FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# Adaptation and Evolution of Service-Based Applications in Cloud Computing Environments

DISSERTATION

zur Erlangung des akademischen Grades

**Doktor der technischen Wissenschaften**

eingereicht von

**Christian Inzinger**
Matrikelnummer 0225558

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Schahram Dustdar

Diese Dissertation haben begutachtet:

(Univ.Prof. Schahram Dustdar) (Prof. Luciano Baresi)

Wien, 03.02.2014

(Christian Inzinger)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# Adaptation and Evolution of Service-Based Applications in Cloud Computing Environments

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

**Doktor der technischen Wissenschaften**

by

**Christian Inzinger**
Registration Number 0225558

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Schahram Dustdar

The dissertation has been reviewed by:

(Univ.Prof. Schahram Dustdar) (Prof. Luciano Baresi)

Wien, 03.02.2014

(Christian Inzinger)

# Erklärung zur Verfassung der Arbeit

Christian Inzinger
Gruschaplatz 2/9, 1140 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

# Acknowledgements

The research leading to this thesis has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 257483 (INDENICA), and from the Austrian Science Fund (FWF) under grant P23313-N23 (Audit 4 SOAs).

# Danksagung

Zuallererst möchte ich meinem Betreuer, Prof. Schahram Dustdar, für seine Unterstützung und die Chance danken, meine Dissertation in der Distributed Systems Group (DSG) in einer Arbeitsgruppe mit exzellenter Arbeitsatmosphäre zu schreiben, in der ich meine Forschungsideen eigenständig verfolgen und entwickeln konnte. Weiters danke ich Prof. Luciano Baresi für viele interessante Diskussionen und für die Zweitbegutachtung dieser Arbeit.

Ich bedanke mich auch bei allen Kollegen in der DSG, die mich über die letzten Jahre mit großartiger Zusammenarbeit begleitet haben. Eine Dissertation ist nie die Leistung nur einer einzelnen Person, deshalb geht an dieser Stelle besonderer Dank an Waldemar Hummer, Benjamin Satzger und Philipp Leitner, die am Großteil meiner Forschungsarbeiten beteiligt waren, und durch Ihren Einsatz die vorliegene Arbeit mitgeformt haben.

Ganz besonders möchte ich mich auch bei meiner Familie und all meinen Freunden bedanken, die mich über die Jahre durch alle Höhen und Tiefen begleitet haben, und mich zu dem Menschen machten, der ich heute bin.

# Abstract

The emergence of the Service Oriented Architecture (SOA) paradigm enabled software architects to efficiently design applications based on the composition of loosely coupled services. However, maintaining such Service-Based Applications (SBAs) over time still poses several challenges. SBAs are expected to successfully perform business tasks in changing environments. Unexpected problems need to be handled gracefully by application control policies. Hardware failures, software issues, and changes in execution environments, such as modifications of partner services, should not lead to service disruptions. Hence, SBAs must be designed for continued functional evolution to account for changing business and technical requirements. The utility-oriented cloud computing paradigm opens up novel possibilities for applications reacting to changes in their environment. The possibility to quickly and easily provision computing resources relieves application architects and operators from having to statically provision infrastructure for peak usage. This allows for the implementation of elastic applications that dynamically adjust their resource usage to current demand given appropriate control policies. The on-demand nature of cloud offerings makes even significant evolutionary changes to an application's architecture feasible. To successfully implement SBAs that can predictably react to changes in their environment and can be safely evolved, practitioners must be able to effectively model, monitor and control relevant application aspects to properly document and execute application adaptation and evolution.

This thesis contributes a set of novel approaches for SBA evolution and adaptation in cloud environments. We introduce a holistic framework for enabling structured evolution and adaptation of SBAs throughout the complete software development process. A novel evolution lifecycle model and accompanying strategies allow for unified handling of change requests in any lifecycle phase, and facilitate the propagation of necessary changes between phases in a controlled manner. We present a method for provider-managed adaptation that enables customers to leverage provider experience managing complex distributed systems without requiring large upfront investments. Using a novel Domain-Specific Language (DSL) to model applications and their control structure, SBAs can effectively and efficiently react to changes in their environment without operators needing to implement custom solutions. To mitigate the effects of unexpected changes in application execution environments, we present an approach based on machine learning techniques to incrementally improve adaptation policies. Finally, we introduce a method for automated identification of service implementation incompatibilities using pooled decision trees for localizing faulty service parameter and binding configurations, explicitly addressing transient and changing fault conditions. The results of our investigations are evaluated based on multiple case studies and show that our approaches can significantly contribute to facilitate structured evolution of SBAs and increase system robustness by autonomically improving adaptation policies.

# Kurzfassung

Serviceorientierte Architektur (SOA) hat sich in den vergangenen Jahren als beliebtes Paradigma zur effizienten Entwicklung von Anwendungen etabliert, basierend auf der Komposition von lose gekoppelten Komponenten. Die Verwaltung solcher Servicebasierten Anwendungen (SBAs) stellt Anwender jedoch weiterhin vor einige Herausforderungen, da SBAs im Allgemeinen über lange Zeit im Einsatz bleiben, und deren Funktionalität trotz Veränderungen in deren Umgebung, oder auftretenden Fehlern, zur Verfügung stellen müssen. Daher müssen SBAs für die funktionelle Weiterentwicklung, oder Evolution, entworfen werden, um sich ändernden Geschäftsprozessen und technischen Anforderungen anpassen zu können. Das Cloud-Computing-Paradigma eröffnet neue Möglichkeiten für SBAs, um auf Veränderungen in ihrer Umgebung zu reagieren. IT-Infrastruktur muss nicht mehr im Vorhinein beschafft und für Spitzenlasten dimensioniert werden, sondern kann, mit Hilfe entsprechender Kontroll-Logik, einfach und schnell an aktuelle Anforderungen angepasst werden. Cloud-Dienste machen auch signifikante evolutionäre Veränderungen der Architektur einer SBA möglich. Um SBAs, die einfach weiterentwickelt und auf Änderungen reagieren können, erfolgreich zu implementieren, müssen Anwender in der Lage sein, relevante Aspekte der SBA einfach zu modellieren, zu überwachen und zu steuern.

In dieser Arbeit werden neuartige Ansätze zur Adaptierung und evolutionären Anpassung von Anwendungen in Cloud-Umgebungen vorgestellt, die den gesamten Softwareentwicklungsprozess betrachten. Das Evolution Lifecycle-Modell ermöglicht vereinheitliche Behandlung von Änderungsanforderungen in jeder Phase des Anwendungsentwicklungsprozesses und erleichtert den Austausch von relevanten Information zwischen Entwicklungsphasen. Weiters wird ein System zur Realisierung von adaptiven Systemen vorgestellt, bei dem die Adaptierung von SBAs durch Cloud-Betreiber verwaltet wird, um deren Erfahrungen mit der Verwaltung komplexer verteilter Systeme ohne große Vorabinvestitionen nutzen zu können. Mit Hilfe einer domänenspezifischen Sprache können adaptive SBAs und deren Kontrollstruktur effizient modelliert werden, ohne dass Anwender maßgeschneiderte Lösungen implementieren müssen. Um die Auswirkungen von unerwarteten Änderungen in Laufzeitumgebungen von SBAs zu minimieren, wird ein Ansatz vorgestellt, der Kontrolllogik schrittweise, mit Hilfe von Techniken des maschinellen Lernens, verbessern kann. Schließlich wird ein Verfahren zur automatischen Identifikation von Implementierungs-Inkompatibilitäten vorgestellt, das mit Hilfe von Entscheidungsbäumen fehlerhafte Parameter-Kombinationen und Partner-Service-Zuordnungen erkennen kann. Die Ergebnisse unserer Untersuchungen werden anhand mehrerer Fallstudien ausgewertet und zeigen, dass unsere Ansätze die strukturierte Weiterentwicklung von Anwendungen wesentlich erleichtern können, und deren Robustheit durch autonome Verbesserung von Adaptierungslogik signifikant erhöht werden kann.

# Contents

# List of Tables

# List of Figures

# List of Publications

The work presented in this thesis is based on research that has been published in the following conference papers, journal articles, and technical reports. For a full publication list of the author please refer to the website at http://dsg.tuwien.ac.at/staff/inzinger.

- Christian Inzinger, Waldemar Hummer, Ioanna Lytra, Philipp Leitner, Huy Tran, Uwe Zdun, and Schahram Dustdar. Decisions, models, and monitoring – A lifecycle model for the evolution of service-based systems. In *Proceedings of the 17th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '13, pages 185–194, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/EDOC.2013.29

- Christian Inzinger, Benjamin Satzger, Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Model-based adaptation of cloud computing applications. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD '13), Special Track on Model-driven Software Adaptation*, MODA '13, pages 351–355. SciTePress, 2013. doi:10.5220/0004381803510355

- Benjamin Satzger, Waldemar Hummer, Christian Inzinger, Philipp Leitner, and Schahram Dustdar. Winds of change: From vendor lock-in to the meta cloud. *IEEE Internet Computing*, 17(1):69–73, 2013. doi:10.1109/MIC.2013.19

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems. *Software: Practice and Experience*, 2014. doi:10.1002/spe.2254. (to appear)

- Christian Inzinger, Benjamin Satzger, Waldemar Hummer, and Schahram Dustdar. Specification and deployment of distributed monitoring and adaptation infrastructures. In *Proceedings of the International Workshop on Performance Assessment and Auditing in Service Computing, co-located with ICSOC '12*, PAASC '12, pages 167–178, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-37804-1_18

- Christian Inzinger, Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Non-intrusive policy optimization for dependable and adaptive service-oriented systems. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 504–510, New York, NY, USA, 2012. ACM. doi:10.1145/2245276.2245373

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Towards identifying root causes of faults in service-based applications. In *Proceedings of the 31st IEEE International Symposium on Reliable Distributed Systems*, SRDS '12, pages 404–405, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/SRDS.2012.78

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Identifying incompatible implementations of industry standard service interfaces for dependable service-based applications. Technical Report TUV-1841-2012-1, Vienna University of Technology, 2012

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Identifying incompatible service implementations using pooled decision trees. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 485–492, New York, NY, USA, 2013. ACM. doi:10.1145/2480362.2480456

CHAPTER 1

# Introduction

Over the last decade, the Service Oriented Architecture (SOA) paradigm [49, 122] found widespread adoption as a popular way to implement enterprise applications. Business concerns are modeled as self-contained services that expose their functionality via platform-independent interfaces and protocols to enable clear separation of concerns and increase reusability. The use of SOA assists software architects in creating enterprise applications by providing access to a variety of loosely-coupled services, both developed in-house as well as by external third parties, that can be easily reused and composed [45] to create complex Service-Based Applications (SBAs), or Service-Based Systems (SBSs)[1]. This loosely-coupled, service-based approach to application design is well suited for (and often mandates) distributed deployment topologies comprising multiple machines to properly handle resource load and isolate failure domains. In recent years, the SOA principles provided the basis for cloud computing [8], a service-based approach for deploying and executing large-scale distributed applications on managed infrastructure. Cloud computing relieves users from provisioning and maintaining dedicated infrastructure. Under the cloud computing umbrella term, a number of offerings are now available for consumption in a service-based manner, such as computing resources, storage, and databases.

The utility-driven, on-demand nature of cloud offerings [29] allows customers to easily and quickly provision the exact type and amount of resources needed for a given task at a given time, a concept also known as elasticity [46]. Applications created according to SOA design principles are a natural fit for deployment on cloud environments, as the service-based approach employed by cloud computing offerings integrates into and extends the already available service stack, allowing for new ways to efficiently orchestrate and manage service deployments. Using cloud techniques, SBAs can dynamically react to varying request loads by provisioning additional resources when necessary to fulfill established Service Level Agreements (SLAs), and releasing superfluous resources when they are not needed, thereby reducing operating costs. Furthermore, ready-made application components, such as databases or messaging infrastructures, can now be

[1] In the context of this thesis, we use the terms SBA and SBS synonymously to denote software systems that are designed and implemented based on SOA principles.

provisioned on demand, and, since they are usually managed by the provider, significantly reduce operational management efforts. This allows software teams to focus on creating value for their customers instead of dealing with the intricacies of deploying, configuring, and maintaining such components.

The cloud computing paradigm moreover serves as enabling technology for iterative and agile software development methods such as the Rational Unified Process [90] and Scrum [151]. Iterative and agile software development methods focus on improving communication with relevant stakeholders by producing runnable software artifacts in short time intervals to gather feedback, identify arising problems as early as possible, and to ensure that customer requirements are properly addressed by the created application. Cloud environments are well suited for such methods since application developers can easily adjust the infrastructure necessary to execute an application without large upfront expenses, as well as use cloud offerings in their application design to reduce development effort [24].

## 1.1 Problem Statement

Enterprise SBAs are expected to continuously retain successful operation, even when facing unexpected problems and changes in their environment. Applications need to be able to adapt in the face of issues such as unavailable partner services, interrupted network links, or hardware failures in the hosting infrastructure. Problems at runtime must be handled gracefully with as little customer-facing impact as possible. Moreover, SBAs must be designed for evolution, as they are expected to remain in service for extended amounts of time. Customer requirements, business processes, or infrastructure constraints can change over time, necessitating fundamental changes to the application implementation, its architecture, or deployment topology. Application management and design methods need to account for these issues and assist practitioners in creating reliable, fault-tolerant applications that are able to dynamically adapt to changes in their runtime environment, as well as support them in realizing evolutionary changes in the application structure.

Cloud computing environments allow for novel ways of efficient execution and management of complex distributed systems, such as elastic resource provisioning and global distribution of application components. However, it also introduces challenges not previously encountered in traditional application design and development. Cloud applications are typically spread over a large number of virtual machines, requiring an application management infrastructure that is able to cope with complex control logic distributed on multiple machines. Furthermore, as the number of involved resources increases, failures of single components become more and more likely. Applications must be able to tolerate these component failures, for instance by gracefully degrading service quality until additional resources are provisioned to reduce SLA violations [100]. This can be achieved by rerouting requests to other components in the system, only serving high-priority requests, or using an external third-party service to (partially) fulfill consumer requests.

Management of enterprise SBAs deployed in the cloud requires comprehensive monitoring of the current state of the complete system, i.e. all its functional components, the supporting infrastructure, as well as relevant partner services, along with effectors to influence the system

to react to changes. The advent of cloud computing presents an opportunity for establishing and implementing novel best practices for unified management and control of applications at runtime, as well as managing their deployment and evolution over time. This applies to both, consumers as well as providers of cloud offerings. While consumers can establish processes, methods and structures for standardized use within their company to improve internal practices, providers have the possibility of significantly improving the quality of applications for a multitude of customers by offering their expertise in application development and management as part of their products.

In literature and practice, these issues are currently not sufficiently addressed. Current approaches mostly focus on improving single phases of the application development lifecycle, such as architectural design [13, 107, 145, 167] and runtime adaptation [32, 81, 85], but do not consider important inter-phase relationships and their influence on application design and management as part of a comprehensive application control framework spanning the complete application lifecycle required for cloud applications.

## 1.2 Research Questions

The problems identified in Section 1.1 serve as motivation for the research conducted throughout this thesis. Specifically, this work addresses the following research questions.

*Research Question I:*
*How can software evolution and adaptation be explicitly incorporated in cloud application design and management?*

As discussed in Section 1.1, the cloud computing paradigm allows for the incorporation of novel mechanisms in enterprise SBA design. Application design should exploit the dynamic nature of cloud offerings by incorporating well-defined methods to document and implement changes in SBAs at any stage of the application lifecycle, whether they are made at design time, during development, deployment, or runtime. Existing work mostly tackles different stages in isolation, but no approach exists that incorporates documentation and management of evolutionary changes across the complete application lifecycle. Application management infrastructures allow for defining monitoring and adaptation concerns to enable deployed SBAs to react to changes in their environment with as little customer-facing impact as possible. Traditional application management is mostly implementing using centralized controllers that gather relevant monitoring data and effect the underlying application based on specified adaptation rules, but no approach exists that explicitly considers the dynamic, distributed nature of cloud environments, assisting practitioners in effectively designing and managing cloud applications and their management infrastructure.

*Research Question II:*
*How can explicit cloud application design and management be autonomously improved in the face of changes in their environment?*

In addition to explicit management of desired application behavior over time, complex SBSs need to be able to cope with unexpected issues that occur in their execution environment. Ap-

plication management policies for deployed components might become unsuitable for the SBS as a whole when the execution environment changes, or incompatibilities might arise due to the complex interactions between multiple components, possibly from different vendors. Such issues need to be handled with as little operator intervention as possible to prevent, or at least minimize, service disruptions and SLA violations.

## 1.3 Scientific Contributions

The work conducted during the course of this thesis, guided by the research questions posed in Section 1.2, has lead to the following contributions to the state of the art in SBA design and management.

*Contribution I:*
*A lifecycle model for documenting and implementing evolution of SBAs throughout the complete application development lifecycle.*

An essential task for enabling controlled evolution of complex SBAs is to model application design decisions, their impact on software architecture, relationships between identified components, and behavior goals governing their functional operation. We present an evolution lifecycle model to support these tasks and enable structured documentation and partial automation of application evolution decisions and procedures. Details are presented in Chapter 3. Contribution I was originally presented in [76].

*Contribution II:*
*An approach for model-based adaptation of cloud applications, both provider-managed as well as using a mediation layer.*

Modeled application behavior goals are well suited for establishing best practices in application adaptation. We introduce a method for provider-managed adaptation in cloud computing infrastructures, enabling practitioners to leverage provider experience managing complex distributed systems without large initial investments. To ease migration to provider-managed adaptation infrastructures, we also present a concept for a provider-independent meta-cloud middleware that prevents vendor lock-in and acts as intermediary until providers support managed adaptation. Details are presented in Chapter 4. Contribution II was originally presented in [78, 141].

*Contribution III:*
*A language and method for specification and optimized deployment of distributed application monitoring and adaptation infrastructures.*

To effectively model and deploy runtime management infrastructure realized in a framework as proposed in Contribution II, we introduce a Domain-Specific Language (DSL) to represent

application structure, monitoring queries, and adaptation rules, along with a distributed management runtime to execute the modeled control infrastructure. Furthermore, we present an algorithm for optimal deployment of monitoring and adaptation operators aimed at minimizing unnecessary network traffic while keeping cost overhead for management infrastructure low. Details are presented in Chapter 5. Contribution III was originally presented in [74, 79].

*Contribution IV:*
*A method for autonomically improving application management policies based on log data analysis without explicit domain knowledge.*

Complex SBSs are comprised of multiple components, each with their own partial control logic. While these control policies might very well be suitable for individual components or certain sets of components, the complex interactions in real-world systems can cause issues in combined application management policies. To alleviate adverse effects of unintended interactions between component management policies, we present a novel approach to incrementally improve runtime adaptation policies without explicit domain knowledge by using machine learning techniques to analyze and act based on application log data. Details are presented in Chapter 6. Contribution IV was originally presented in [75].

