Engineering and Continuously Operating Self-Adaptive Software Systems: Required Design Decisions

André van Hoorn[1], Wilhelm Hasselbring[2], and Matthias Rohr[1,3]

[1] Graduate School TrustSoft, University of Oldenburg, D-26111 Oldenburg
[2] Software Engineering Group, University of Kiel, D-24098 Kiel
[3] BTC AG – Business Technology Consulting AG, D-26121 Oldenburg

Abstract: Self-adaptive or autonomic systems are computing systems which are able to manage/adapt themselves at runtime according to certain high-level goals. It is appropriate to equip software systems with adaptation capabilities in order to optimize runtime properties, such as performance, availability, or operating costs. Architectural models are often used to guide system adaptation. When engineering such systems, a number of cross-cutting design decisions, e.g. instrumentation, targeting at a system's later operation/maintenance phase must and can be considered during early design stages.

In this paper, we discuss some of these required design decisions for adaptive software systems and how models can help in engineering and operating these systems. The discussion is based on our experiences, including those gathered from evaluating research results in industrial settings. To illustrate the discussion, we use our self-adaptation approach SLAstic to describe how we address the discussed issues. SLAstic aims to improve a software system's resource efficiency by performing architecture-based runtime reconfigurations that adapt the system capacity to varying workloads, for instance to decrease the operating costs.

1 Introduction

Self-managed or autonomic systems are those systems which are able to adapt themselves according to their environment [11]. Self-adaptation can be described as a cycle of three logical phases [19]: observation, analysis, and adaptation. The observation phase is concerned with monitoring (e.g., system behavior or system usage). The analysis phase detects triggers for adaptation and, if required, selects suitable adaptation operations, which are executed in the adaptation phase. A recent survey on self-adaptation research can be found in [5].

When engineering such systems, various non-functional aspects need to be considered. Trade-offs between QoS (Quality of Service) requirements, such as a performance, availability, and operating costs have to be addressed. Thus, requirements of system operation have a significant impact on the required design decisions of system engineering.

We illustrate the discussion of required design decisions based on the description of our self-adaptation approach SLAstic [24]. SLAstic aims to improve the resource efficiency

of component-based software systems, with a focus on business-critical software systems. Architectural models, specifying both functional and non-functional properties, play a central role in the SLAstic approach as they are not only used for the specification of the system at design time. The same models are updated by measurement data at runtime and used for the required analysis tasks in order to achieve the self-adaption goals. SLAstic explicitly considers a flexible system capacity in the software architecture.

The contribution of this paper is a discussion of design decisions that are required at *design time* for an effective operation at *runtime* of continuously operating software systems, such as business-critical software systems. We discuss the role of models in this context. This discussion is based on our experience and lessons learned from lab studies and industrial field studies with our monitoring framework Kieker [20].[1] We illustrate how these design decisions are addressed in our SLAstic approach for adaptive capacity management.

This paper is structured as follows: Section 2 describes the architecture-based self-adaptation approach SLAstic aiming to support a resource-efficient operation of software systems. Based on this, Section 3 discusses design decisions for engineering and operating self-adaptive software systems and how they are addressed in the SLAstic approach. The conclusions are drawn in Section 4.

2 The SLAstic Framework

With SLAstic, resource efficiency of continuously operating software is improved by adapting the number of allocated server nodes and by performing fine-grained architectural reconfigurations at runtime according to current system usage scenarios (workload) while satisfying required external QoS objectives (as specified in service level agreements). It is not the goal of this paper to present the SLAstic framework in detail. Instead, it is intended to be used for illustration in our discussion in Section 3.

2.1 Assumptions on Software System Architectures

We assume a hierarchical software system architecture consisting of the following types of hardware and software entities: (1) *server node*, (2) *component container*, and (3) *software component*. This allows to design the SLAstic self-adaptation framework in a way that provides reusability among different system implementations sharing common architectural concepts. Figure 1 illustrates this hierarchical structure with a Java EE example.

