreizy, P., Gorlick, M., Taylor, R., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An architecture-based approach to self-adaptive software. Intelligent Systems and Their Applications **14**(3) (1999) 54–62

[7] Garlan, D., Cheng, S.W., Schmerl, B.: Increasing system dependability through architecture-based self-repair. In: Architecting Dependable Systems. Volume 2677 of Lecture Notes in Computer Science., Springer (2003) 23–46

[8] Monroe, R.T.: Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University (2000) Version 2.3.

[9] Boskovic, M., Warns, T., Hasselbring, W.: Model Driven Instrumentation for Relational Event Traces. Radioelektronic and Computer Systems **6**(18) (2006) 124–129

[10] OMG Object Management Group: UML 2.0 object constraint language (OCL) specification (2003)

[11] Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In Leavens, G.T., Sitaraman, M., eds.: Foundations of Component-Based Systems. Cambridge University Press, New York, NY (2000) 47–67