*Contribution V:*
*An approach for identifying incompatibilities between implementations of collaborating services using pooled decision trees.*

While one of the claims of the SOA paradigm is simplified interoperability by employing services with well-defined interfaces, problems still arise in practice. Different implementations of the same interface, while syntactically identical, can significantly differ in terms of semantics, due to issues like different interpretations of the service's intended functionality, or bugs in the implementation. While each service implementation might work well in isolation, complex composition and interaction structures in enterprise SBSs can trigger these issues and produce faults that are hard to identify and debug. We present a method for automatically detecting such incompatible service implementations to aid in the discovery and prevention of faults leading to reduced service quality or SLA violations. Details are presented in Chapter 7. Contribution V was originally presented in [72, 73, 77].

## 1.4 Organization of this Thesis

The remainder of this thesis is structured as follows. *Chapter 2* provides background information on basic concepts used throughout thesis. Specifically, the topics of cloud computing (Section 2.1), software evolution (Section 2.2), and autonomic computing (Section 2.3) are introduced. The main matter of this thesis, contributions I-V, is presented in chapters 3 to 7. *Chapter 3* discusses the application evolution lifecycle model that allows for structured documentation and realization of evolution decisions throughout the software development lifecycle. In *Chapter 4*,

we make a case for managed adaptation of cloud applications to reduce required implementation and management efforts while encouraging best practices, along with the meta cloud approach for transitional, provider-independent management of application deployment and runtime management. *Chapter 5* introduces a language and tool set for specification and deployment of distributed application management infrastructures, along with an approach for optimized placement of monitoring and adaptation operators within available infrastructure. To assist application management at runtime, we present an approach for non-intrusive policy improvement based on log data analysis in *Chapter 6*. In *Chapter 7*, we introduce our fault localization method based on pooled decision trees that allows to identify service implementation incompatibilities to prevent faults in complex SBSs. Finally, *Chapter 8* provides conclusions, discusses the presented contributions in light of the posed research questions, and offers an outlook for ongoing and future research.

CHAPTER 2

# Background

In this chapter, we introduce several basic concepts that are used in the remainder of this thesis. First, we illustrate the fundamental properties of the cloud computing paradigm, followed by an introduction of the notion of software evolution, as these topics represent the context and motivation of the work conducted as part of this thesis. Then, we cover the basics of autonomic computing and machine learning that form the foundation for realizing adaptive and evolvable SBSs.

## 2.1 Cloud Computing

Cloud computing [7, 8, 25, 29, 50, 165] is a paradigm that emerged in recent years "for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [109].

The foundational properties of cloud offerings as compared to traditionally provisioned data center infrastructure are (1) *on-demand provisioning*, allowing customers to quickly and easily request an arbitrary, apparently unbounded, number resources, (2) *pay-as-you-go pricing*, charging customers only for actual resources consumed, without requiring long-term contracts or upfront investments, made possible by (3) *economies of scale* through consolidation of computing resources in large data centers, increasing utilization and thereby reducing unnecessary energy and maintenance overhead. These properties enable the realization of elastic applications that dynamically adjust the amount and type resources required to perform business tasks to current demand. As the utility-driven nature of cloud computing [29] is inherently service-oriented, cloud offerings are generally based on SOA principles and offered as services that can be consumed without mandatory user interaction.

Services offered by providers vary in their granularity from low-level infrastructure offerings, such as Virtual Machines (VMs), storage, networking, and load balancing, over partially managed platforms shifting some of the application management process to the provider, to fully-managed

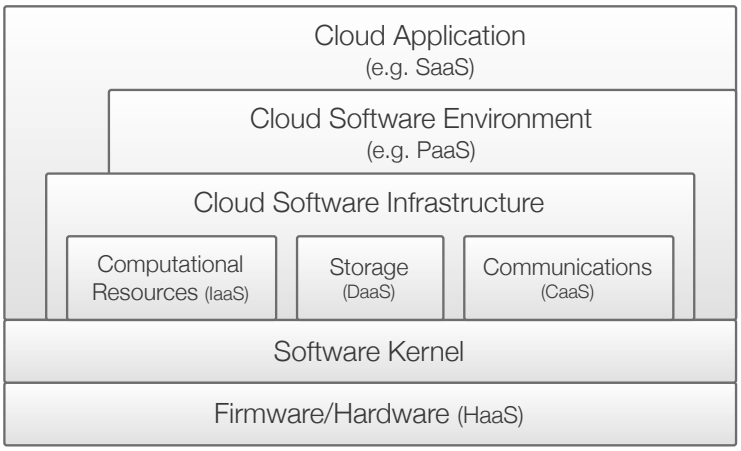



Figure 2.1: Layers of the Cloud Computing Stack (from [162])

services, that are designed to be consumed by end users. These different levels are often categorized into Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [25, 104, 162], as illustrated in Figure 2.1. The three categories of cloud services can (mostly) be mapped to the distribution of management tasks between provider and consumer, as discussed in the following.

Low-level infrastructure products, or IaaS offerings, allow for the provisioning of VMs that are subsequently under complete control of the cloud user. Operating system maintenance, application installation and management, as well as security updates are managed by the cloud user, while provisioning of physical hardware, network links, power, and data center are managed by the cloud provider. The IaaS model offers a high degree of flexibility to customers (as they basically manage their own "virtual data center" in the cloud), but – unsurprisingly – requires them to have the necessary expertise to actually manage the provisioned resources, as mentioned above. Notable examples of IaaS offerings include Amazon Elastic Compute Cloud (EC2)[1], Google Compute Engine[2], VMware vCloud[3], Rackspace Cloud Servers[4], DigitalOcean[5], and Microsoft Windows Azure Compute[6]. In addition to IaaS products for deploying VMs, several providers also offer infrastructure-level services for storing and retrieving data, i.e., Data as a Service (DaaS), such as block storage (e.g., Rackspace Cloud Block Storage[7]) object stores (e.g., Amazon Simple Storage Service (S3)[8]), in-memory caches (e.g., Amazon ElastiCache[9]), Relational

[1] http://aws.amazon.com/ec2/
[2] http://developers.google.com/compute/
[3] http://vcloud.vmware.com/
[4] http://www.rackspace.com/cloud/servers/
[5] http://digitalocean.com/
[6] http://www.windowsazure.com/en-us/services/virtual-machines/
[7] http://www.rackspace.com/cloud/block-storage/
[8] http://aws.amazon.com/s3/
[9] http://aws.amazon.com/elasticache/

Database Management Systems (RDBMSs) (e.g., Rackspace Cloud Databases[10], Amazon Relational Database Service (RDS)[11], Heroku Postgres[12]), and NoSQL databases (e.g., Cloudant[13]).

To reduce the administration overhead of managing virtual infrastructure, PaaS products were created that provide pre-configured software environments for cloud users to simplify deployment, management, and scaling of their applications [92]. When using PaaS products, cloud users sacrifice some of the flexibility of IaaS for the convenience and simplicity of relying on provider-managed infrastructure. Cloud providers offer PaaS products with a variety of different levels of managed behavior. On the most basic level, PaaS products assist application developers by offering a set of pre-configured components that can be assembled and then deployed into separate (mostly unmanaged) containers to ease development setup (e.g., Red Hat OpenShift[14]). Somewhat more advanced, some products offer managed VM containers, with operating system and software development stack maintenance tasks handled by the provider, while still offering developers certain freedom to customize and modify application execution environments (e.g., Heroku[15], and Amazon Elastic BeanStalk[16]). Finally, there also exist solutions that, in addition to fully managing the underlying VMs, performing operating system maintenance and security updates, only provide a restricted (often domain-specific) software environment for cloud users to create their applications in (e.g., Google App Engine[17], Salesforce1[18]). PaaS offerings are aimed at increasing developer productivity and reducing software costs [92] for customers, while providers can leverage information about deployed applications, which would not be available in IaaS-based applications, to improve management policies, optimize deployment topologies, and consolidate infrastructure.

In contrast to the previously discussed infrastructure and platform layers, SaaS, the topmost layer in Figure 2.1, denotes offerings that are targeted at end users as opposed to application developers and operators. SaaS products are usually created to meet certain business needs in a special domain, and are often designed for human interaction via web or native Graphical User Interfaces (GUIs), Notable examples of SaaS offerings include storage solutions, such as Dropbox[19] and Google Drive[20], business productivity tools, such as Google Docs[21] and Basecamp[22], streaming services, such as Pandora[23] or Netflix[24], and various others.

The cloud computing paradigm offers new possibilities for managing and deploying SBSs, such as dynamic resource provisioning and distributed deployment of application components,

[10] http://www.rackspace.com/cloud/databases/
[11] http://aws.amazon.com/rds/
[12] https://www.heroku.com/postgres/
[13] http://www.cloudant.com/
[14] http://www.openshift.com/
[15] http://www.heroku.com/
[16] http://aws.amazon.com/elasticbeanstalk/
[17] http://developers.google.com/appengine/
[18] http://force.com/
[19] http://www.dropbox.com/
[20] https://drive.google.com/
[21] http://docs.google.com/
[22] http://www.basecamp.com/
[23] http://www.pandora.com/
[24] http://www.netflix.com/

but also poses new challenges, since application are executed on “foreign” infrastructure, need to properly scale according to changes in demand while considering infrastructure and management costs. These possibilities and challenges are the fundamental motivation for the research presented in this theses. In this work, we address several challenges of effective application management in cloud environments to ease cloud application management and deployment.

## 2.2 Software Evolution

Software evolution, or software maintenance, is an important part of the software development lifecycle, addressing the fact that applications need to be able to evolve over time to adjust to changes in their requirements, as well as their environment [110]. Early software development processes, such as the well-known waterfall model [134], consider software maintenance as a process to be performed after it has been released or deployed in production to correct bugs and make minor adjustments. However, scholars and practitioners soon realized that software evolution should not just be treated as an after-thought but needs to be incorporated in the software production process. applications are expected to be in service for extended periods of time, during which business requirements, application execution environments, or external dependencies might change. The observations formalized in Lehman’s laws of software evolution [93, 94, 95] clearly illustrate that software will need to be modified over time to react to changes in its environment, and that the necessity for adapting software is inevitable and not a result of bad planning or implementation.

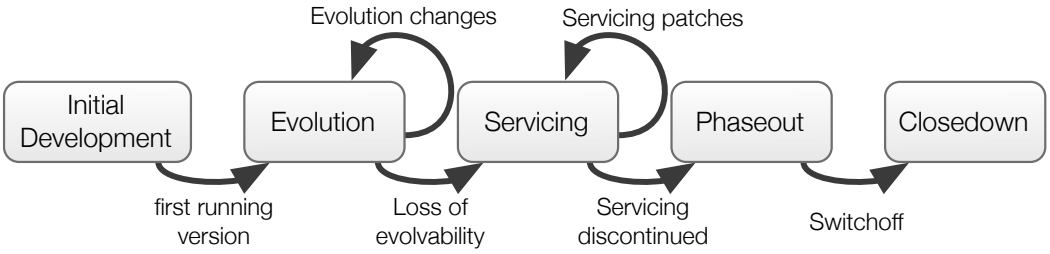



Figure 2.2: Simple Staged Model for the Software Lifecycle (adapted from [21, 131])

One of the early works addressing the necessity of explicitly modeling the evolution stage as integral part of the software lifecycle is the staged model [21, 22, 131], as shown in Figure 2.2. After initial development the application under development enters the *evolution* stage, where any kind of modification to the code can be performed, given that architectural integrity is maintained [110]. When this condition can no longer be satisfied, the software has encountered *loss of evolvability* and enters the *servicing* stage. In this stage, only small changes to the software are performed to keep it running. When it is no longer financially feasible to continue servicing an application it transitions to the *phase out* stage that leads to the eventual *closedown* of the application. A variation of this model is the versioned staged model, as shown in Figure 2.3. This model extends the simple staged model by treating evolution as the backbone of the software

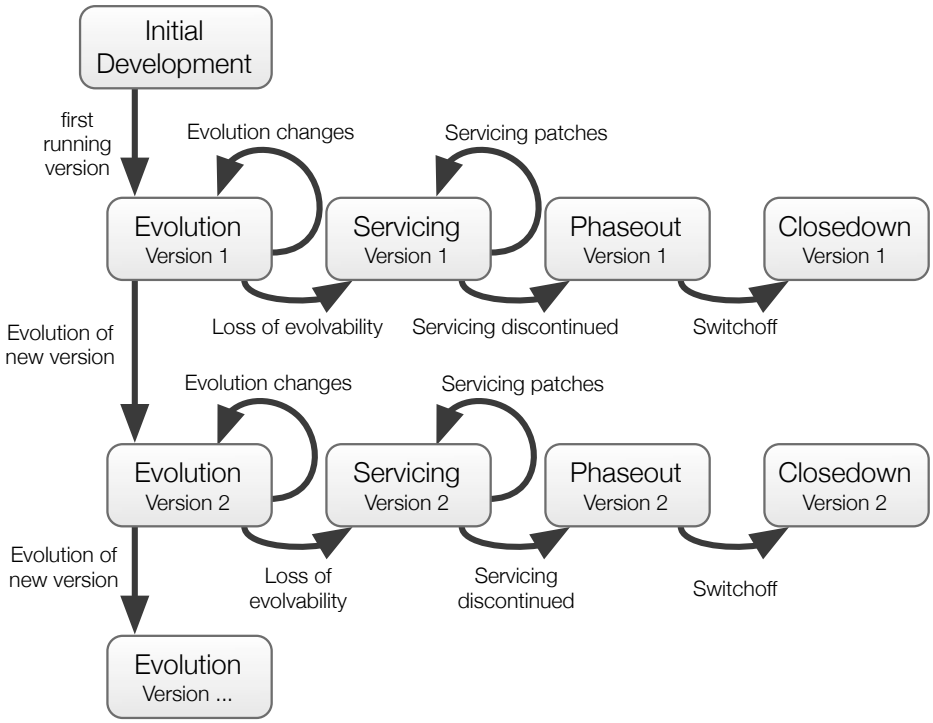



Figure 2.3: Versioned Staged Model for the Software Lifecycle (adapted from [21, 131])

lifecycle [131]. Software versions are produced according to the simple staged model introduced above, but new versions evolve from the codebase at regular intervals to release updated versions to customers. This allows for handling one of the realities of traditional software development, namely that multiple versions of the software will be deployed with customers at any given time.

Building on the principles of evolutionary development processes such as the staged model discussed above, iterative and agile software development methods, such as the Rational Unified Process (RUP) [90], Goal-driven Software Development [143], Feature Driven Development (FDD) [121], Extreme Programming (XP) [19], and Scrum [151], embrace the notion of constant change and acknowledge the need for software to evolve by splitting the software development process into small iterations, aimed at producing tangible results in predictable intervals that can be tested and validated by stakeholders to guide further progress, facilitate early identification of problems, and quickly react to changing requirements [130]. Recent research in software evolution supplements these development models and addresses various important aspects to increase software quality and maintainability, such as predicting bugs by mining bug repositories [168], analyzing software repositories to gather relevant information about application evolution [47], migrating legacy systems to SOA [61], as well as managing evolution of software at the architecture level [13]. In this thesis, we focus on challenges encountered in the development and evolution of SBS and address the need for a well-defined method to handle changes, whether they are made at runtime, deployment, development, or design time in Chapter 3.

## 2.3 Autonomic Computing

The vision of autonomic computing is to create "computing systems, that can manage themselves given high-level objectives from administrators" [88]. In day-to-day operations, human intervention should not be required to keep a software system operational and performing according to specifications. Administrators interact with systems using high-level goals, representing target states a system should be in, as opposed to concrete, detailed steps required to reach these states. To enable this vision of self-managed applications, a number of *self-** properties have been identified [65]. The four properties discussed in IBM's original work [63] are: (1) *self-configuration*: systems should automatically configure, install and deploy components according to high-level policies; (2) *self-optimization*: applications constantly strive to improve their own performance; (3) *self-healing*: systems automatically detect, mitigate, or fix problems that occur in its components or runtime environment; (4) *self-protection*: systems are able to automatically defend against malicious attempts to circumvent security measures or deny service to legitimate users.

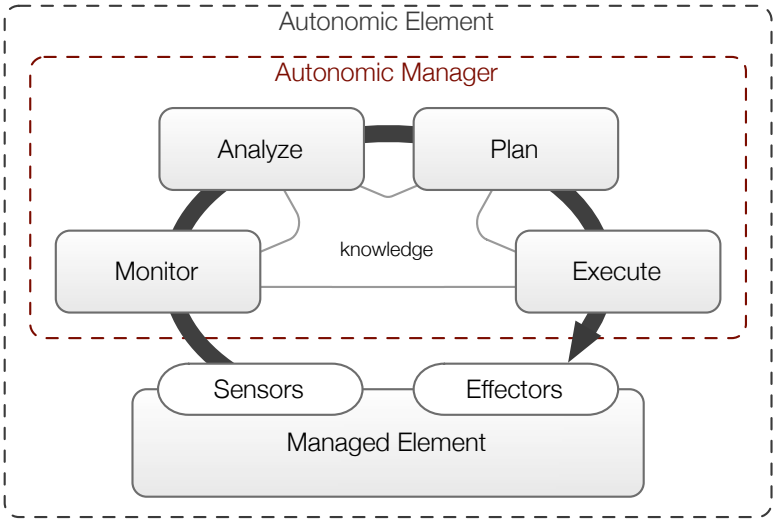



Figure 2.4: MAPE Cycle (adapted from [88])

The usual approach to implement self-* behavior is to model application components as autonomic elements that govern their own behavior. As illustrated in Figure 2.4, an autonomic element consists of a *managed element* supervised by an *autonomic manager*. The managed element represents a regular application component that will be controlled by the autonomic manager using data gathered from sensors and actions executed using effectors. The autonomic manager monitors and controls the managed element using a four-step process forming the well-known Monitor, Analyze, Plan, Execute (MAPE) cycle [65, 88], along with a common knowledge base accessible to all process stages. In the *monitor* step, data is gathered from the managed element through sensors, that are subsequently processed and enriched in the *analyze* stage. Analyzed data is then used to construct a *plan* for how to get the managed element into a more desirable state. Finally, the constructed plan is *executed* using the managed element's

effectors that realize concrete changes in the application logic. In Chapter 5 of this thesis, we build on the concepts discussed above and introduce an autonomic manager for cloud application deployment and management that allows operators to define desired application behavior using a simple DSL.