The business logic of a software system is implemented in a number of software components with well-defined provided and required interfaces. Software components constitute units of deployment [22] and can be assembled or connected via their interfaces. Through their provided interfaces, they provide services to other software components or systems. Software components are deployed into component containers, which constitute

[1] `http://kieker.sourceforge.net`

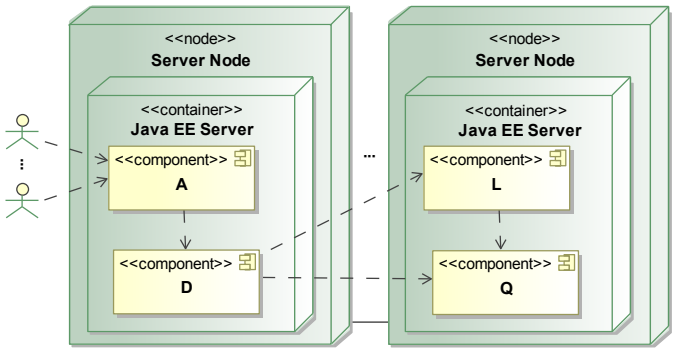


Figure 1: Illustration of a hierarchical software system architecture

their execution context and provide middleware services, such as transaction management. Moreover, a component container manages the thread pool used to execute external service requests. Web servers or Java EE application servers are typical examples for component containers. A component container is installed on a server node consisting of the hardware platform and the operating system. Server nodes communicate via network links. For a software system, a set of server nodes is allocated from a *server pool*. We denote the set of allocated server nodes, the assembly of software components, as well as its deployment to component containers as the software system's *architectural configuration*.

Structural and behavioral aspects of a software system's architecture can be described using architecture description languages (ADL). Components, interfaces, and connectors for architectural configurations are common (structural) features of ADLs [16]. A recent survey of ADLs can be found in [23].

2.2 Framework Architecture

The SLAstic framework aims to equip component-based software systems, as described in the previous Section 2.1, with self-adaptation capabilities in order to improve its resource efficiency. Figure 2 depicts how the concurrently executing SLAstic components for monitoring (SLAstic.MON), reconfiguration (SLAstic.REC), and adaptation control (SLAstic.CONTROL) are integrated and how they interact with the monitored software system. It shows the typical, yet rather general, external control loop for adaptation [9].

The software system is instrumented with monitoring probes which continuously collect measurement data from the running system. The SLAstic.MON component provides the monitoring infrastructure and passes the monitoring data to the SLAstic.CONTROL component. The SLAstic.CONTROL component, detailed later in Figure 4 of Section 3.2, analyzes the current architectural configuration with respect to the monitoring data and, if required, determines an adaptation plan consisting of a sequence of reconfiguration operations. The adaptation plan is communicated to the SLAstic.REC component which

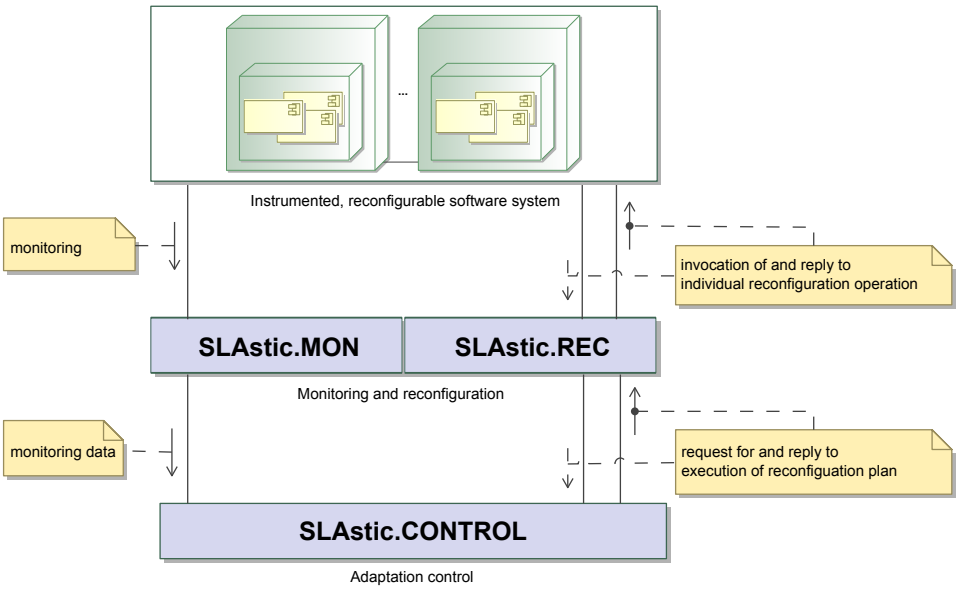


Figure 2: SLAstic self-adaptation system overview

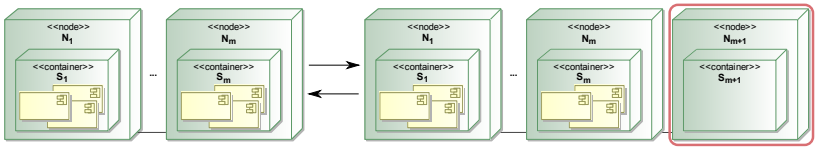
is responsible for executing the actual reconfiguration operations. SLAstic employs our Kieker framework for continuous monitoring and online analysis.

2.3 Reconfiguration Operations

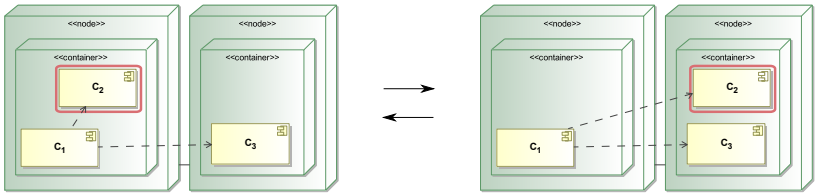
Although, the SLAstic framework, in principle, supports arbitrary architectural reconfiguration operations, we will restrict us to the following three operations, illustrated in Figure 3.

(1) **Node Allocation & Deallocation.** A server node is allocated or deallocated, respectively. In case of an allocation, this includes the installation of a component container, but it does not involve any (un)deployment operation of software components. Intuitively, the goal of the allocation is the provision of additional computing resources and the goal of the deallocation is saving operating costs caused by power consumption or usage fees, e.g. in cloud environments.

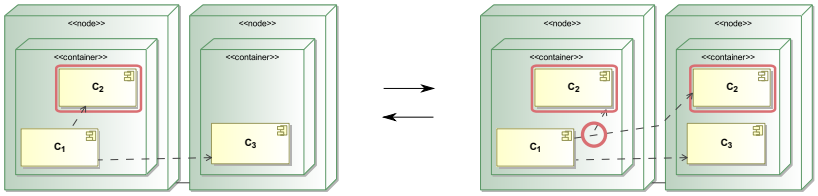
(2) **Component Migration.** A software component is undeployed from one execution context and deployed into another. The goals of this fine-grained application-level operation are both to avoid the allocation of additional server nodes or respectively to allow the deallocation of already allocated nodes by executing adaptation operation (1).



(a) node (de-)allocation



(b) component migration



(c) component (de-)replication

Figure 3: SLAstic reconfiguration operations

(3) **Component (de-)Replication** This application-level operation consists of the duplication of a software component and its deployment into another execution context (as well as the reverse direction). Future requests to the component are distributed between the available component instances. The goals of this application-level operation are the same as the goals of operation (2).

3 Required Design Decisions and the Role of Models

The following three Subsections 3.1–3.3 discuss design decisions and the role of models related to monitoring, analysis, and adaptation, respectively.