## 2.4 Reinforcement Learning

Reinforcement learning describes a branch of the wider field of machine learning, itself a part of the area of artificial intelligence research. Reinforcement learning tackles "the problem faced by an agent that must learn behavior through trial-and-error interactions with a dynamic environment" [87]. Figure 2.5 shows how reinforcement learning can be used to realize an agent

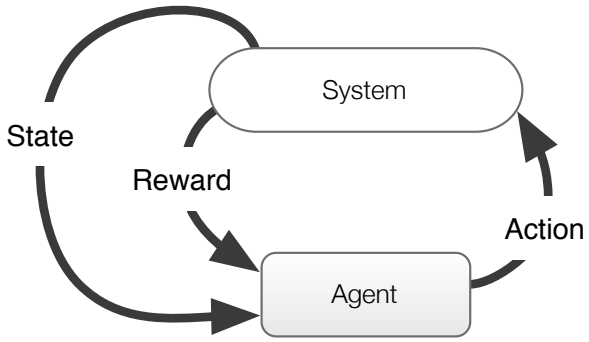



Figure 2.5: Reinforcement Learning Interaction Loop (adapted from [152])

adaptively controlling a system. The system issues information about its current state and provides feedback in the form of rewards, which is used by a reinforcement learning algorithm to improve the agents policy. This policy recommends which action to take based on the system state defining a system controller.

### *2.4.1 Markov Decision Process*

In this section, we provide an introduction to Markov Decision Processes (MDPs) [135], a mathematical framework for decision making, which serves as basis for our adaptive control mechanism. MDPs deal with decision making in an uncertain world, where performance depends on a sequence of decisions. At any point in time a decision making agent chooses among a number of available actions. The execution of an action affects the current state and a reward is given to the decision maker as feedback. How the state is affected probabilistically depends on the current state and the chosen action, but not on previous states. An MDP $(S,A,T,R)$ is formally defined by a set of states $S$, a set of actions $A$, a transition model $T : S \times A \times S \to [0,1]$, and a reward function $R : S \to \mathbb{R}$. The transition function $T(s,a,s')$ determines the probability of reaching state $s' \in S$ when executing action $a \in A$ while being in state $s \in S$. The utility of a state $s$ is defined by the reward function $R(s)$.

The main problem for MDPs is to find a *policy* $\pi$ that specifies which action to take for any state. In particular, $\pi(s)$ gives the action recommended by the policy for state $s$. An optimal policy is one that maximizes the expected rewards over time. Some algorithms, such as value iteration and policy iteration [135], are able to compute an optimal policy if transition model and reward function are known in advance. Reinforcement learning on the other hand solves the online variant of an MDP, where transition model and rewards are initially not known. Additional knowledge gathered by observations of transitions and rewards are used to iteratively improve a policy. Prominent reinforcement learning algorithms include Q-Learning [135] and Dyna-Q [150]. A core contribution of the work presented in Chapter 6 is to show how MDPs, a mathematically sound and well understood model, can be leveraged to manage and improve complex real world control policies of SBSs.

CHAPTER 3

# A Lifecycle Model for the Evolution of Service-Based Applications

The process of engineering and provisioning SBAs follows a complex and dynamic lifecycle with different phases and levels of abstraction. In this chapter, we tackle the problem of making this lifecycle explicit, providing development time and runtime support for evolutionary changes in such systems. SBSs are modeled as integrated ecosystems consisting of four conceptual layers (or phases): design, implementation, deployment, and runtime. Our work is driven by the notion that identifying the right changes (monitoring) and effecting of these changes (adaptation) usually takes place individually on each layer. While considering changes on a single layer (e.g., runtime adaptation) is often sufficient, some cases require systematic escalation to adjacent layers. We present a generic lifecycle model that provides an abstracted view of the problem domain and can be mapped to concrete artifacts on each individual layer. We introduce a real-life scenario taken from the telecommunications domain, which serves as the basis for discussion of the challenges and our solution. Based on the scenario and our experience from a research project on Virtual Service Platforms, we evaluate three concrete use cases which illustrate the diversity of evolutionary changes supported by the approach.

## 3.1 Overview

Current enterprise applications are usually built on the notion of a SOA, i.e., they use and reuse existing infrastructure assets and platform services while themselves providing services to be used by other applications. Such SBSs are typically built for the long haul. Consequently, adapting SBSs to changing environments, or simply improving SBSs to eliminate problems of earlier versions, become central.

While the overall development process of SBSs is by now well-understood, the design of adaptive systems that evolve automatically or semi-automatically along with the environment they live in is still rather uncharted. Specifically, little research work exists that feeds runtime

monitoring data back into the artifacts of previous phases of the development process. There are existing approaches in the field of self-adaptation that enable explicit feedback-loops in order to help software systems adjusting their behaviors according to their perception of the surrounding environment [37, 136]. A number of studies in the area of log mining aim at supporting the extraction of off-line log information for analysis and verification [156]. To the best of our knowledge, none of the existing approaches targets the reflection of runtime data to early development phases in order to support the continuous evolution of software systems. Examples of the reflection are to validate the configuration options selected during system modeling or deployment or to verify the assumptions and rationale of architectural decisions. Without adequate links between runtime monitoring and design-time artifacts, targeted improvement and evolution of SBSs become much more difficult.

In this chapter, we present a novel approach to support a continuous development lifecycle of SBSs. Our approach is a realization of the model-driven development paradigm that extends the traditional development process with feedback loops that can feed runtime information to the corresponding artifacts of the adequate phase. During the course of the development phases, software architects and developers use different models to capture various types of development artifacts, such as architectural design decision, component models, or monitoring rules. Based on these models, deployment configurations, monitoring directives, and adaptation rules can be automatically generated. At the heart of our approach, we introduce a lifecycle evolution model to formally represent the relationships between monitoring information and the development artifacts. The evolution model can be (semi-)automatically achieved with reasonable efforts by extending model-to-model and model-to-code transformation rules. Monitoring information collected from the running code can be fed back into the artifacts of each phase to support on-line or off-line analysis and evolution of all artifacts of the SBS. The evolution model, on the one hand, can help to identify which particular artifacts at which phase may influence a certain unexpected or undesired incident, for instance, performance reducing, policy violation, and so forth. On the other hand, the trace links recorded in the evolution model can significantly enrich the context of the incident for better understanding and analysis. For instance, if monitoring unveils a performance problem at runtime, it is non-trivial to decide if the best way of coping with this situation is to simply reconfigure the system, or to improve system design or even architecture.

## 3.2 Scenario

In this section, we present a motivating scenario from the telecommunications services domain. The enhanced Telecom Operations Map (eTOM) [154], which forms part of the Frameworx[1] program, is a widely adopted industry standard for implementation of business processes promoted by the TM Forum (TMF). Our scenario is condensed from the TMF's Case Study Handbook [153] as well as two eTOM-related IBM publications on practical application of SOA in such systems [52, 58]. Figure 3.1 depicts the service delivery process in Business Process Modeling Notation (BPMN). It consists of six activities $i_1, \ldots, i_6$ (referred to as *interfaces* or *abstract services*). Each

[1] http://www.tmforum.org/TMForumFrameworx/1911/home.html

abstract service activity has alternative sub-activities which we denote as *concrete service implementations* (denoted $c_1, \ldots, c_{12}$ in the figure). At runtime the process selects and executes one concrete service for each service interface.

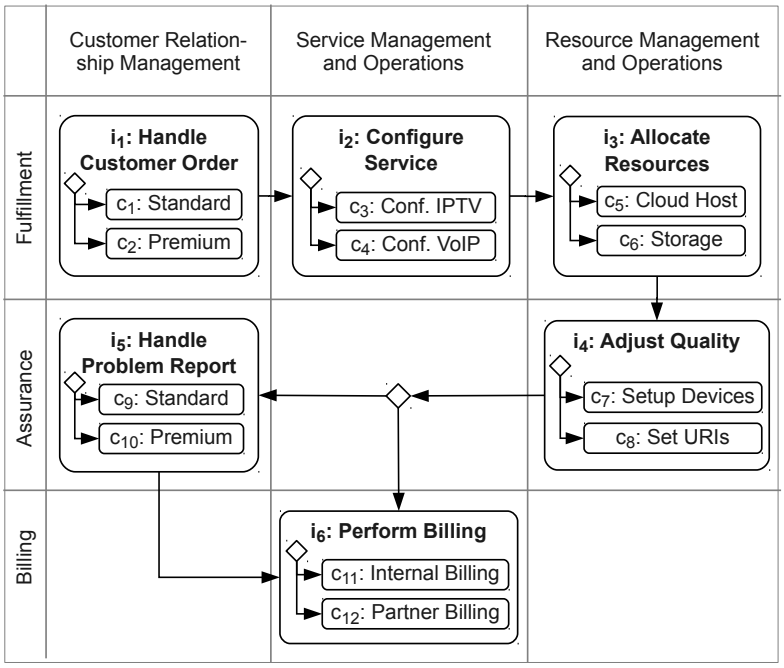



Figure 3.1: Service-Based Scenario Application

The process is initiated by the abstract service $i_1$ (Handle Customer Order) which is offered in two variants for standard and premium users. Depending on the order input, the process then configures a particular service (IPTV or VoIP). The third abstract service allocates the resources required for delivering the service (e.g., a cloud host or storage). Telecommunication services are typically associated with Quality of Service (QoS) attributes, which are fine-tuned by abstract service $i_4$. For instance, this activity configures parameters in the VoIP device or sets the location URI (Uniform Resource Identifier) of IPTV endpoints, in correspondence with QoS requirements. If a problem is detected at runtime, the optional reporting service is executed in activity $i_5$. Finally, the process terminates after storing billing information, either for paying partner providers or for internal accounting if the service was delivered in-house. Besides regular termination, the process may also be interrupted by exceptions at any stage of execution (not depicted in Figure 3.1). We assume that the information whether the execution has terminated regularly or exceptionally is available for each instance of the process.

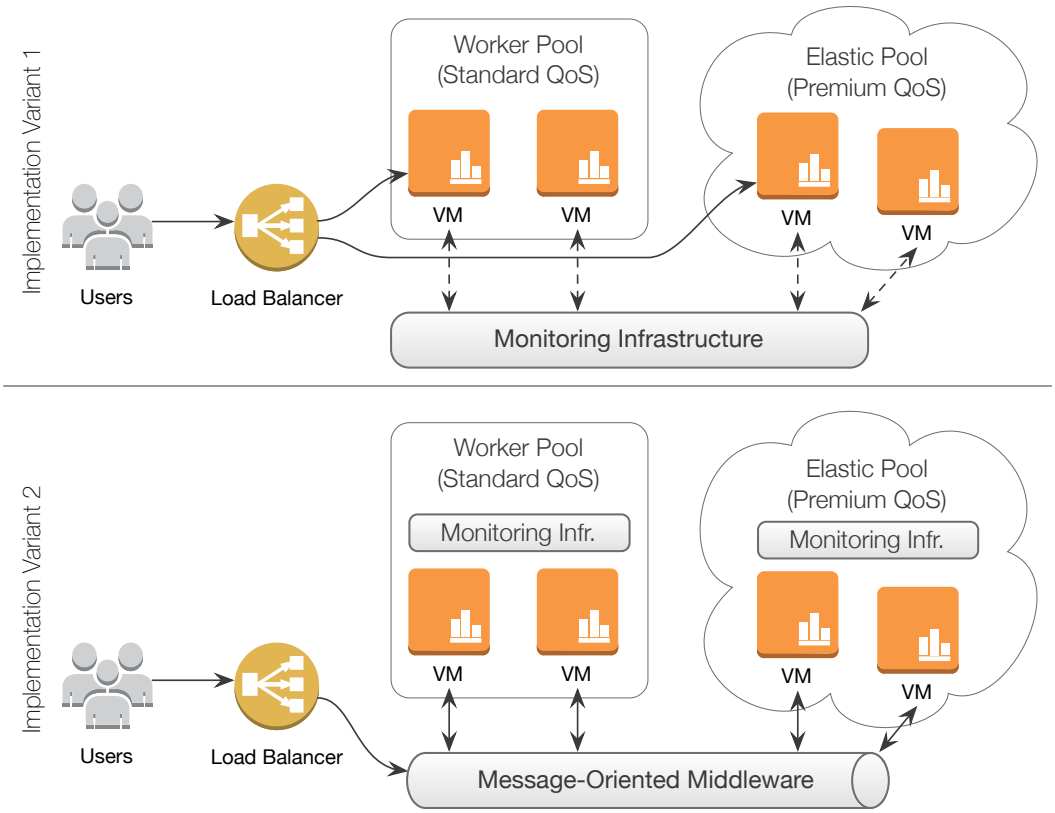



Figure 3.2: Scenario Implementation Variants for a Single Provider

### *3.2.1 Service Variants and Evolution*

When the abstract business process in Figure 3.1 is mapped to a concrete infrastructure, several implementation variants are possible. Two variants are illustrated in Figure 3.2. In variant 1, the clients access a load balancer, which forwards requests to selected VMs. A fixed pool of worker machines supports the standard QoS, and additional VMs are requested from an elastic pool to serve premium clients. A centralized monitoring infrastructure collects performance metrics from VMs in both pools. As the system evolves, we anticipate that the load balancer becomes a bottleneck and hence gets replaced in variant 2 by a Message-Oriented Middleware (MOM) to achieve stronger decoupling. Moreover, the infrastructure becomes decentralized to achieve better locality of the monitoring components within the VM pools. Providing support for implementation variants reflects architectural decisions made at design time or runtime based on given requirements and desired goals.

One defining characteristic of eTOM and the presented scenario is process decomposition, which means that business processes are modeled at different levels of abstraction, from the high-level business goals view down to the technical implementation level. In our scenario this is illustrated by the distinction between abstract and concrete services, though in fact the number of abstraction levels can be higher than two.

### 3.2.2 Challenges for Service Delivery and Evolution

The scenario outlined in Section 3.2 entails the following challenges that are typically encountered when engineering service-based applications:

**Architectural Decisions.** The process of developing service platforms follows recurring architectural design decisions [106], which should be explicit, systematic, and reusable.

**View-Based Modeling.** The platform models need to capture the architectural components and processes, thereby distinguishing multiple external (e.g., service interfaces) and internal (e.g., monitoring infrastructure) views for different stakeholders [155].

**Cross-Provider Platform Integration.** The scenario integrates service platforms from different providers, hence requiring well-defined communication interfaces as well as shared application models.

**Platform Monitoring.** The service platform is subject to fluctuations in request load, hence the service delivery process is governed by monitoring of QoS metrics [112].

**Lifecycle and Adaptation.** Based on the monitoring information and changes in the environment, the platform needs to support short-term adaptation [78] (e.g., scale-out due to load bursts) and long-term evolution (e.g., architectural reconfiguration).

Interface standardization per se does not guarantee compatibility of services originating from different partners. The interactions among services contain complex dependencies and data flows. The number of variations, i.e., possible instantiations of the process, grows exponentially with the combination of concrete services as well as the provided user input. Hence, comprehensive upfront verification and validation in terms of integration testing is not always feasible and can only cover a certain percentage of the possible instantiations. Therefore, in addition to rigorous testing methods, reliable operation of business-critical SBSs requires proactive monitoring to analyze and avoid incompatible configurations at runtime. We present an approach to detect such service incompatibilities in Chapter 7.

## 3.3 Evolution Lifecycle Model

In this section we present a novel application evolution lifecycle model to allow for runtime adaptation of service-based applications, as well as their controlled evolution over time in a unified manner.

### 3.3.1 Application Lifecycle Overview

First, we discuss the application development lifecycle phases usually implemented by iterative and agile development processes, such as the RUP [90], Goal-driven Software Development [143], and Scrum [151]. These approaches facilitate the necessary flexibility required for implementing complex SBSs. Complementary to these approaches we suggest to adapt existing software models and artifacts based on the feedback from monitoring and adaptation rules at runtime. Figure 3.3

summarizes the application development phases at design time and runtime into *Architectural Design*, *Modeling/Implementation*, *Deployment/Configuration*, and *Monitoring/Adaptation*. Before getting into the details of the feedback propagation between and across the different phases during software evolution, we briefly discuss each phase in the lifecycle process.

An emerging practice in architectural design is to not only document solution structures, but explicitly record architectural decisions that led to these structures [82]. Recurring architectural decisions can be documented in architectural decision models, thus increasing reuse and minimizing documentation effort [167].

In practice, in order to describe software architectures various architectural views for different stakeholders' needs are used [16]. Model-driven techniques allow us to transform actual architectural decisions into architectural views automatically in a reusable manner [107]. In the modeling phase, high level views—such as the component-and-connector view—can be refined and enriched to generate lower level technology and platform-specific views using techniques such as the View-based Modeling Framework (VbMF) [155]. The view models are used to generate code skeletons, configuration and implementation artifacts for common boilerplate constructs. The generated code is then augmented with hand-written code by developers to complete the application implementation.

After development leads to an application release, a deployment plan is derived, according deployment descriptors are created, and configuration files are prepared. Deployment descriptors contain information about where to physically deploy the application, which dependencies need to be satisfied for a successful deployment, and how to actually instantiate the application code.

At runtime, the SBS is controlled using previously defined monitoring queries and adaptation rules that allow the application to react to changes in the environment in order to maintain desired behavior. In our approach we extend the notion of adaptation not only to the deployed application and its configuration, but also to the view models and architectural decisions from the earlier phases of software development.

The lifecycle allows for iterations across phases to address necessary changes in architecture, modeling, implementation and runtime management according to the used development method. In the following, we introduce an evolution lifecycle meta model that augments the artifacts produced during each lifecycle phase to assist application evolution.

### *3.3.2 Lifecycle Model Overview*

The lifecycle model is designed to assist application adaptation decisions using relevant data from artifacts in each phase to enable reasoning over costs and benefits of possible application changes. This model contains the required links between the artifacts of the aforementioned phases for propagating the feedback and enabling the appropriate adaptation. Results from monitoring are used to automatically perform adaptations. If automatic adaptation is not possible, the model provides the system operators with recommendations about possible escalation strategies to achieve a given goal. The feedback propagation and adaptation actions can apply to different phases of the application lifecycle. For example, configuration files may change or an alternative application version needs to be deployed at runtime. If the given goals are still not achieved this way, a refactoring of the architectural design and a reconsideration of the existing architectural

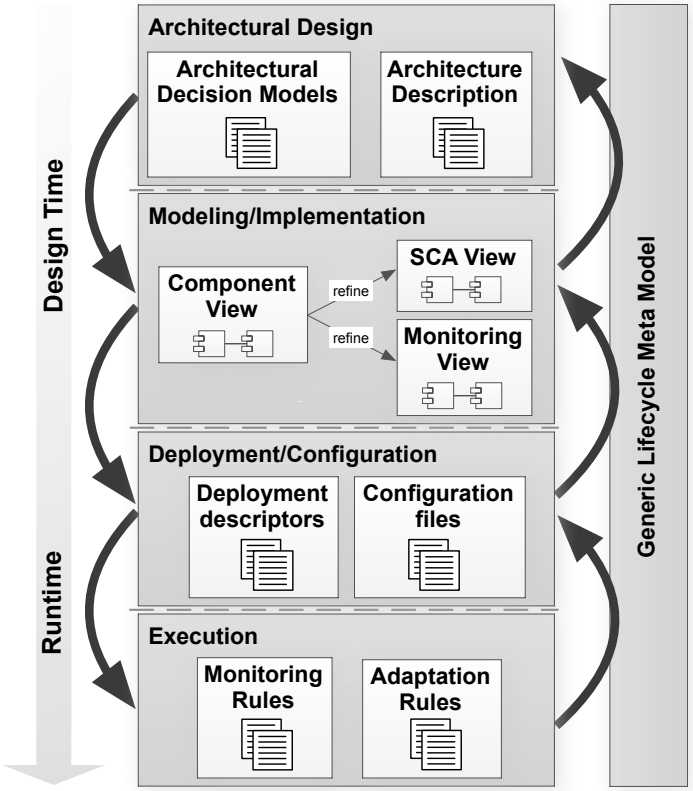



Figure 3.3: Application Lifecycle Overview

decisions will have to take place. In the following, we present the evolution lifecycle model in more detail and discuss relevant properties and instances in each lifecycle phase.