3.1 Monitoring

In order to obtain the required information to control adaptation, a system needs to be instrumented with monitoring probes. Monitoring can be performed at the different abstraction levels of a software system, i.e., platform (e.g., CPU utilization), container (e.g., number of available threads in a thread pool), and application (e.g., service response times)

level. For example, SLAstic continuously requires performance data, such as CPU utilization, workload (system usage), and service response times.

Since adaptation concerns a running system in production use, continuous monitoring of the system state is required. We argue that the integration of monitoring should be designed in a systematic way and although mainly becoming relevant not before a system's operation phase, its integration should be considered early in the engineering process. In the following paragraphs we will discuss design decisions regarding continuous monitoring. In [7], we present a process model for application-level performance monitoring of information system landscapes. The design decisions should be integrated into such an engineering process.

Selection of Monitoring Probes A monitoring probe contains the logic which collects and possibly pre-processes the data of interest from within the application. Probes can measure externally visible behavior, e.g. service response times, but also application-internal behavior, such as calling dependencies between components.

SLAstic requires different types of monitoring probes, collecting workload, resource usage, and timing data. An example not focusing on self-adaptation is application level failure diagnosis [1, 14] requiring probes for monitoring response times and internal control flow of operation executions.

In practice, new application-internal monitoring probes are often only introduced in an ad-hoc manner, as a result of a system failure. For example, a probe for monitoring the number of available database connections in a connection pool may have been introduced. The selection of the types of monitoring probes must be driven by the goal to be achieved by the gathered monitoring data and depends on the analysis concern.

Number and Position of Monitoring Points In addition to the above-mentioned decision of what types of monitoring probes are integrated into the system, important and very difficult decisions regard the number and the exact locations of monitoring points. This decision requires a trade-off between the information quality available to the analysis tasks and the overhead introduced by possibly too fine-grained instrumentation leading to an extensive size of the monitoring log. As the type of monitoring probes to use, does the number and position depend on the goal of monitoring. Additionally, the different usage scenarios of the application must be considered, since an equally distributed coverage of activated monitoring points during operation is desirable.

Intrusiveness of Instrumentation A major maintainability aspect of application-level monitoring is how monitoring logic is integrated into the business logic. Maintainability is reduced if the monitoring code is mixed with the source code of the business logic, because this reduces source code readability.

We regard the use of the AOP (Aspect-Oriented Programming) paradigm [12] as an extremely suitable means to integrate monitoring probes into an application [8]. A popular Java-based AOP implementation is AspectJ. Many middleware technologies provide sim-

ilar concepts. Examples are the definition of filters for incoming Web requests in the Java Servlet API, the method invocation interceptors in the Spring framework, or handlers for incoming and outgoing SOAP messages in different Web service frameworks. Kieker currently supports AOP-based monitoring probes with AspectJ, Spring, Servlets, and SOAP.

Physical Location of the Monitoring Log The monitoring data collected within the monitoring probes is written to the so-called monitoring log. The monitoring log is typically located in the filesystem or in a database. The decision which medium to use depends on the amount of monitoring data generated at runtime, the required timeliness of analysis, and possibly restricted access rights or policies. A filesystem-based monitoring log is fast, since usually no network communication is required. The drawback is that online analysis of the monitoring data is not possible or at least complicated in a distributed setting. A database brings the benefit of integrated and centralized analysis support such as convenient queries but is slower than a file system log due to network latencies and the overhead introduced by the DBMS. Moreover, the monitoring data should not be written to the monitoring log synchronously since this has a considerable impact on the timing behavior of the executing business service.

Since SLAstic performs the analysis online, it requires fast and continuous access to recent monitoring data. For this reason, the SLAstic.MON and SLAstic.CONTROL components communicates monitoring data via JMS messaging queues. Kieker includes different synchronous and asynchronous monitoring log writers for filesystem, database, and for JMS queues.[2] Customized writers can be integrated into Kieker.

Monitoring Overhead It is clear that continuous monitoring introduces a certain overhead to the running system. Of course, the overall overhead depends on the number of activated monitoring points and its activation frequency. The overhead for a single activated monitoring point depends on the delays introduced by the resource demand and process synchronizations in the monitoring probes, the monitoring control logic, and particularly I/O access for writing the monitoring data into the monitoring log.

It is a requirement that the monitoring framework is as efficient as possible and that the overhead increases linearly with the number of activated monitoring points, i.e., each activation of a monitoring point should add the same constant overhead. Of course, this linear scaling is only possible up to a certain point and depends on the average number of activated monitoring, i.e., the granularity of instrumentation.

Kieker has been used to monitor some production systems in the field. For these systems, no major impact was reported, despite the fact that detailed trace information is monitored for each incoming service request. In benchmarks, we observed an overhead that was in the order of microseconds on modern desktop PCs for each activated instrumentation point. We are currently working on a systematic assessment of the overhead introduced by the Kieker monitoring framework.

[2] `http://java.sun.com/products/jms/`

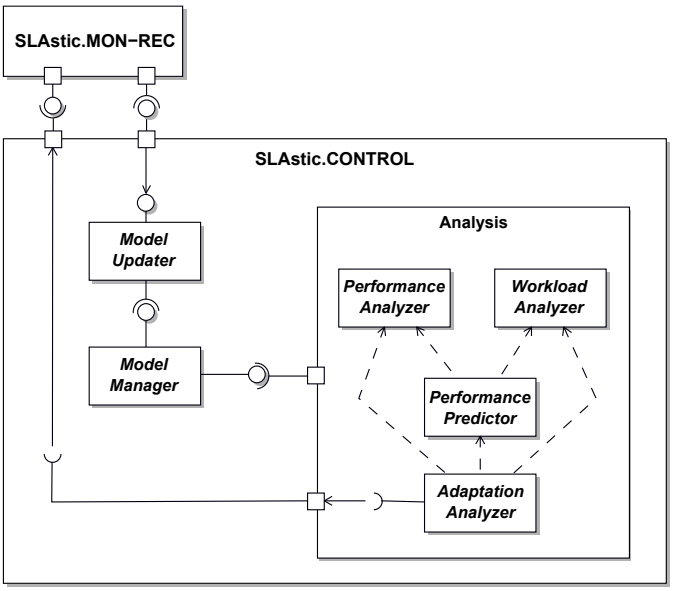


Figure 4: Internal architecture of the SLAstic.CONTROL component

Model-Driven Instrumentation Model-driven software development (MDSD) [21] provides a convenient means to lift the level of abstraction for instrumentation from source code to the architectural or even business level. Static or dynamic design models can be annotated with monitoring information whose transformation into the platform-specific instrumentation can be integrated into the MDSD process. Possible annotation variants are tagging, i.e., adding the instrumentation to the functional part of the model, and decoration, i.e., the definition of a dedicated monitoring model referring to the architectural entities captured in external models. Examples for the model-driven instrumentation for monitoring SLAs employing (standardized) meta-models (e.g., [6, 18]), come from the domain of component-based [4] and service-based systems [17].

3.2 Analysis

Analysis or adaptation control involves analyzing and, if required, planning and triggering the execution of system adaptation. Analysis and planning is based on architectural knowledge about the system, the interpretation of monitoring data, and specified adaptation. Analysis and planning may be performed autonomically or on-demand and manually by the system administrator.

SLAstic analyzes the instrumented and reconfigurable system while it is running, i.e. online. The internal architecture of the SLAstic.CONTROL component with its internal, concurrently executing subcomponents is shown in Figure 4.

Runtime Models Design models can be used as or transformed into architectural runtime models constituting an abstract representation of the current state and thus being usable as a basis for analysis. Models more suitable for the performed analysis methods can be automatically derived from the architectural models. For example, for performance prediction, a number of transformations from design models to analysis models, such as variants of queueing networks, were developed [2].