An overview of the proposed model is shown in Figure 3.4. System state information aggregates observed metrics from various application artifacts to represent relevant status information. Application goals are specified along with actions that can be taken to achieve these goals. Artifacts from all lifecycle stages are represented along with their adaptation and monitoring capabilities. Adaptation capabilities represent assets of artifacts that can be modified externally. Similarly, monitoring capabilities provide information about artifact assets. Actions aggregate possible steps necessary to achieve a given application goal, specifying required adaptation capabilities for execution, as well as monitoring capabilities to verify goal fulfillment.

In the following, we discuss the model elements presented in Figure 3.4 in more detail.

**State.** This class represents possible application states, composed of state properties and their values. If (and only if) a State entity accurately reflects the current system state, the *activated* attribute is set to true; The data is gathered from monitoring capabilities of artifacts, and

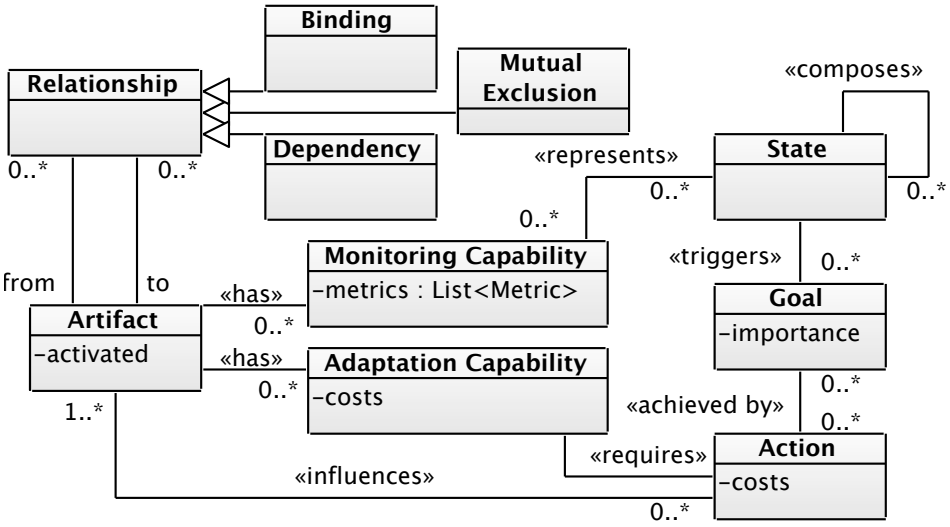



Figure 3.4: Lifecycle Evolution Model

low-level monitoring information is aggregated into higher-level representations suitable for triggering adaptation decisions.

**Goal.** Application goals are derived from requirements and represent system behavior objectives. These objectives are gathered from nonfunctional requirements and represent concerns such as response time or service quality. Application goals modeled in the evolution lifecycle model augment the functional requirements implemented in the traditional development process.

**Action.** Actions encapsulate high-level measures for achieving goals. A goal such as 'minimize response time' can have multiple actions associated with it, e.g., add additional resources, reduce service quality, or change application architecture. For any given action multiple adaptation capabilities offered by application artifacts might be suitable.

**Adaptation capability.** Artifacts in the lifecycle model can have adaptation capabilities associated, representing means of changing them. Adaptation capabilities contain indicators for cost of performing adaptations as well as the supported degree of automation.

**Monitoring capability.** Monitoring capabilities represent relevant properties of artifacts that can be observed and are aggregated in system states to represent high-level application status information.

**Artifact.** All relevant artifacts produced during the application lifecycle are represented in the model, along with indicators representing their value for the application, cost of changing them, as well as their name. These indicators are used to improve adaptation decisions. Artifacts may be related to other artifacts, which allows us to introduce dependencies between the artifacts of the different phases, e.g., between architectural decisions and views, between design views and deployment descriptors, etc.

In the following, we discuss specific artifacts used in different lifecycle stages.

### *3.3.3 Architectural Design*

Architectural decision models and architectural decisions are the basic artifacts produced during the architectural design phase. An architectural decision model contains reusable architectural decisions addressing recurring design issues. An architectural decision contains alternative options which can be realized using design patterns. One or more architectural decisions get reflected on the elements of the architectural design view. The links between architectural decisions and designs allow us to trace back affected decisions from a monitoring rule. These links can be (semi-)automatically established using the mapping techniques presented in [107] for bridging architectural decisions and design models.

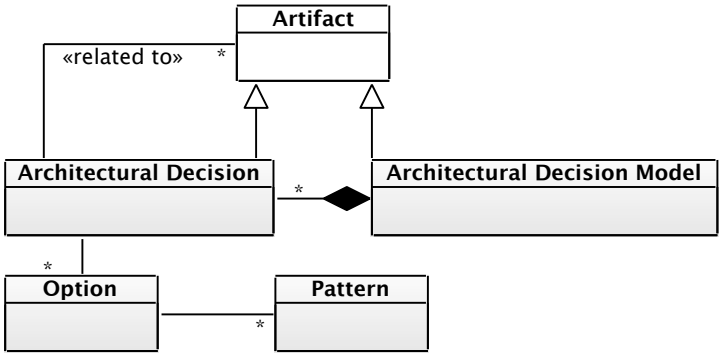



Figure 3.5: Architectural Decision Model

In addition, one or more architectural decisions might have various adaptation capabilities. Imagine, for instance, the case of a selected option of a low-level decision for satisfying low response time. The triggering of an adaptation rule will switch to an alternative option.

### *3.3.4 Modeling and Implementation*

In this stage, the design models are created according to the architecture decision that have been made in the previous stage. The initial creation of design models can be performed automatically using the transformation in [107] based on the architectural decision model and/or manually manipulated by the developers. We show in Figure 3.6 an excerpt of the view-based design models and their relationship with the architectural decisions. View models are composed of view elements representing application subsystems, components, and their interactions. Views are used to capture various perspectives of modeling software systems and help the developers focusing on particular aspects of the system under consideration [155].

The links between view models or view elements and architectural decisions described in Figure 3.6 are based on the mapping techniques mentioned above and actually derived from the association "*related to*" shown in Figure 3.5. Artifacts created in this phase (i.e., views and

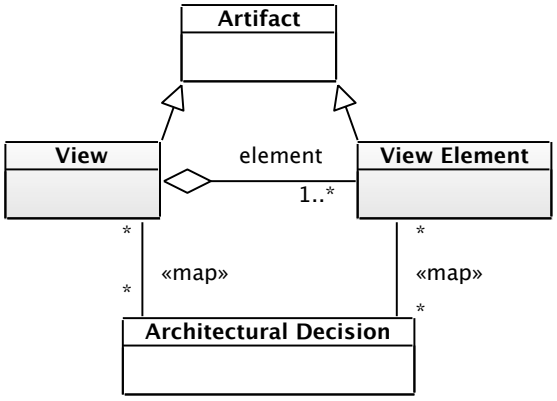



Figure 3.6: View-based Model

view elements) can furthermore expose appropriate adaptation capabilities, specifying the cost of changes as well as the supported degree of automation. Mature applications can benefit from all previously implemented design decisions and model elements by (semi-)automatically reusing components derived in earlier iterations.

View models can be used to generate code artifacts that comprises monitoring capabilities to observe all relevant aspects of the developed application along with adaptation capabilities to modify behavior at runtime. The modeled capabilities are mapped to according actions in the lifecycle evolution model, signifying their influence on the fulfillment of actions leading to desired goals.

### *3.3.5 Deployment and Configuration*

In this stage, the physical deployment structure, as well as the configuration of the application instance to be run are created. As shown in Figure 3.7, code artifacts are bundled in deployable packages, and deployment descriptors are populated with information necessary to instantiate the created application on physical infrastructure.

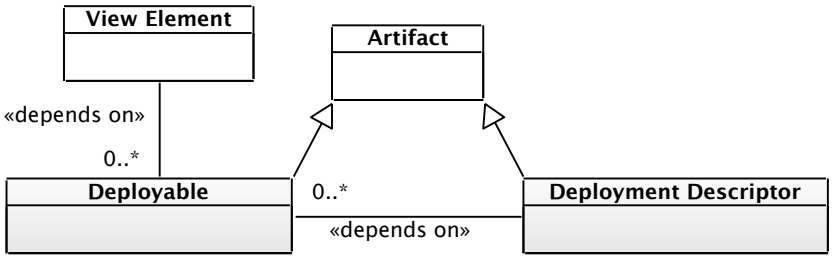



Figure 3.7: Deployment Model

The created artifacts are represented in the lifecycle model. Deployable packages include dependency relations to model elements implemented in the previous phase. Deployment de-

scriptors encapsulate relevant information about component configuration, dependencies, as well as deployment structure. Deployment-specific monitoring capabilities are provided, allowing to observe how the application is instantiated, e.g., how many physical machines are used, and how components are distributed among them. Furthermore, adaptation capabilities allow for the modification of the deployment structure. As mentioned previously, the modeled monitoring and adaptation capabilities are used by actions representing objectives to be achieved by the application and are later used to allow application evolution strategies to consider changes in the deployment structure.

### *3.3.6 Execution and Runtime Monitoring*

After successfully deploying an application, runtime monitoring provides comprehensive data about the fulfillment of specified application goals. Figure 3.8 shows monitoring queries that are executed alongside the SBS are represented in the lifecycle model and map to according monitoring capabilities. Monitoring capabilities represent low-level data, such as response time, number of service calls, and occurred errors, emitted from the SBS at runtime that is assigned to appropriate actions by system designers.

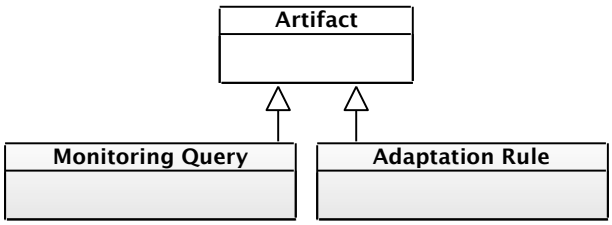



Figure 3.8: Runtime Model

Similarly, adaptation rules adjusting application behavior at runtime are modeled to document links to according adaptation capabilities. As mentioned above, adaptation capabilities represent changes to the application that can be performed at runtime, such as modify service quality, defer background processing, and modify external service binding. These capabilities are used by actions representing higher-level objectives that can be achieved by executing adaptations.

## 3.4 Adaptation and Escalation Strategy

In this section we discuss how the lifecycle evolution model is used to enable adaptation and evolution across the complete development process leading to more efficient, cost-effective and higher quality applications.

Our objective is to perform adaptation actions to achieve application goals in the most cost-effective and automatic way possible. Whenever an application goal violation is detected, an adaptation strategy is derived from the SBS's lifecycle evolution model.

Available adaptation actions are performed automatically at runtime if possible to adjust the SBS's behavior towards the defined goals. Runtime adaptations are usually cheap and can be

performed quickly, allowing for fast reaction to environmental changes. This is similar to how current applications perform runtime adaptation. However, the presented approach is designed to improve on the state of the art by allowing for incorporation of information from all phases of the application development lifecycle to enable more sophisticated, higher quality decisions about SBS evolution considering not only runtime, but also deployment, implementation and design aspects.

When runtime adaptation is not sufficient to reach a defined application behavior goal, modifying the application deployment might be suitable. The deployment model can be modified in several ways, e.g. by replicating application parts, migrating deployable artifacts to different physical machines, or adjusting deployment configuration. After modifying the deployment model, the adaptation strategy will incrementally redeploy the application to minimize downtime. Deployment adaptation can be executed automatically using defined adaptation rules.

The process for selecting adaptation actions to be executed is illustrated in Algorithm 1. The algorithm is periodically executed and supplied with the set of system states $S$ that currently hold, and aims to return a set of actions $A$ that should improve system state towards currently unfulfilled goals $G^u$. We first gather the set of goals that are currently not satisfied, as seen on line 2. Then, the algorithm creates an associative array mapping available actions to a utility value incorporating the action's execution costs as well as the importance of currently unsatisfied goals (lines 6, 11–18). The most valuable action $a_{max}$ is added to the set of actions $A$ to be executed, and goals $G^{ac}_{a_{max}}$ that are achieved by $a_{max}$ are removed from the set of unfulfilled goals $G^u$ (lines 20–23). If no actions are found to achieve a goal $g$, a system-provided escalation action $a^e_g$ is added to the set of actions $A$ to be executed to indicate the need for operator intervention to satisfy goal $g$ (lines 7–10).

If an adaptation triggers an implementation change, adaptation costs as specified in the model are updated according to metrics extracted from the code to improve subsequent adaptation decisions. Metrics, such as changed lines of code, code churn, and time taken, are incorporated to accurately reflect costs of performed adaptation actions.

Similarly, if adaptation decisions lead to changes in view models, according artifacts in the lifecycle model are updated with the cost of the performed adaptation, including implementation changes resulting from model changes.

If architectural decisions are changed, the according cost of change in the lifecycle model includes not only metrics for necessary changes in application architecture, but also view models, as well as related implementation changes.

## 3.5 Related Work

The adaptation of service-oriented systems to rapidly uncertain and changing environment and settings has been studied at various levels in the literature. At the component-level, for instance, FRACTAL [28] relates components with a set of control capabilities to allow adaptations of component properties, addition or deletion of components, etc. for supporting dynamic configuration of distributed systems. At the requirements level, Peng et al. [124] address the self-configuration

**Algorithm 1** Action selection

**Input:** $S$: currently active states
**Output:** $A$: actions to execute

$G^u$ currently unfulfilled goals
$A_g^{ac}$ actions achieving goal $g$
$G_a^{ac}$ goals achieved by action $a$
$a_g^e$ escalation action for goal $g$

$A \leftarrow \emptyset$
$G^u \leftarrow \{g' \in G | \text{«}triggers\text{»}(s, g'), s \in S\}$
**while** $G^u \neq \emptyset$ **do**
    $v_{actions} \leftarrow empty\ dictionary$
    **for all** $g \in G^u$ **do**
        $A_g^{ac} \leftarrow \{a' \in A | \text{«}achieved\_by\text{»}(g, a')\}$
        **if** $A_g^{ac} = \emptyset$ **then**
            $A \leftarrow A \cup a_g^e$
            $G^u \leftarrow G^u \setminus g$
        **end if**
        **for all** $a \in A_g^{ac}$ **do**
            $i_a \leftarrow 0$
            $G_a^{ac} \leftarrow \{g' \in G | \text{«}achieved\_by\text{»}(g', a)\}$
            **for all** $g_a \in G_a^{ac}$ **do**
                $i_a \leftarrow i_a + g_a.importance$
            **end for**
            $v_{actions}[a] \leftarrow i_a / a.costs$
        **end for**
    **end for**
    $a_{max} \leftarrow \arg\max_{a \in A}(v_{actions}[a])$
    $G_{a_{max}}^{ac} \leftarrow \{g' \in G | \text{«}achieved\_by\text{»}(g', a_{max})\}$
    $G^u \leftarrow G^u \setminus G_{a_{max}}^{ac}$
    $A \leftarrow A \cup a_{max}$
**end while**
**return** $A$

of software systems by introducing a formal reasoning procedure at runtime for supporting dynamic quality trade-off among alternative OR-decomposed goals.

However, existing approaches for self-adaptation of requirements goals and architectural models focus on the adaptation only at one layer. However, in our approach, we explore the possibility of performing adaptations at different layers/phases (Architectural Design, Modeling/Implementation, Deployment/Configuration) for satisfying the same goal. We achieve this by introducing traceability links between artifacts of the different layers and by relating the artifacts to monitoring and adaptation capabilities.

Reconfiguration of software systems at runtime for achieving specific goals has been studied in many contexts with focus on the area of service-oriented architectures. Rainbow framework [57] uses abstract architectural models for monitoring a system's runtime properties and proposes adaptations that can be directly reused at the running system. Irmert et al. [81] perform adaptations in service-oriented component models. Their approach utilizes Aspect-Oriented Programming (AOP) for transparent and atomic replacement of service implementations at runtime. Samimi et al. [137] describe an infrastructure for self-adaptive (autonomic) communication services that improve QoS using dynamic service instantiation and reconfiguration. Yet, these approaches concentrate on the reconfiguration and redeployment of the implementations and do not consider any adaptations at architectural modeling or design layer.

Similar to our approach, Shen et al. [145] propose a quality-driven adaptation approach at three different layers: requirements, design decisions and runtime architecture. In their work, the adaptation plans affect all three layers/phases in a unified manner, that is, the adaptation of the requirements goal model triggers the corresponding design decision deduction and runtime architecture reconfiguration. However, the adaptation plans in our approach can also be performed independently at one or more layers, thus providing more flexibility and alternatives.

## 3.6 Discussion

To provide a hands-on discussion of the presented approach, we consider three concrete lifecycle use cases related to the scenario in Section 3.2. The first case (C1) is concerned with short-term adaptation of the internal monitoring and adaptation platform of a telecommunications provider. The second case (C2) deals with more coarse-grained evolution of the platform architecture, and the third case (C3) shows how the proposed escalation mechanism is used extend the model to handle previously unforeseen circumstances.

### *3.6.1 Use Case C1: VM Adaptation*

For use case C1, we consider the runtime artifacts in Figure 3.9 for an implementation of variant 1 of the previously introduced scenario application (cf. Figure 3.2). Deployable *vm1* represents a VM in the "Elastic Pool" handling requests for premium customers. For the given case, we assume that *vm1* is currently exhibiting unusually high RAM usage. The worker pool is managed by deployable *pool1*.

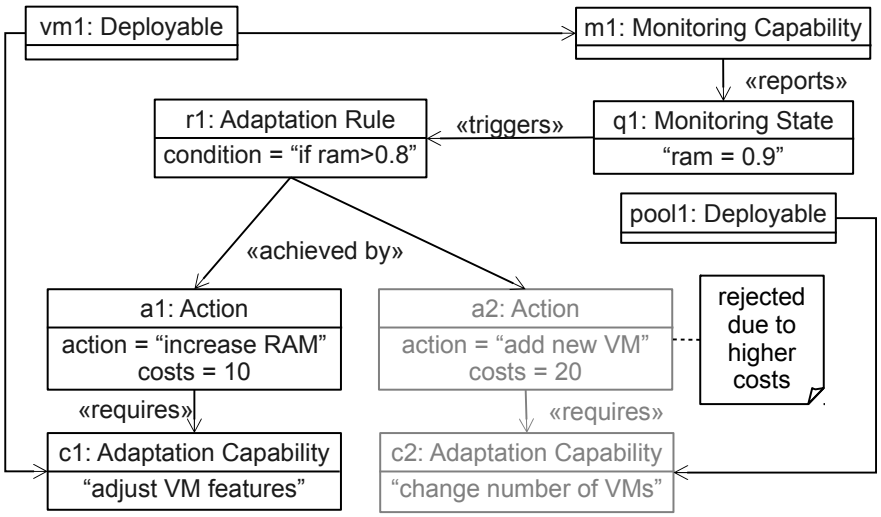



Figure 3.9: Artifacts of Use Case C1

The VM exposes, amongst others, a monitoring capability *m1* reporting general machine health information, such as CPU usage, amount of used RAM, and total RAM. Furthermore, *vm1* offers an adaptation capability *c1* that allows to adjust VM features, such as number of virtual CPUs, available instance storage, and total RAM, at runtime. Worker pool *pool1* exposes an adaptation capability *c2* that allows to start and stop worker VMs. Furthermore, Monitoring state *q1* extracts the current RAM usage from *m1* and asserts the ‘current RAM usage too high’ status. This state triggers adaptation goal *r1*: “RAM usage should not be too high”. Goal *r1* can be achieved by multiple actions, mitigating the encountered problem. In our case, two actions, *a1* and *a2* are available for execution. Action *a1* uses adaptation capability *c1* of *vm1* to increase the amount of RAM available to the worker at runtime, whereas action *a2* uses adaptation capability *c2* of *pool1* to add another worker VM to the pool to spread the work load over more machines. Since *a1* can be performed much quicker than *a2*, we reject *a2* in this case and execute *a1* to mitigate the problem.

The system will subsequently monitor state *q1*, as well as fulfillment of goal *r1* to ensure that the performed adaptation has the desired effect. If the performed adaptation does not lead to the removal of *q1*, and the amount of RAM available to *vm1* cannot be increased further, *c1* becomes inactive, leading to the execution of *a2*.

### *3.6.2 Case C2: Architectural Refinement*

In the second use case, we discuss how our approach can be used during application design to document and improve the development of SBSs. The discussion is based on the scenario introduced in Section 3.2. Consider an enterprise that uses the presented approach for all their software projects. During application design for a new customer, stakeholders face the decision of whether to realize communication between the load balancer worker components using remote

procedure invocation (Variant 1) or messaging (Variant 2), as illustrated in Figure 3.2.

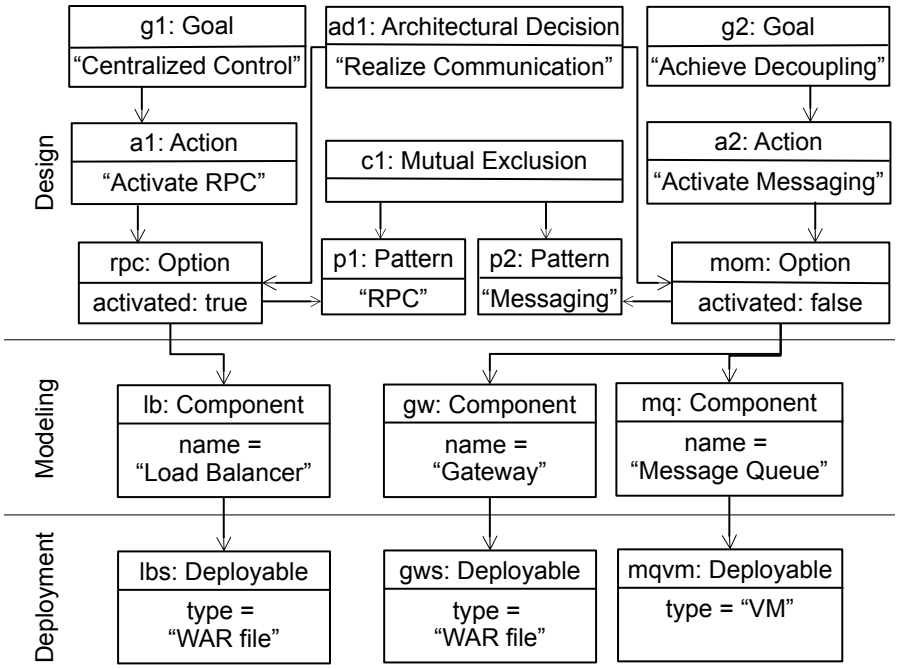



Figure 3.10: Artifacts of Use Case C2

Depending on the application requirements, either variant might be suitable. Remote procedure invocation allows for greater control over communication paths and provides for lower absolute latency. On the other hand, messaging reduces coupling between components and enables horizontal scalability independent of the load balancer component.

The presented approach assists application architects by storing artifacts from previously created applications in an artifact repository that can be queried for past solutions. The repository acts as an enterprise-wide application development knowledge base documenting experiences gathered in past projects. In our case, architects specify goals *g1* and *g2* for the application to be, as shown in Figure 3.10. The repository is queried for the created goals and provides actions for using remote procedure invocation (*a1*) as well as messaging (*a2*), along with according patterns and components to be implemented. Furthermore, a constraint *c1* applies to patterns *rpc* and *mom*, stating mutual exclusion. Since goals *g1* and *g2* are conflicting, application architects provide their decision by setting the *importance* attributes of *g1* and *g2* according to application requirements. If decision leads to the execution of either *a1*, action *a2* becomes inactive due to constraint *c1*. Pattern *rpc*, component *lb*, and deployable *lbs* are merged into the current application model from the repository, allowing operators to adjust their properties to the application at hand.

In the context of product line engineering, the presented approach can furthermore be used to significantly improve knowledge transfer and reuse between product variants. During design of

a new variant, application architects can reuse ‘application slices’ from previously implemented variants. If the load balancer component was realized using patterns *rpc* and *mom* in previous variants, the stakeholders’ decision will lead not only to the inclusion of relevant model elements, but also the accompanying code artifacts.

### *3.6.3 Case C3: Adaptation Escalation*

In the third use case we illustrate the escalation model employed in our approach to enable cross-stage adaptation, as well as incorporating operator intervention. As before, we consider the scenario as described in Section 3.2. We consider the artifacts shown in Figure 3.11. The scenario application is implemented using variant 1 as shown in Figure 3.2. For the given case, we assume that the load among the worker VMs is unevenly distributed due to a bug in the load balancer component. For brevity, application artifacts representing worker VMs, pool manager, load balancer, as well as their monitoring and adaptation capabilities are omitted in the figure.

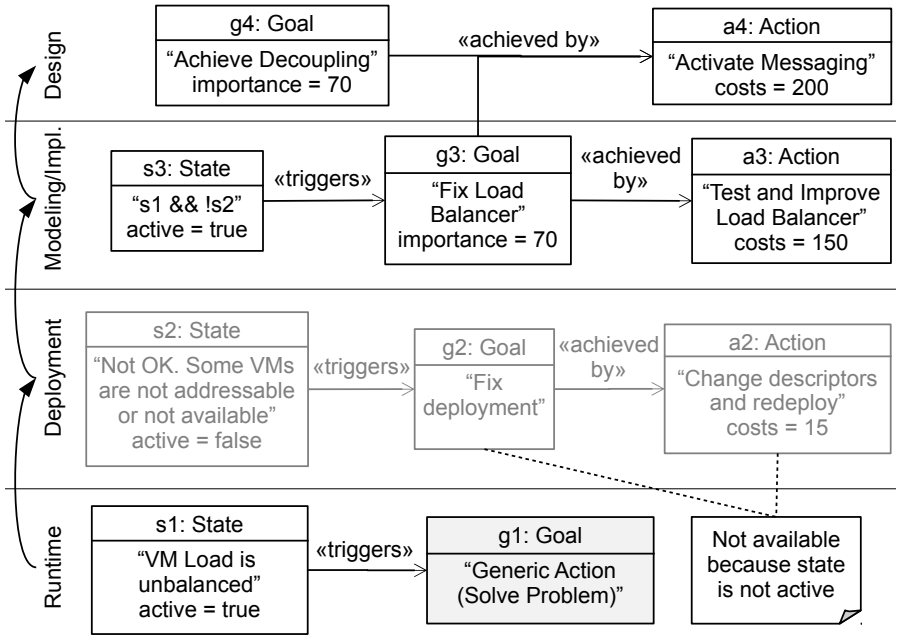



Figure 3.11: Artifacts of Use Case C3

Worker VMs report their CPU load through monitoring capabilities as described in case C1, and uneven load distribution activates state *s1* in the lower part of Figure 3.11. We assume that *s1* was put in place by application designers as a precaution to notice a system state that should not occur according to the specification. Hence, there is no mitigation strategy defined for *s1*, i.e., no adaptation goals are specified to execute adaptation actions. A generic goal *g1* is provided by the system to indicate that this state, while not yet handled, should be fixed if it occurs.

Since *g1* does not have any candidate actions associated that could be triggered automatically, the problem is escalated to system operators. Support staff assess the situation and suspect that the problem cannot be solved using runtime adaptations, but a faulty deployment could have caused the problem to appear. The deployment structure is validated using state *s2* that is active if there is a problem with VM deployment. In our case, deployment is valid and the associated mitigating actions need not be executed. Operators now suspect that a bug in the load balancer component is responsible for the problem.

A composite state *s3* is created to represent the facts gathered during problem analysis. Furthermore, goal *g3* is created to document the desired system state, along with action *a3* representing the maintenance effort to fix the bug in the load balancer component. Additionally, actions to mitigate the problem are queried from the artifact repository. In our case, action *a4* from the repository is found to also solve the problem, albeit at higher cost. In addition to fulfilling goal *g3*, action *a4* furthermore fulfills goal *g4*, which was not satisfied before due to low importance. However, facing the problem at hand, *a4* is now the most feasible action to execute due to the increased benefits of satisfying *g3* as well as *g4*. If a messaging-based implementation of the load balancer component is already available from previous application variants (as discussed in C2), *a4* can be applied with minimal operator intervention. Otherwise, developers are requested to provide the necessary implementation and relevant application components are re-deployed.

## 3.7 Summary

The systematic evolution of service-based systems is currently not well supported, particularly when considering integration of the four conceptual lifecycle layers: design, implementation, deployment, and runtime. Our proposed methodology for addressing this problem is to distill the concepts and artifacts from each layer into a generic lifecycle model, which allows for specific adaptation within a layer and at the same time escalation to adjacent layers. Escalation to a layer of higher abstraction (e.g., change of design decision as in Figure 3.11) is typically followed by downward traversal of the lifecycle phases (e.g., re-generation of code, re-deployment of components, re-initialization of monitoring queries). Technically, the process of adaptation is triggered by monitoring primitives, which can be combined into aggregated information, and are eventually correlated with artifacts from the lifecycle model. The correlation between measurable monitoring metrics and the lifecycle artifacts is the cornerstone for identification of system dependencies and possible adaptation actions.

Modeling the set of goals with associated alternative actions allows to make decisions about the best action to take in specific situations. While the decision for actions on lower layers (deployment, runtime) can be done mostly automatic, more high level decisions (concerning architecture or modeling) are often semi-automatic and require intervention by human experts. To enable the process of automatic selection, we propose using a cost/benefit tradeoff model associated with actions and the respective goals they achieve. Quantifying the benefit (importance) of goals is specific to the application domain, whereas costs of actions can typically be quantified precisely, e.g., man-hours for implementation, or usage fees for elastic Cloud computing resources.

The three case studies, presented to illustrate the feasibility of our approach, highlight runtime adaptation, architecture refinement, as well as adaptation escalation. We argue that the concept

of a generic lifecycle model provides a solid basis for extended aspects related to reliability and enforcement of QoS. In future work we aim to incorporate fault detection techniques to automatically assert previously unknown fault states when they occur, as well as adaptation policy improvement to improve on the modeled rules and augment system control policies. We also envision that the approach can be integrated with automated testing [70] to identify incompatible configurations of activated artifacts.

CHAPTER 4

# Model-Based Adaptation of Cloud Computing Applications

In this chapter we propose a provider-managed, model-based adaptation approach for cloud computing applications, allowing customers to easily specify application behavior goals or adaptation rules. Delegating control over corrective actions to the cloud provider will pose advantages for both, customers and providers. Customers are relieved of effort and expertise requirements necessary to build sophisticated adaptation solutions, while providers can incorporate and analyze data from a multitude of customers to improve adaptation decisions. The envisioned approach will enable increased application performance, as well as cost savings for customers, whereas providers can manage their infrastructure more efficiently.

## 4.1 Overview

The cloud computing paradigm has found widespread adoption throughout the last couple of years. The pay-per-use model of cloud computing proved to be convenient in many respects. It allows to cope with services of time-varying resource demand and peak loads. Cloud computing helps to avoid high initial investments in IT infrastructures, as resources can be dynamically provisioned even if the demand for a service is unknown in advance [8].

However, in order to benefit from the resource elasticity provided by cloud computing, applications need to be properly built. We argue that developers should be able to create models describing their application's adaptation capabilities together with adaptation goals defining their preferences. Cloud computing providers use this information to control and adapt the application according to the customers' objectives. Customers have only a limited view on the execution infrastructure, while cloud computing providers, when given access to necessary information from the application, have a complete view and can incorporate low-level infrastructure details to make adaptation more efficient and effective.

Models are already used for improving deployment of cloud computing applications. An example is Amazon's CloudFormation [5] service, providing a DSL for defining the infrastructure required by an application, which can be provisioned in a single step. Research has brought forth sophisticated approaches for adapting application topologies and resource allocations, e.g., for latency requirements [35], power and cost considerations [86], or deadline-driven workflow scheduling [89], among other purposes. In practice, however, cloud providers currently do not provide mechanisms for further managing an application at runtime. We propose to use models allowing application developers to create "management hooks" for the cloud provider. This has the advantages that all applications can benefit from a sophisticated control mechanisms offered by the provider. Application developers use models to state the objectives for application control, relieving them from implementing complex adaptation schemes. Cloud providers have a deeper understanding of the application infrastructure, which they can use to improve application adaptation, but also to optimize resource usage. Also, providers can leverage the experiences and data from similar applications to improve control and adaptation over time.

## 4.2 Models in Cloud Computing

The main responsibilities of cloud application lifecycle management are infrastructure provisioning, application deployment, and finally control and adaptation at runtime.

Infrastructure provisioning involves setting up virtual machines, installing and configuring required software packages, and initializing cloud services, such as load balancers, database instances, persistent storage, or caches. Manual infrastructure provisioning is cumbersome and error-prone, and limits the level of reusability, e.g., to set up identical environments for development and testing. Furthermore, proper change management of infrastructure is not supported. Model-based approaches for infrastructure provisioning that have recently emerged help to overcome these limitations. Amazon's CloudFormation provides a service to create infrastructure templates used to create a collection of related infrastructure resources and provision them in an automated way. These models can be easily reused and shared using infrastructure template repositories, helping to establish and distribute best practices for different kinds of applications. The provisioned resources, e.g., virtual machines, still need to be properly configured to include all necessary application dependencies, as well as respective links between system components. To enable a predictable and repeatable process for this, deployment models are used to enable automated server configuration according to predefined specifications. Popular approaches for modeling software deployment include Configuration Management (CM) tools like Chef [120], Puppet [127], or cfengine [9]. Resource configuration is modeled using vendor-specific DSLs, specifying required software packages and libraries, as well as necessary configuration files and parameters to support the application to be deployed and all its components. The CM will attempt bring all managed resources into the state defined for its particular role by installing packages, deploying configuration files, and (re-)starting affected services.

Application deployment and redeployment deal with packaging and copying artifacts to according infrastructure resources, configuring them, establishing links between them, and finally making sure they are properly launched. Deployment models are used to ensure efficient and effective application roll-outs and updates based on declarative specifications, e.g. [74]. Again,

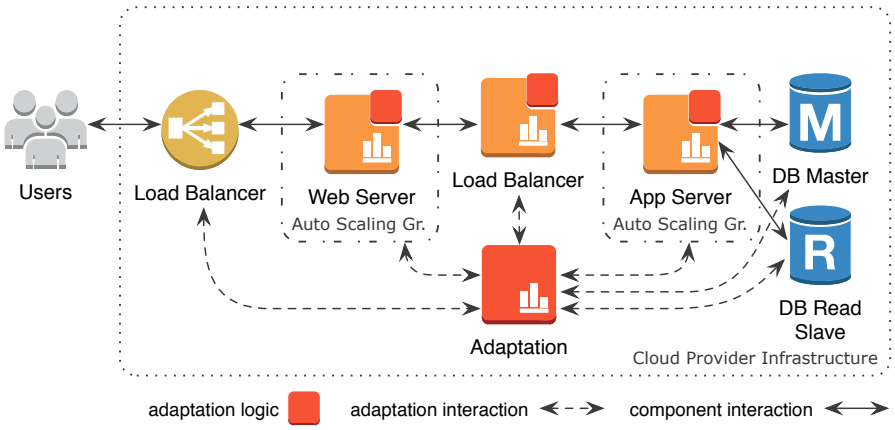



Figure 4.1: Traditional Cloud Application Architecture

CM tools, as mentioned above, can be used to model application components, their dependencies, and required parameter settings.

While deployment models enable the predictable, testable and repeatable provisioning of runtime infrastructure, their responsibility stops at deployment time. The models are not designed to consider application specific runtime aspects or application behavior goals that should be reached. Cloud providers offer means to monitor basic infrastructure or application metrics, but adaptation is mostly limited to starting or stopping instances to cope with varying load patterns. Complex applications, however, have a variety of possibilities to cope with changes to the environment without necessarily resorting to scaling out available resources. Certain non-critical process steps could, for instance, be deferred to a later time if current load is too high, or service quality could be adapted according to environmental circumstances. Today's cloud providers, however, employ a black box model for applications deployed on their infrastructure, leaving customers responsible for realizing their own adaptation infrastructure to implement application-level reactions to changes in the environment. Future cloud service providers will offer means for integrating advanced application adaptation into their offerings using a model-based adaptation approach, as discussed in the next section.

## 4.3 A Case for Model-based Adaptation

In this section, we present an approach for provider-assisted model-based application adaptation that will yield benefits for both providers and customers.