The architectural SLAstic runtime models for the analysis are accessible through the *Model Manager* component which synchronizes access to these models in order to maintain consistent model state. The *Model Updater* component updates the runtime models based on the monitoring data received from the SLAstic.MON component. In the SLAstic framework, we are planning to use (extended) model instances of the Palladio metamodel [3], a domain-specific ADL designed for performance prediction of component-based software systems.

Runtime Analysis Methods A challenging decision is the selection of appropriate runtime analysis methods, which depends on a number of factors, such as the arrival rate of incoming monitoring data, variability in the data, available computational resources, required accuracy of the analysis et cetera. The integration of the adaptation control in a feedback loop with the controlled system, usually requires timely analysis results and adaptation decisions.

The *Performance Predictor* in the SLAstic.CONTROL component predicts the performance of the current architectural configuration based on the performance and workload analysis, including a forecast of the near-future workload, performed by the *Performance Analyzer* and *Workload Analyzer* components. The *Adaptation Analyzer* determines an adaptation plan which is communicated to the SLAstic.REC component for execution. The *Model Updater* updates the runtime model with respect to the new architectural configuration. A number of analysis methods is imaginable in each of the analysis components. The SLAstic framework allows to evaluate different analysis methods by its extendable architecture.

3.3 Adaptation

The actual execution of an adaptation plan involves the execution of the included architectural reconfiguration operations. The reconfiguration can be performed offline or at runtime. Offline reconfiguration means, that the running system is shut down, reconfigured, and restarted. When reconfigured at runtime, a system is changed without a restart, and while it is running and serving requests. In practice, adaptation or maintenance, e.g. for system repair, is usually performed offline. Many companies have fixed “maintenance windows”, i.e., timeframes during which the provided services are not available and a new version of the software system is deployed. This section focuses on reconfiguration at runtime.

Specification of Reconfiguration Operations State charts, or other modeling techniques can be used to specify how reconfiguration operations are executed, e.g., in terms of a reconfiguration protocol. The benefit of having a formal protocol is that its correctness can be verified using formal analysis, and that quantitative analyses regarding the reconfiguration may be performed. This is helpful in determining an adaptation plan using certain reconfiguration properties. For example, regarding the SLAstic *Node Allocation* operation (see Section 2.3), the average time period between an allocation request for a node and its readiness for operation.

In this paper, we focus on business-critical software systems. In such systems, service requests are usually executed as transactions. A specification of the SLAstic component migration operation would include a definition of how transactions are handled. For example, when migrating a software component, active transactions using the component to be migrated would be completed and newly arriving transactions would be dispatched to the migrated instances. In [15], we described our J2EE-based implementation of a redeployment reconfiguration operation, that is based on a reconfiguration protocol defined using state charts.

Design of Reconfigurable Software Architectures Reconfiguration operations can be defined based on architectural styles [10]. This allows to re-use the reconfiguration operations for systems having this architectural style. The SLAstic reconfiguration operations can be applied to systems complying to the architectural assumptions sketched in Section 2.1.

This reconfiguration capability needs to be integrated into the software system architecture. Likewise to the annotation of design models with information regarding monitoring instrumentation, as described in the previous Section 3.1, reconfiguration-specific annotations can be added to design models. For example, software components could be annotated with the information whether they are designed to be replicated or migrated at runtime using the SLAstic reconfiguration operations.

Transactional Reconfiguration An adaptation plan should be executed as a transaction, in order to bring the system from one consistent architectural configuration into another [13]. This means, that an adaptation plan is only committed if all included reconfiguration operations were successfully executed. In SLAstic we will only permit the execution of one adaptation plan at a time.