Figure 4.1 shows a traditional cloud application deployment for an exemplary web application. Incoming requests are distributed across several front end web servers using the provider's load balancer service. The web servers access the back end infrastructure using a custom middle tier load balancer, evenly distributing load on the available back end application servers. The application servers execute business logic and access the database cluster. Web and application server instances can be added or removed according to their current and predicted future load. The

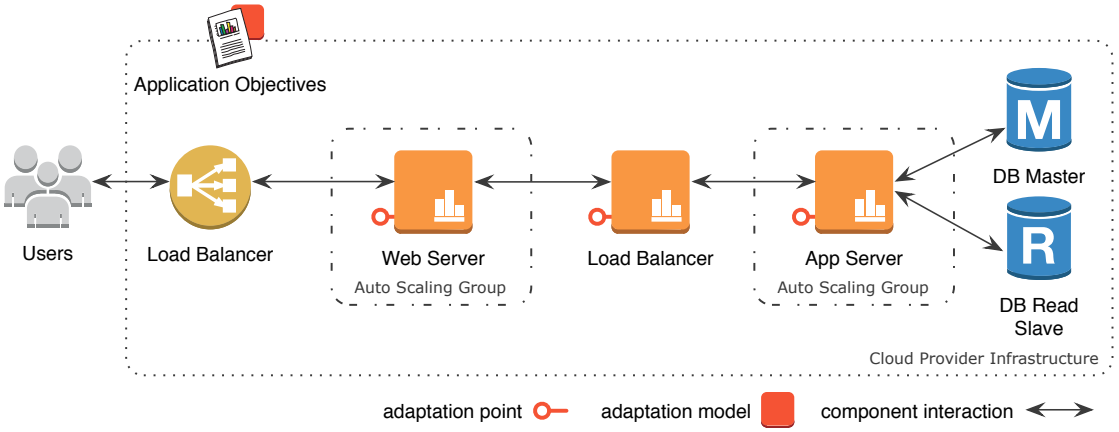



Figure 4.2: Cloud Application Architecture using Provider-Managed Adaptation

custom adaptation component controls service quality by performing corrective actions on all application services according to defined behavior goals, realized as autonomic control loop [65]. It will, for instance, incrementally raise the fragment cache expiry time out up to a certain threshold on the web servers when an increase in request traffic is detected. Similarly, a recommendation module running on the app server instances will set to only return cached user group recommendations, or a list of the most purchased products, instead of personalized recommendations, in response to high order volumes. This will lower stress on the database, as well as the application servers and starting new instances is not immediately required. Analogously, when application load is low, the web servers will always serve fresh content and the recommendation module will always deliver personalized recommendations. Since these adaptation actions influence service quality and thus user experience, the customer will employ a utility function within the adaptation component to determine, when to change service quality and when to scale. Currently, cloud providers only offer support for automatic scaling of application components based on infrastructure and application metrics, but, as deployed applications are considered black boxes, do not provide means to alter component runtime behavior to react to changes in the environment. As indicated in the figure, customers need to deploy their own adaptation infrastructure, often scattered across deployed components.

We propose an approach for model-based cloud application adaptation. Customers specify emitted metrics, available adaptation points and desired objectives using provided models, in order to make applications ready for provider-driven adaptation and control. These models can be realized using the evolution lifecycle model presented in Chapter 3, or utilize previous research such as [164]. This gives customers the ability to declaratively specify desired application behavior, while the provider has necessary information to take over control. Figure 4.2 shows the cloud application discussed above, deployed using the envisioned model-based provider-managed adaptation approach. A dedicated custom adaptation component is not needed anymore, as customers describe desired application behavior objectives in the adaptation model and delegate its execution to the cloud provider. This results in significant cost savings for the customer,

as creating sophisticated and comprehensive adaptation solutions requires considerable effort and expertise. The business logic components do not contain any adaptation logic, but only provide designated adaptation points to allow for external management. Available adaptation points are referenced in the adaptation model for use in corrective actions. As a result, component design is simplified through the clear separation of adaptation capabilities and adaptation logic. As desired application behavior is declaratively specified in the application adaptation model, it can easily be reused or modified to accommodate changes without altering components. In the presented exemplary application, the customer specifies the following adaptation points: `SetFragmentCacheTimeout` to modify the fragment cache timeout in the web server component, and `SetRecommendationStrategy` to switch between personalized recommendations, cached customer group recommendations, or a list of the most sold products overall in the application server. Additionally, the `ScalePool` adaptation point supplied by the cloud provider can be used to start and stop application instances. Monitoring metrics data required to take adaptation decisions are modeled with the application objectives, or a push-based approach similar to current provider-assisted scaling solutions such as the previously Amazon CloudWatch is used. The model allows for specification of conventional Event-Condition-Action (ECA) rules to govern application behavior, so existing applications can easily migrate their current adaptation strategy. Alternatively, the customer can define a simple utility function to reflect application objectives according to desired characteristics, such as 'response time should not exceed 500ms', 'service quality should be as high as possible', and 'infrastructure cost should be as low as possible'. This is possible since adaptation points contain indications on how they will affect application behavior. Adaptation point `SetFragmentCacheTimeout`, for instance, will indicate that higher parameter values should decrease response time, but will also decrease service quality. These indications will initially be supplied by customers, but the cloud provider will monitor the effects of performed corrective actions and adjust adaptation point information during the runtime of the application. The provider-managed adaptation service uses the customer-supplied objectives and infers optimized actionable adaptation strategies. A cloud provider can not only use monitoring data from the application to be controlled, but also consider historical data from different but similar customers, leveraging time series analysis to better anticipate load patterns or even adaptation action effects for common components such as databases.

Allowing for provider-assisted application adaptation at runtime has a number of distinct advantages. Outsourcing some control over adaptation decisions will benefit customers. Providers can offer optimized adaptation decisions based on data from similar customers. Furthermore, providers are able to improve resource provisioning strategies and offer better service to customers.

By handing off control over the execution of adaptation actions, customers can benefit from the provider's expertise in reliability and resilience engineering. They are relieved of implementing their own complex adaptation mechanisms and can use an advanced, thoroughly tested solution offered by the cloud provider without investing large amounts of capital or personnel. Leveraging economies of scale, the cloud provider in turn can offer their adaptation solution to customers with competitive pricing strategies while retaining appropriate return on investment.

Using data from a multitude of customers, the cloud provider can make better adaptation decisions for customers, enabling higher levels of service, as well as cost savings for customers

and the provider. By analyzing time series data of a large number of customers, the provider can anticipate, for instance, future load patterns much more accurately than a single customer could with their data alone. This allows for better adaptation decisions, as the provider can establish categories of customers, enabling more precise load predictions based on signals like traffic patterns and geographical distribution. Using this data, the provider can offer targeted adaptation decisions or recommendations for geographic placement of instances, or even migration of instances to more suitable locations.

Furthermore, the additional information available to the provider will enable highly optimized resource provisioning, keeping over-provisioning at even lower levels than with traditional elastic applications that can only analyze traffic patterns from their own past. Customers naturally benefit from lower resource usage by paying less, while the provider now has access to previously unused but, due to inefficient provisioning, inaccessible resources. Furthermore, the supplied adaptation goals can be used to react in different ways to varying environmental conditions. Depending on the utility, it might be more reasonable for an application to defer certain processing steps to a later time, thus reducing load on application instances, before scaling out by adding new machines. Similarly, when request traffic declines it might be beneficial to process queued tasks on the now underused instances – at least until their current billing interval is over – before terminating them to reduce infrastructure costs. The customer-provided utility functions guide the adaptation decisions considering application performance, service quality, as well as infrastructure costs. While the presented approach is ideally deployed by the cloud provider, a transitional implementation could also be realized as part of a cloud abstraction layer like the meta cloud, as discussed in Section 4.4 below, allowing the use of a managed adaptation solution without explicit cloud provider support.

## 4.4 The Meta Cloud Abstraction Layer

The emergence of more and more cloud offerings from a multitude of service providers calls for a meta cloud, which smoothens the jagged cloud landscape. We discuss our proposal for such a meta cloud, and explain how it solves the lock-in problems that current users of public and hybrid clouds face.

The cloud computing paradigm has found widespread adoption throughout the last years. The reason for the success of cloud computing is the possibility to use services on-demand with a pay as you go pricing model, which proved to be convenient in many respects. Because of low costs and high flexibility, migrating to the cloud is indeed compelling. Despite the obvious advantages of cloud computing, many companies hesitate to “move into the cloud”, mainly because of concerns related to availability of service, data lock-in, and legal uncertainties [8]. Lock-in is particularly problematic for the following reasons. Firstly, even though availability of public clouds is generally high, eventual outages still occur [133]. If this is the case, businesses locked into such a cloud are essentially at a standstill until the cloud is back online. Secondly, public cloud providers generally do not guarantee particular service level agreements [146], i.e., businesses locked into a cloud have no guarantees that this cloud will continue to provide the required QoS tomorrow. Thirdly, the terms of service of most public cloud providers allow the

provider to unilaterally change pricing of their service at any time. Hence, a business locked in a cloud has no mid- or long-term control over their own IT costs.

At the core of all of these problems, we can identify a need for businesses to permanently monitor the cloud they are using, and to be able to rapidly "change horses", i.e., migrate to a different cloud if monitoring discovers problems or estimations foresee issues in the future. However, migration is currently far from trivial. A plethora of cloud providers is flooding the market with a confusing body of services, such as compute services e.g., Amazon Elastic Compute Cloud (EC2) and VMware vCloud, or key-value stores, e.g., Amazon Simple Storage Service (S3). Evidently, some of these services are conceptually comparable to each other, others are vastly different, but all of them are, ultimately, technically incompatible and follow no standards but their own. To further complicate the situation, many companies are not (only) building on public clouds for their cloud computing needs, but combine public offerings with their own private cloud, leading to so-called hybrid cloud setups [165].

In this section we introduce the concept of a meta cloud consisting of a combination of design time and runtime components. Meta cloud abstracts away from technical incompatibilities of existing offerings, thus mitigating vendor lock-in. It helps to find the right set of cloud services for a particular use case, and supports an applications initial deployment and runtime migration.

### *4.4.1 The Current Weather in the (Meta) Cloud*

Firstly, **standardized programming APIs** are required to enable the development of cloud-neutral applications, which are not hardwired to any single provider or cloud service. Cloud provider abstraction libraries such as libcloud [55], fog [18], and jclouds [54] provide unified APIs for accessing cloud products of different vendors. Using these libraries, developers are relieved of technological vendor lock-in, as they can switch cloud providers for their applications with relatively low overhead.

As a second ingredient, the meta cloud makes use of **resource templates** to define concrete features that the application requires from the cloud. For instance, an application needs to be able to specify that it requires a given number of computing resources, internet access, and database storage. Some current tools and initiatives (e.g., Amazon's CloudFormation [5], or the upcoming TOSCA specification [118]) are working towards similar goals, and can be adapted to provide these required features for the meta cloud.

In addition to resource templates, the automated formation and provisioning of cloud applications, as envisioned in the meta cloud, also depends on sophisticated features to actually deploy and install applications automatically. Predictable and controlled **application deployment** is a central issue for cost-effective and efficient deployments in the cloud, and even more so for the meta cloud. Several application provisioning solutions exist today, enabling the declarative specification of deployment artifacts and dependencies to allow for repeatable and managed resource provisioning. Notable examples include Opscode Chef [120], Puppet [127], and juju [31].

At runtime, an important aspect of the meta cloud is **application monitoring**. Monitoring enables meta cloud to decide whether new instance of the application should be provisioned, or whether parts of the application need to be migrated. Currently, various vendors are providing tools for cloud monitoring, ranging from system-level monitoring (e.g., CPU, bandwidth), application-level monitoring (e.g., Amazon's CloudWatch [6]) up to monitoring of service level

agreements (e.g., monitis [114]). However, the meta cloud requires more sophisticated monitoring techniques, and in particular approaches for making automated provisioning decision at runtime based on context and location of current users of the applications.

### *4.4.2 Inside the Meta Cloud*

To some extent, the meta cloud can be realized based on a combination of existing tools and concepts, part of which are presented in the previous section. The main components of the meta cloud, depicted in Figure 4.3, are described in the following and their interplay is illustrated using the previously introduced sports betting portal example. The components of the meta cloud

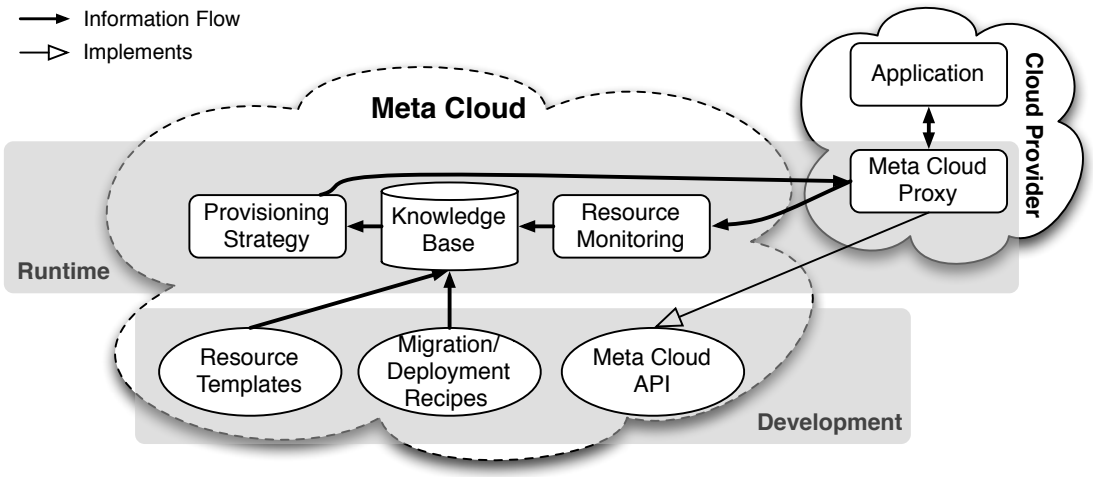



Figure 4.3: Conceptual Overview of the Meta Cloud.

can be distinguished whether they are mainly important for cloud software engineers during development time or whether they perform tasks during runtime.

#### Meta Cloud API

The meta cloud Application Programming Interface (API) provides a unified programming interface to abstract from the differences among provider API implementations. For customers, the use of the meta cloud API prevents their application from being hard-wired to a specific cloud service offering. The meta cloud API can be built upon available cloud provider abstraction APIs, like libcloud [55], fog [18], and jclouds [54], as previously mentioned. While these mostly deal with key-value stores and compute services, in principle all services can be covered that are abstract enough to be offered by more than one provider and whose specific APIs do not differ too much, conceptually.

### Resource Templates

Developers describe the cloud services necessary to run an application using resource templates. They allow to specify service types with additional properties, and a graph model is used to express the interrelation and functional dependencies between services. The meta cloud resource templates are created using a simple DSL, allowing for the concise specification of required resources. The resource definitions is based on a hierarchical composition model, allowing for the creation of configurable and reusable template components, enabling developers and their teams to share and reuse common resource templates in different projects. Using the DSL, developers model their application components and their basic runtime requirements, such as (provider-independently normalized) CPU, memory, and I/O capacities, as well as dependencies and weighted communication relations between these components. The weighted component relations are used by the provisioning strategy to determine the optimal deployment configuration for the application. Moreover, resource templates allow for the definition of constraints based on costs, component proximity, and geographical distribution.

### Migration/Deployment Recipes

Deployment recipes are an important ingredient for automation in the meta cloud infrastructure. The recipes allow for controlled deployment of the application including installation of packages, starting of required services, managing package and application parameters, and establishing links between related components. Automation tools such as Chef provide an extensive set of functionalities, which are directly integrated into the meta cloud environment. Migration recipes go one step further and describe how to migrate an application during runtime, e.g., migrating storage functionality from one service provider to another. Recipes only describe initial deployment and migration, the actual process is executed by the provisioning strategy and the meta cloud proxy using aforementioned automation tools.

### Meta Cloud Proxy

The meta cloud provides proxy objects, which are deployed with the application and run on the provisioned cloud resources. They serve as mediator between the application and the cloud provider. These proxies expose the meta cloud API to the application, transform application requests into cloud provider specific requests, and forward them to the respective cloud services. The proxy provides means to execute deployment and migration recipes triggered by the meta cloud's provisioning strategy. Moreover, proxy objects send QoS statistics to the resource monitoring component running within the meta cloud. The data are gained by intercepting calls of the application to the underlying cloud services and measuring their processing time, or by executing short benchmark programs Applications can also define and monitor custom QoS metrics that are sent to the resource monitoring component via the proxy object to enable advanced, application-specific management strategies. To avoid high load and computational bottlenecks, the communication between proxies and the meta cloud is kept at a minimum. Proxies do not run inside the meta cloud, and regular service calls from the application to the proxy are not routed through the meta cloud, either.

**Resource Monitoring**

The resource monitoring component is responsible for receiving data collected by meta cloud proxies about the resources they are using, on application's request. These data are filtered and preprocessed, and then stored to the knowledge base for further processing. This helps to generate comprehensive QoS information of cloud service providers and the particular services they are providing, including response time, availability, and more service specific quality statements.

**Provisioning Strategy**

The main task of the provisioning strategy component is to match an application's cloud service requirements to actual cloud service providers. It is able to find and rank cloud services based on data in the knowledge base. The initial deployment decision is based on the resource templates, specifying the resource requirements of an application, together with QoS and pricing information about service providers. The result is a ranked list of possible combinations of cloud services regarding expected QoS and costs. At runtime, the component is able to reason about whether migration of a resource to another resource provider is beneficial based on new insights into the application's behavior and updated cloud provider QoS or pricing data. Reasoning about migrating additionally involves calculating migration costs. Decisions of the provisioning strategy result in executing customer defined deployment or migration scripts.

**Knowledge Base**

The knowledge base serves as store for data about cloud provider services, their pricing and QoS, and information necessary to estimate migration costs. Customer provided resource templates and migration/deployment recipes are stored in the knowledge base as well. Also, the knowledge base indicates which cloud providers are eligible for a certain customer. These usually comprise all providers the customer has an account with and providers that offer possibilities to create (sub) accounts on the fly. A number of different information sources contribute to the knowledge base: meta cloud proxies regularly send data about application behavior and cloud service QoS. Pricing and capabilities of cloud service providers may be either added manually or by crawling techniques able to get this information automatically, like [43].

### *4.4.3 A Meta Cloud Use Case*

A meta cloud compliant application accesses cloud services using the meta cloud API and does not directly talk to the cloud provider specific service APIs. For the particular case this means the application does not depend on Amazon's EC2, Simple Queue Service (SQS), or RDS service APIs, but on meta cloud's compute, message queue, and relational database service APIs.

For initial deployment the developer submits the application's resource template to the meta cloud. It specifies not only the three types of cloud services needed for running the sports application, but also their necessary properties and how they depend on each other. For compute resources, for instance, CPU, RAM, and disk space can be specified, according to terminology defined by the meta cloud resource template DSL. Each resource can be named in the template, which allows for referencing during deployment, runtime, and migration. The resource template

specification should also contain interdependencies, like the direct connection between the web service compute instances and the message queue service.

The rich information provided by resource templates helps provisioning strategy component to make profound decisions about cloud service ranking. The working principle for initial deployment can be explained by web search analogy, in which resource templates are queries, cloud service provider QoS and pricing information represent indexed documents. Algorithmic aspects of the actual ranking are beyond the scope of this article. If some resources in the resource graph are only loosely coupled, then it is more likely that resources from different cloud providers may be selected for a single application. In our use case, however, we assume that the provisioning strategy ranks the respective Amazon cloud services first, and that the customer follows this recommendation.

After the resources are determined the application together with an instance of the meta cloud proxy is deployed, according to customer provided recipes. During runtime, the meta cloud proxy mediates between the application components and the Amazon cloud resources, and sends monitoring data to the resource monitoring component running within the meta cloud.

Monitoring data is used to refine the application's resource template and the provider's overall QoS values, both stored in the knowledge base. This updated information is regularly checked by the provisioning strategy component, which might trigger a migration. Front end nodes could be migrated to other providers to place them closer to the application's users, for example. Another reason for a migration can be updated pricing data. After a price cut by Rackspace services may migrate to their cloud offerings. For making these decisions, potential migration costs regarding time and money need to be taken into account by the provisioning strategy component. The actual migration is performed based on customer provided migration recipes.

## 4.5 Summary

In this chapter we presented a novel approach for provider-managed, model-based adaptation of cloud computing applications. Customers are not required to implement their own adaptation solution, but can use a simple model to specify desired application behavior. The presented approach has the potential for significant cost savings and increased application quality for customers by utilizing a sophisticated adaptation infrastructure managed by the cloud provider that can offer better adaptation decisions by considering data from multiple different, but similar, customers. Furthermore, cloud providers will be able to manage their infrastructure more efficiently due to reduced resource over-provisioning resulting from improved adaptation decisions.

Deferring control over adaptation decisions to a cloud provider poses several challenges that need to be addressed. Our approach requires that customers trust providers with potentially confidential information about their applications' inner structure, such as adaptation strategies and request traffic patterns. The widespread adoption of cloud computing has shown that customers already trust reputable providers with hosting their applications, so we argue that the addition of adaptation strategies will not be problematic, provided that service contracts clearly state how provided data will be used. Customers that do not want their data used to improve adaptation decisions for others could still benefit of the provider's adaptation infrastructure, saving them implementation effort, but will in turn not be able to receive adaptation decisions optimized

using data from others. Furthermore, service level agreements need to explicitly state the new DSLs and tools needed, as well as responsibilities of both customers and providers when using managed adaptation infrastructure.

To enable the transition to a provider-managed adaptation approach, we also introduced the meta cloud abstraction layer that can help to mitigate vendor lock-in and promises transparent use of cloud computing services. Most of the basic technologies necessary to realize the meta cloud already exist, yet lack integration. For avoiding meta cloud lock-in it is critical that the community drives the ideas, to create a truly open meta cloud with added value for all customers with broad support for different providers and implementation technologies.

CHAPTER 5

# Generic Event-based Monitoring and Adaptation Methodology for Heterogeneous Distributed Systems

The Cloud computing paradigm provides the basis for a class of platforms and applications that face novel challenges related to multi-tenancy, adaptivity, elasticity, and more. To account for service delivery guarantees in the face of ever increasing levels of heterogeneity, scale and dynamism, service provisioning in the Cloud has raised the demand for systematic and flexible approaches to monitoring and adaptation of applications. In this chapter, we tackle this issue and present a framework for efficient runtime management of Cloud environments, and distributed heterogeneous systems in general. A novel domain-specific language (DSL) termed MONINA is introduced that allows to define integrated monitoring and adaptation functionality for controlling such systems. We propose a mechanism for optimal deployment of the defined control operators onto available computing resources. Deployment is based on solving a quadratic programming problem, which aims at achieving minimized reaction times, low overhead, as well as scalable monitoring and adaptation. The monitoring infrastructure is based on a distributed messaging middleware, providing high level of decoupling and allowing new monitoring nodes to join the system dynamically. We provide a detailed formalization of the problem domain, discuss architectural details, highlight the implementation of the developed prototype, and put our work into perspective with existing work in the field.

## 5.1 Overview

Efficient monitoring and adaptation of large-scale heterogeneous systems, integrating a multitude of components, possibly from different vendors, is challenging. Huge amounts of monitoring data and sophisticated adaptation mechanisms in complex systems render centralized processing of control logic impractical, as the significant network overhead could interfere with production

traffic, requiring the use of sophisticated monitoring strategies selecting only necessary status information in order to minimize communication overhead. Moreover, complex interactions and interdependencies of system components call for advanced adaptation mechanisms, allowing for simple and clear specification of overall system behavior goals, as well as fine-grained control over individual components. In distributed systems it is desirable to keep relevant monitoring and adaptation functionality as local as possible, to reduce traffic and to allow for timely reaction to changes.

In this chapter, we present an architecture and methodology for managing complex heterogeneous systems using a combination of Complex Event Processing (CEP) [115, 163] techniques to manage and enrich monitoring data, and production rule systems for defining system and component behavior goals to perform necessary adaptations. Furthermore, we introduce a domain-specific language (DSL) to easily and succinctly specify system components and their monitoring and adaptation relevant behavior. It allows to define integrated monitoring and adaptation functionality to realize applications based on top of heterogeneous, distributed components. Using the introduced DSL we then outline the process of deploying the integration infrastructure, focussing on the efficient placement of monitoring and adaptation functionality onto available resources. The presented approach is especially suited for deployments in cloud computing environments, as efficient deployment strategies are suitable to reduce infrastructure costs and increase application performance.

## 5.2 Scenario

In this section we introduce a motivating scenario based on the problems tackled in the Indenica[1] FP7 EU project. The project aims at providing methods and tools for describing, deploying and managing disparate platforms based on Virtual Service Platforms (VSPs), which integrate and unify their services.

Complex service-based business applications consist of a multitude of components, both developed in-house, as well as from third parties. Often, multiple alternative products from different vendors exist that offer similar functionalities but exhibit significant fragmentation regarding technology, cost, or quality. A flight booking service from vendor A might, for instance, be implemented to offer SOAP web service endpoints for communication, charge for every request to the system, and offer flights at competitive rates. A competing service from vendor B on the other hand might provide an Advanced Message Queueing Protocol (AMQP) interface, charge only for booked flights, and offer comparatively expensive flight rates. Depending on the application to be created, either of the offers may be more suitable, and even a combination of multiple services might be appropriate. The problem of deciding on suitable components gets exacerbated when implementing complex applications, as a large number of similar alternatives by different vendors will be available to use, each with different properties regarding dimensions such as technology, cost, or quality. It is therefore increasingly important to design applications to allow for easy and controlled migration of functionality between different components and providers.

[1] http://indenica.eu

Due to different fragmentation aspects, coordination and control of involved services must adapt to changes introduced by switching providers. Service access must be mediated to accommodate for technology differences, whereas coordination and control must be designed to easily compensate for fragmentation of aspects such as cost or quality, i.e., differences in provided functionality as well as different control policies.

Furthermore, deployment mechanisms for complex applications and their control infrastructure must be able to account for available processing capacity on involved hosts, as well as network connection properties such as cost and capacity. This is especially important for cloud applications, as efficient deployment of components results in minimized infrastructure costs and maximized application performance.

In the following, we present an architecture and framework to ease the creation, deployment, and management of applications as described above.

## 5.3 Architecture

In this section, we present an architecture for VSPs to tackle the problems outlined in the scenario above, allowing for integration of heterogenous service platforms, unified management of orchestrated behavior, as well as the addition of domain-specific functionality to be consumed by client applications.

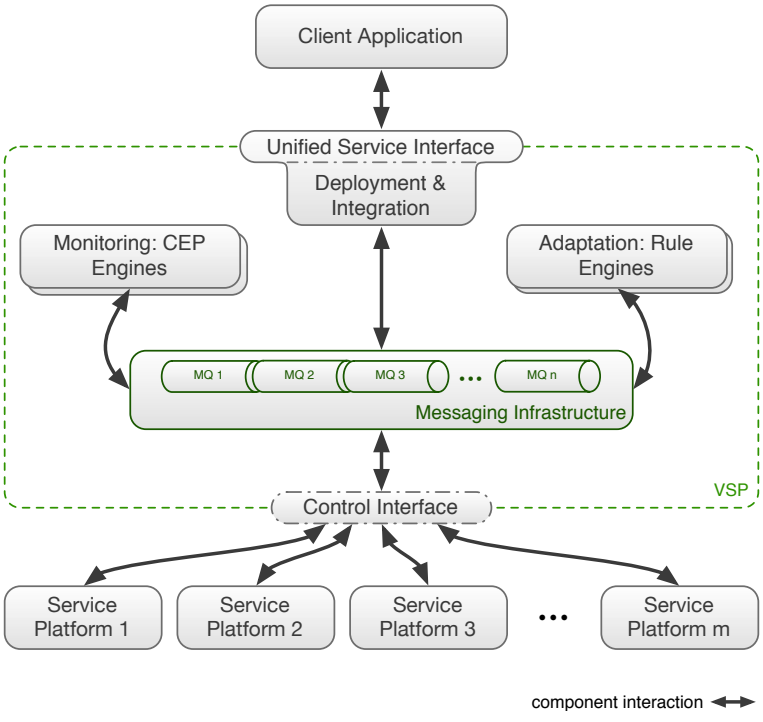



Figure 5.1: VSP Runtime Architecture

The VSP runtime architecture is presented in Fig. 5.1. A VSP provides a unified view on the functionality of the integrated service platforms that are connected by control interfaces. Monitoring and adaptation are performed by Complex Event Processing (CEP) engines and production rule engines, respectively. Communication within the VSP is based on a distributed messaging infrastructure.

The control interface allows for integration of external services using a wire format transformation layer to accommodate various technologies, such as SOAP, REpresentational State Transfer (REST), Remote Method Invocation (RMI) or messaging based solutions such as Java Messaging Service (JMS) or AMQP. Furthermore, this interface allows for the specification of emitted monitoring events, as well as supported adaptation actions of connected service platforms.

Monitoring events emitted by integrated services are used within the monitoring infrastructure to derive composite events by aggregating and enriching data emitted by multiple sources (such as the integrated platforms and VSP components) using CEP techniques. The monitoring engines allow for the specification of monitoring queries to derive complex events in order to model the system state in domain-specific terms relevant to stakeholders, abstracting from low-level metrics and system details.

The modeled system state is then used in the adaptation infrastructure by transforming state change events to facts in the adaptation knowledge base. The adaptation infrastructure utilizes production rule systems to enable sophisticated reasoning on the modeled application state to control the VSP behavior. It allows for the specification of adaptation rules that can influence the integrated systems using actions specified in the according control interface.

This clear separation of monitoring and adaptation concerns allows for independent evolution of data derived from the system state and control logic to facilitate the creation of monitoring and adaptation hierarchies. Business rule experts can specify high-level goals for the modeled application's behavior that are evaluated based on domain-specific system state indicators derived from composite monitoring events specified by system experts. The architecture is furthermore designed to allow operators to focus on specifying control logic and let the framework handle decisions about where and how the specified control infrastructure is physically deployed.

Communication between components is realized using a distributed messaging fabric that enables to minimize unnecessary network traffic (compared to a centralized message bus deployment) and further allows components to move freely within the network without changing connection bindings or losing connectivity to the system. The execution of monitoring and adaptation on top of multiple engines further allows for scalable control using distributed resources.

To enable the simplified specification of an application based on the presented architecture we introduce a new DSL called MONINA, which allows the user to specify service platform capabilities, monitoring queries, and adaptation rules. The MONINA language is presented in Section 5.4. While the presented architecture allows for simple distributed deployment of complex runtime environments, efficient and effective distributed deployment by optimizing component placement poses several challenges. In a deployed system, operating cost and network overhead should minimal, but the provisioned compute resources must be able to handle the processing load of all deployed components. When external services are integrated, involved components should furthermore be placed "close to" their communication peers to reduce network latency and possibly

transmission cost. Strategies to tackle the presented problems and deploy the specified functionality onto available resources will be discussed in Section 5.5. The prototype implementation based on the presented concepts is discussed in Section 5.6.

## 5.4 MONINA Language

In this section we introduce MONINA (MONitoring, INTegration, Adaptation) a DSL allowing for concise, easy, and reusable specification of platforms integrated into a VSP, along with monitoring and adaptation rules governing their behavior.

The language is developed using the Xtext [48] language development framework, allowing for tight integration in the Eclipse platform. The plugin offers syntax highlighting, as well as several automated sanity checks to ease system specification. The language plugin is furthermore integrated into the overall Indenica tool suite, allowing for the usage of existing system models stored in the infrastructure repository. Future versions of the plugin will offer a graphical abstraction in addition to the textual DSL for increased simplicity and ease of use.

The listing in Figure 5.2 shows a simple definition for a service platform to be integrated into a VSP The 'ApplicationServer' component emits 'RequestFinished' events after processing requests and supports a 'DecreaseQuality' action, which can be triggered by adaptation rules. Emitted events are processed by the 'AggregateResponseTime' query, which aggregates them over five minutes, creating an 'AverageProcessingTime' event. This event is converted to a fact, which might trigger 'DecreaseQualityWhenSlow' adaptation rule. The physical infrastructure consists of hosts 'vm1' and 'vm2'. Runtime elements without defined costs are assigned default values, which are refined at runtime. In the following, we will discuss the most important language constructs of MONINA in more detail.

**event** Monitoring events are described listing attributes contained in emitted messages. Events are then used in component definitions, monitoring query declarations, as well as facts.

**fact** Facts constitute the knowledge base for adaptation actions. Fact definitions reference an event type and a partition key.

**action** Similar to events, adaptation actions list all their valid parameters. Actions are used in component definitions as well as adaptation rules.

**component** A component definition references all monitoring events the platform can emit (including their frequency), all adaptation actions that can be performed, as well as its processing requirements. Furthermore, it is correlated with a concrete instance of the component in question at deployment.

**query** Monitoring queries are used to define the aggregation, filtering and enrichment of emitted monitoring data in a CEP fashion. Monitoring rules will either emit complex aggregated events to be consumed by other monitoring rules, directly issue adaptation actions, or emit facts to be used in adaptation rules.

```
event RequestFinished {
  request_id : Integer
  processing_time_ms : Integer
}

event AverageProcessingTime {
  processing_time_ms : Integer
}

action DecreaseQuality {
  amount : Double
}

component ApplicationServer {
  endpoint {
    at "/app_server"
    emit RequestFinished
    action AdjustQuality
  }
  host vm1
  cost 32
}

host vm1 { capacity 128 }
host vm2 { capacity 256 }

query AggregateResponseTimes {
  from ApplicationServer
  event RequestFinished as req
  emit AverageProcessingTime(avg(req.processing_time_ms))
  window 5 minutes
}

fact {
  from AverageProcessingTime
}

rule DecreaseQualityWhenSlow {
  from AverageProcessingTime as apt
  when apt.processing_time_ms > 2000
  execute ApplicationServer.DecreaseQuality(5)
}
```

Figure 5.2: Sample MONINA System Definition

**rule** Adaptation rules allow for the usage of complex business management rules to govern system behavior. Monitoring rules emit facts to be used for reasoning over the current system state. Adaptation rules can either publish new facts or issue adaptation actions.

**host** Hosts represent possible deployment locations of components, monitoring queries and adaptation rules. A host description contains its processing capacity.

### *5.4.1 Event*

In our work, we follow the event-based interaction paradigm [66]. Events are emitted by components to signal important information. Furthermore, events can be emitted by monitoring queries as a result of the aggregation or enrichment of one or more source events.

Figure 5.3 shows a simplified grammar of the event construct in Extended Backus-Naur Form (EBNF). Event declarations start with the **event** keyword and an event type identifier. As shown in the figure, an event can contain multiple attributes, defined by specifying name and type separated by a colon. Currently, supported event types are a variety of Java types such as `String`, `Integer`, and `Decimal`, and `Map<?,?>`.

⟨*event*⟩ ::= '`event`' ⟨*ID*⟩ '`{`' ⟨*attr*⟩* '`}`'

⟨*attr*⟩ ::= ⟨*attr-name*⟩ '`:`' ⟨*type*⟩

⟨*attr-name*⟩ ::= ⟨*ID*⟩

Figure 5.3: Simplified Event Grammar in EBNF

Since listing all available event types for every application would be a tedious and error-prone task, we automatically gather emitted event types from known components to improve reusability and ease of use. This procedure is described in more detail in Section 5.4.4.

More formally, we assume that $E$ is the set of all event types, $T$ is the set of all data types, and each event type $E' \in E$ is composed of event attribute types $E' = (\alpha_1, \ldots, \alpha_k)$, $\alpha_i \in T \; \forall i \in \{1, \ldots, k\}$. $\mathscr{I}_E$ denotes the set of monitoring event instances (or simply events), and each event $e \in \mathscr{I}_E$ has an event type, denoted $t(e) \in E$. The attribute values contained in event $e$ are represented as a tuple $e = (\pi_{\alpha_1}(e), \ldots, \pi_{\alpha_k}(e))$, where $\pi_{\alpha_x}(e)$ is the projection operator (from relational algebra), which extracts the value of some attribute $\alpha_x$ from the tuple $e$.

### *5.4.2 Action*

Complementary to monitoring events described above, adaptation actions are another basic language element of MONINA. Adaptation actions are invoked by adaptation rules and executed by corresponding components to modify their behavior. Figure 5.4 shows a simplified grammar of the action construct in EBNF. Action declarations start with the **action** keyword followed by the action type identifier. Furthermore, actions can take parameters, modeled analogously to event attributes shown in Figure 5.3.

⟨*action*⟩ ::= 'action' ⟨*ID*⟩ '{' ⟨*attr*⟩* '}'

Figure 5.4: Simplified Action Grammar in EBNF

Similar to events, adaptation actions offered by known components do not need to be specified manually, but are automatically gathered from component specifications, as mentioned in Section 5.4.4.

The symbol $A$ denotes the set of all types of adaptation actions, and each type $A' \in A$ contains attribute types: $A' = (\alpha_1, \ldots, \alpha_k)$, $\alpha_i \in T \ \forall i \in \{1, \ldots, k\}$. The set $\mathscr{I}_A$ stores all action instances (or simply actions) that are issued in the system. The values of an action $a \in \mathscr{I}_A$ are evaluated using the projection operator (analogously to event attributes): $a = (\pi_{\alpha_1}(a), \ldots, \pi_{\alpha_k}(a))$.

### *5.4.3 Fact*

Facts constitute the knowledge base for adaptation rules and are derived from monitoring events. A fact incorporates all attributes of the specified source event for use by adaptation rules. Figure 5.5 shows a simplified grammar of the fact construct in EBNF. Fact declarations start with the **fact** keyword and an optional fact name. A fact must specify a source event type that is used to derive the fact from. Furthermore, an optional partition key can be supplied. If the fact name is omitted, the fact will be named after its source event.