4 Conclusions

Requirements of system operation have a significant impact on the required design decisions in system engineering. In this paper, we discussed the design decisions that are required at *design time* for an effective operation at *runtime* of continuously operating software systems, such as business-critical software systems. This discussion is based on our experiences and lessons learned from lab studies and industrial field studies with our

monitoring framework Kieker, and illustrated using our self-adaptation approach SLAstic. SLAstic aims to improve the resource efficiency of component-based software systems.

Adaptation impacts running systems in production use, thus, continuous monitoring of the system state is required if the systems are business-critical. The integration of monitoring should be considered early in the engineering process. Our message is that requirements of system operation should be considered early in the design process, not as an afterthought as often observed in practice. Architectural models should be used both at design and runtime. However, many of the discussed design decisions are not only relevant to online self-adaptation, but also to offline maintenance activities. Concerns of self-adaptation should, similar to monitoring, be considered as cross-cutting concerns.

References

[1] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Problem determination using dependency graphs and run-time behavior models. In *15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2004)*, volume 3278 of *LNCS*, pages 171–182, 2004.

[2] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[3] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[4] K. Chan and I. Poernomo. QoS-aware model driven architecture through the UML and CIM. *Information Systems Frontiers*, 9(2-3):209–224, 2007.

[5] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*. Springer, 2009.

[6] Distributed Management Task Force. Common Information Model (CIM) standard. `http://www.dmtf.org/standards/cim/`, May 2009.

[7] T. Focke, W. Hasselbring, M. Rohr, and J.-G. Schute. Ein Vorgehensmodell für Performance-Monitoring von Informationssystemlandschaften. *EMISA Forum*, 27(1):26–31, Jan. 2007.

[8] T. Focke, W. Hasselbring, M. Rohr, and J.-G. Schute. Instrumentierung zum Monitoring mittels Aspekt-orientierter Programmierung. In *Tagungsband Software Engineering 2007*, volume 106 of *LNI*, pages 55–59. Gesellschaft für Informatik, Bonner Köllen Verlag, Mar. 2007.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[10] D. Garlan, S.-W. Cheng, and B. R. Schmerl. Increasing system dependability through architecture-based self-repair. In *Architecting Dependable Systems*, volume 2677 of *LNCS*, pages 61–89. Springer, 2003.

[11] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, Jan. 2003.

[12] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 2007 European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[13] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[14] N. S. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*, pages 47–57. IEEE, Mar. 2009.

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

[16] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[17] C. Momm, T. Detsch, and S. Abeck. Model-driven instrumentation for monitoring the quality of web service compositions. In *Proceedings of the 2008 12th Enterprise Distributed Object Computing Conference Workshops (EDOCW '08)*, pages 58–67, Washington, DC, USA, 2008. IEEE Computer Society.

[18] OMG. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. `http://www.omg.org/spec/QFTP/`, Apr. 2008.

[19] M. Rohr, S. Giesecke, W. Hasselbring, M. Hiel, W.-J. van den Heuvel, and H. Weigand. A classification scheme for self-adaptation research. In *Proceedings of the International Conference on Self-Organization and Autonomous Systems in Computing and Communications (SOAS'2006)*, Sept. 2006.

[20] M. Rohr, A. van Hoorn, J. Matevska, N. Sommer, L. Stöver, S. Giesecke, and W. Hasselbring. Kieker: Continuous monitoring and on demand visualization of Java software behavior. In *Proceedings of the IASTED International Conference on Software Engineering 2008 (SE 2008)*, pages 80–85. ACTA Press, Feb. 2008.

[21] T. Stahl and M. Völter. *Model-Driven Software Development – Technology, Engineering, Management*. Wiley & Sons, 2006.

[22] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2. edition, 2002.

[23] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. John Wiley & Sons, Inc., 2009.

[24] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In N. Medvidovic and T. Tamai, editors, *Proceedings of the 2nd Warm-Up Workshop for ACM/IEEE ICSE 2010 (WUP '09)*, pages 41–44. ACM, Apr. 2009.