⟨*fact*⟩ ::= 'fact' ⟨*ID*⟩? '{' ⟨*ID*⟩ ⟨*partition-key*⟩? '}'
⟨*partition-key*⟩ ::= 'by' ⟨*ID*⟩

Figure 5.5: Simplified Fact Grammar in EBNF

The partition key construct is used to enable the creation of facts depending on certain event attributes, allowing for the concise declaration of multiple similar facts for different system aspects. For instance, a fact declaration for the event type `ProcessingTimeEvent` that is partitioned by the `component_id` attribute will create appropriate facts for all encountered components, such as `ProcessingTime(Component1)`, ..., `ProcessingTime(ComponentN)`. In contrast, a fact declaration for the `MeanProcessingTimeEvent` without partition key will result in the creation of a single fact representing the system state according to the attribute values of incoming events.

Formally, a fact $f \in F$ is represented as a tuple $f = (\kappa, e)$, for event type $e \in E$ and partition key $\kappa$. The optional partition key $\kappa$ allows for the simplified creation of facts concerning specified attributes, to model facts relating to single system components, using $\pi_\kappa(e)$, the projection of attribute $\kappa$ from event $e$. Alternatively, the type of event $e$ itself acts as the partition key, aggregating all events of the same type to a single fact.

### 5.4.4 Component

A component declaration incorporates all information necessary to integrate third-party system into the Indenica infrastructure. Figure 5.6 shows a simplified grammar of the component construct in EBNF. Component declarations start with the **component** keyword and a component identifier. A component specifies all monitoring events it will emit with an optional occurrence frequency, supported adaptation actions, as well as a reference to the host the component is deployed to.

⟨*component*⟩ ::= '`component`' ⟨*ID*⟩ '`{`' ⟨*metadata*⟩? ⟨*c-elements*⟩* ⟨*host-ref*⟩ '`}`'

⟨*metadata*⟩ ::= ('`vendor`' ⟨*STRING*⟩)? ('`version`' ⟨*STRING*⟩)? ...

⟨*c-elements*⟩ ::= ⟨*endpoint*⟩ | ⟨*refs*⟩

⟨*refs*⟩ ::= ⟨*event-ref*⟩ | ⟨*action-ref*⟩

⟨*action-ref*⟩ ::= '`action`' ⟨*ID*⟩

⟨*event-ref*⟩ ::= '`event`' ⟨*ID*⟩ ⟨*frequency*⟩?

⟨*endpoint*⟩ ::= '`endpoint`' ⟨*ID*⟩? '`{`' ⟨*e-addr*⟩ ⟨*refs*⟩* '`}`'

⟨*frequency*⟩ ::= '`every`' ⟨*Number*⟩ '`seconds`' | ⟨*Number*⟩ '`Hz`'

⟨*host-ref*⟩ ::= '`host`' ⟨*ID*⟩

Figure 5.6: Simplified Component Grammar in EBNF

For brevity, further elements such as endpoint addresses, are omitted in the presented grammar snippet but are included in Appendix B.

As mentioned before, it is usually not necessary to manually specify component, action, and event declarations. The Indenica infrastructure provides for means to automatically gather relevant information from known components through the control interface shown in Figure 5.1.

Formally, components $c \in C$ are represented with the signature function[2]

$$sig : C \rightarrow \mathscr{P}(A) \times \mathscr{P}(\{(e_j, \nu_j) | e_j \in E, \nu_j \in \mathbb{R}_0^+\}) \times \mathbb{R}_0^+ \times H$$

and the signature for a component $c_i$ is

$$sig(c_i) \mapsto (I_i^A, \Omega_i^E, \psi_i, h_i)$$

The signature function *sig* extracts relevant information from the according language construct for later use by the deployment infrastructure. Monitoring events emitted by the component are represented by $\Omega_i^E$, and for each emitted event type $e_j$ an according frequency of occurrence $\nu_j$ is supplied. Adaptation actions supported by the component are denoted by $I_i^A$, its processing cost is represented by $\psi_i$, and $h_i$ identifies the host the component is deployed to.

[2]For clarity, we use the same symbol *sig* for signatures of components (Section 5.4.4), monitoring queries (Section 5.4.5), adaptation rules (5.4.6), and hosts (Section 5.4.7).

### 5.4.5 *Monitoring Query*

Monitoring queries allow for the analysis, processing, aggregation and enrichment of monitoring events using CEP techniques. In the context of the Indenica project we provide a simple query language tailored to the needs of the specific solution.

A simplified EBNF grammar of the monitoring query construct is shown in Figure 5.7. A query declaration starts with the **query** keyword and a query identifier. Afterwards, an arbitrary number of event sources for the query is specified using the **from** and **event** keywords to specify source components and event types. A query then specifies any number of event emission declarations, denoted by the **emit** keyword followed by the event type and a list of expressions evaluating the attribute assignments of the event to be emitted. For brevity we omit the specification of ⟨*cond-expression*⟩ clause that represents a SQL-style conditional expression. Queries can be furthermore designed to operate on event stream windows using the **window** keyword, specifying either a number of events to create a batch window or a time span to create a time window. Conditions expressed using the **where** keyword are used to limit query processing to events satisfying certain conditions, using the conditional expression construct mentioned above. Finally, queries can optionally indicate the rate of incoming vs. emitted events, as well as an indication of required processing power. These values are user-defined estimations in the initial setup, and are adjusted continuously during runtime to accommodate changes in the environment.

⟨*query*⟩ ::= 'query' ⟨*ID*⟩ '{' (⟨*sources*⟩ | ⟨*emits*⟩)*
⟨*window*⟩? ⟨*condition*⟩? ⟨*io-ratio*⟩? ⟨*cost*⟩? '}'

⟨*sources*⟩ ::= 'source' ⟨*ID*⟩ (',' ⟨*ID*⟩)*
'event' ⟨*ID*⟩ (',' ⟨*ID*⟩)*

⟨*emits*⟩ ::= 'emit' ⟨*ID*⟩ (⟨*attr-emit*⟩*)*

⟨*attr-emit*⟩ ::= ⟨*cond-expression*⟩ ('as' ⟨*ID*⟩)?

⟨*window*⟩ ::= 'window' (⟨*batch-window*⟩ | ⟨*time-window*⟩)

⟨*batch-window*⟩ ::= ⟨*Integer*⟩ 'events'

⟨*time-window*⟩ ::= ⟨*Integer*⟩ ('seconds' | 'minutes' | 'days' | ...)

⟨*condition*⟩ ::= 'where' ⟨*cond-expression*⟩

⟨*io-ratio*⟩ ::= 'ratio' ⟨*Number*⟩

⟨*cost*⟩ ::= 'cost' ⟨*Number*⟩

Figure 5.7: Simplified Monitoring Query Grammar in EBNF

In addition to the query construct presented above, the language infrastructure allows for the integration of other CEP query languages, such as Esper Event Processing Language (EPL) [51] if necessary.

The set of queries $q_i \in Q$ is represented using the signature

$$sig : Q \mapsto \mathscr{P}(E) \times \mathscr{P}(E) \times \mathbb{R}_0^+ \times \mathbb{R}_0^+$$

and the signature for a query $q_i$ is

$$sig(q_i) \mapsto (I_i^E, O_i^E, \rho_i, \psi_i)$$

Input and output event streams are denoted by $I_i^E$ and $O_i^E$ respectively, while $\rho_i$ represents the ratio of input events processed to output events emitted, and $\psi_i$ represents the processing cost of the query.

### *5.4.6 Adaptation Rule*

Adaptation rules employ a knowledge base consisting of facts to reason on the current state of the system and modify its behavior when necessary using a production rule system. Figure 5.8 shows a simplified grammar of the adaptation rule construct in EBNF. A rule declaration starts with the **rule** keyword and a rule identifier. After importing all necessary facts using the **from** keyword, a rule contains a number of **when**-statements where the condition evaluates a ⟨*cond-expression*⟩ as described above, referencing imported facts, and the **then** block specifies a number of adaptation action invocations including any necessary parameter assignments. Optionally, a rule can indicate processing requirements (cf. Figure 5.7) that will be adjusted at runtime.

⟨*rule*⟩ ::= '`rule`' ⟨*ID*⟩ '`{`' (⟨*r-sources*⟩)+ ⟨*stmt*⟩+ ⟨*cost*⟩?'`}`'

⟨*r-sources*⟩ ::= '`from`' ⟨*ID*⟩ ('`as`' ⟨*ID*⟩)?

⟨*stmt*⟩ ::= '`when`' ⟨*cond-expression*⟩ '`then`' ⟨*action-expr*⟩+

⟨*action-expr*⟩ ::= ⟨*ID*⟩ '`(`' ⟨*action-attr*⟩ ('`,`' ⟨*action-attr*⟩)* '`)`'

⟨*action-attr*⟩ ::= ⟨*cond-expression*⟩ ('`as`' ⟨*ID*⟩)?

Figure 5.8: Simplified Adaptation Rule Grammar in EBNF

As with monitoring queries, the adaptation rule module is tailored to the requirements of the Indenica infrastructure but also allows for the usage of different production rule languages, such as the Drools [83] rule language, if more complex language constructs are required.

More formally, the set of rules $r_i \in R$ is represented with the signature function

$$sig : R \mapsto \mathscr{P}(F) \times \mathscr{P}(A) \times \mathbb{R}_0^+$$

and the signature for a rule $r_i$ is

$$sig(r_i) \mapsto (I_i^F, O_i^A, \psi_i)$$

The set of facts from the knowledge base used by the adaptation rule are denoted by $F_i$, while $A_j$ represents the adaptation actions performed, and $\psi_i$ represents the processing cost of the rule.

### 5.4.7 Host

Hosts represent the physical infrastructure available for deployment of infrastructure components. Figure 5.9 shows a simplified grammar of the host construct in EBNF. A host declaration starts with the **host** keyword and a host name. An address in the form of a Fully-Qualified Domain Name (FQDN) or an IP address can be supplied. If no address is given, the host name will be used instead. Furthermore, a **capacity** indicator is provided that will be used for deployment decisions.

⟨*host*⟩ ::= 'host' ⟨*ID*⟩ '{' ⟨*address*⟩? ⟨*capacity*⟩ '}'

⟨*address*⟩ ::= ⟨*fqdn*⟩ | ⟨*ip-address*⟩

⟨*capacity*⟩ ::= 'capacity' ⟨*Number*⟩

Figure 5.9: Simplified Host Grammar in EBNF

The set of hosts $h_i \in H$ is represented with the signature function

$$sig : H \mapsto \mathbb{R}_0^+$$

and the signature for a host $h_i$ is

$$sig(h_i) \mapsto (\psi_i)$$

with the capacity of a host represented by $\psi_i$.

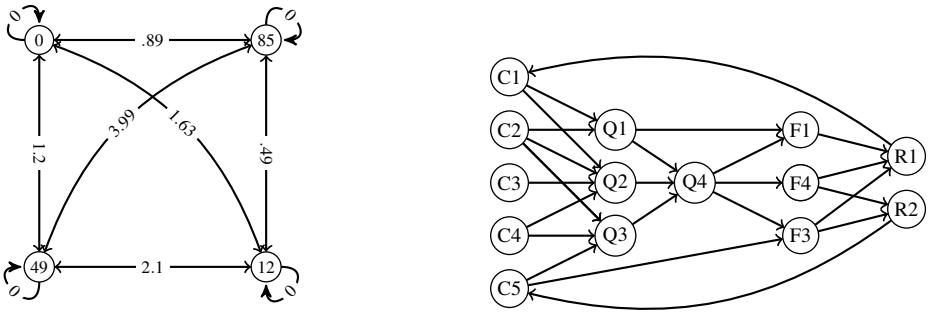
As mentioned above, the EBNF snippets illustrating the introduced concepts have been shortened for simplicity and clarity. A complete listing of the language specification can be found in Appendix B.

## 5.5 Deployment of Monitoring Queries and Adaptation Rules

In this section, we propose a methodology for efficiently deploying runtime elements. Deployment is based on a MONINA definition. The deployment strategy tries to find an optimal placement with regard to locality of information producers and consumers, resource usage, network load, and minimal reaction times. Our deployment procedure consists of three main stages. First, an infrastructure graph is generated from the host declarations in the MONINA definition to create a model of the physical infrastructure. Then, a dependency graph is derived from component, query, fact, and rule definitions. Finally, a mathematical optimization problem is formulated based on both graphs, which finds an optimal deployment scheme.

### 5.5.1 Infrastructure Graph

The infrastructure graph models the available infrastructure. Its nodes represent execution environments. We will refer to execution environments as hosts, even though they might not only




(a) Infrastructure Graph

(b) Dependency Graph

Figure 5.10: Graphs generated from a MONINA description

represent single machines, but more complex execution platforms. The graph's edges represent the connection between hosts. Formally, the infrastructure graph $G_I = (V_I, E_I)$ is a directed graph. Capacity function $c_I : V_I \to \mathbb{R}_0^+$ assigns each host its capacity for hosting runtime elements, e.g., monitoring queries or adaptation rules. A capacity of zero prohibits any runtime elements on the host. Edge weight function $w_I : E_I \to \mathbb{R}_0^+$ models the delay between two hosts. Values close to zero represent good connection. For the sake of convenience we assume that each vertex has a zero weighted edge to itself. Figure 5.10a shows an exemplary infrastructure graph.

The infrastructure graph is generated based on a MONINA description, i.e., its node set $V_I$ is taken from the description file, which also contains the hosts' physical addresses. The next step is the exploration of the edges based on the *traceroute* utility, which is available for all major operating systems. It allows, amongst others, measuring transit delays. Furthermore, node capacities can be read by operating system tools to complement missing MONINA values. In Unix-like operating systems, for instance, the /proc pseudo-filesystem folder provides information about hardware and its utilization.

### *5.5.2 Dependency Graph*

Dependency graphs model the dependencies between components, monitoring queries, facts, and adaptation rules. A dependency graph $G_D = (V_D, E_D)$ is a directed, weighted graph, whose node set $V_D = C \cup Q \cup F \cup R$ is composed of pairwise disjoint sets $C$, $Q$, $F$, and $R$. These represent components, queries, facts, and rules, respectively. Edges represent dependencies between these entities, i.e., exchange of events, and weight function $w_D : E_D \to \mathbb{R}_0^+$ quantifies the relative number of events. Another function $e_D : E_D \to E$ maps edges to events they are based on, where $E$ is the set of event types. Components are event emitters, which may be consumed by queries or may be converted into a fact in a knowledge base. Queries consume events from components or other queries producing new events. Knowledge bases convert certain events into facts. Rule engines work upon knowledge bases, and trigger rules if respective conditions become true. Edges link event emitters (components or queries) to respective event consumers (queries or knowledge bases). They also connect knowledge bases to rules relying on facts they are managing. Finally,

rules are linked to the components they are adapting, i.e., components in which they trigger adaptation actions. Thus, the edge set is limited to the following subset $E_D \subseteq (C \times Q) \cup (C \times F) \cup (Q \times Q) \cup (Q \times F) \cup (F \times R) \cup (R \times C)$. Figure 5.10b shows an exemplary dependency graph. Event types and edges weights are omitted for readability.

The generation of a dependency graph is based on a MONINA description. Initially, the dependency graph $G_D = (V_D, E_D)$ is created as a graph without any edges, i.e., $V_D = C \cup Q \cup F \cup R$ and $E_D = \emptyset$, where $C$, $Q$, $F$, $R$ are taken from the MONINA description. Then, edges are added according to the following edge production rules.

**Component → Query.** An edge $c \xrightarrow{\psi} q$ is added to $E_D$ for every component $c \in C$, query $q \in Q$, and event $e \in (O^E \cap I^E)$, where $sig(c) = ((O^E, \bullet), \bullet, \psi)$ and $sig(q) = (I_i^E, \bullet, \bullet, \bullet)$. In case an edge $c \xrightarrow{\psi_2} q$ is supposed to be added to $E_D$, but $E_D$ already contains $c \xrightarrow{\psi_1} q$, then the latter is replaced by $c \xrightarrow{\psi_1 + \psi_2} q$. For all following edge production rules we assume that edges that already exist are merged by adding weights, like here.

**Component → Fact.** An edge $c \xrightarrow{\psi} f$ is added to $E_D$ for every component $c \in C$, fact $f \in F$, and event $e \in O^E$, where $sig(c) = ((O^E, \bullet), \bullet, \psi)$ and $f = (\bullet, e)$.

**Query → Query.** An edge $q_1 \xrightarrow{\rho} q_2$ is added to $E_D$ for all queries $q_1, q_2 \in Q$ and event $e \in (O^E \cap I^E)$, where $q_1 \neq q_2$, $sig(q_1) = (\bullet, O^E, \rho, \bullet)$ and $sig(q) = (I_i^E, \bullet, \bullet, \bullet)$.

**Monitoring Query → Fact.** An edge $q \xrightarrow{\rho} f$ is added to $E_D$ for every query $q \in Q$, fact $f \in F$, and event $e \in O^E$, where $sig(q) = (\bullet, O^E, \rho, \bullet)$ and $f = (\bullet, e)$.

**Fact → Adaptation Rule.** An edge $f \to r$ is added to $E_D$ for every fact $f \in F$ and adaptation rule $r \in R$, where $f \in I^F$ and $sig(r) = (I^F, \bullet, \bullet)$.

**Adaptation Rule → Component.** An edge $r \to c$ is added to $E_D$ for every adaptation rule $r \in R$ and component $c \in C$, where $a \in (O^A \cap I^A)$, $sig(r) = (\bullet, O^A, \bullet)$ and $sig(c) = (I^A, \bullet, \bullet)$.

### *5.5.3 Quadratic Programming Problem Formulation*

Quadratic programming [17] is a mathematical optimization approach, which allows to minimize/maximize a quadratic function subject to constraints. Assume that $\mathbf{x}, \mathbf{b}, \mathbf{c}, \mathbf{d} \in \mathbb{R}^n$ are column vectors, and $Q \in \mathbb{R}^{n \times n}$ is a symmetric matrix. Then, a quadratic programming problem can be defined as follows.

$$\min_x \ f(\mathbf{x}) = \tfrac{1}{2}\mathbf{x}^T Q\mathbf{x} + \mathbf{c}^T\mathbf{x}$$

Subject to

$$E\mathbf{x} = \mathbf{d} \qquad \text{(Equality constraint)}$$
$$A\mathbf{x} \leq \mathbf{b} \qquad \text{(Inequality constraint)}$$

We want to achieve an optimal mapping of the dependency graph onto the infrastructure graph. Runtime entities described in the dependency graph that depend on each other should

be as close as possible, in the best case running on the same host. This results in fast reactions, timely adaptations, and low network overhead. On the other hand, hosts have capacity restrictions, which have to be considered. Adding more hosts (scaling out) is often the only possibility to cope with growing load. Our mapping approach is able to find the *optimal* tradeoff between the suboptimal strategies (1) putting everything on the same host and (2) evenly/randomly distribute runtime elements among the available hosts.

Since we want to get a mapping from the optimization process, we introduce placement variables $p_{v_I,v_D}$ for each host $v_I \in V_I$ in the dependency graph and each runtime element $v_D \in V_D$ in the dependency graph. Each of these variables has a binary domain, i.e., $p_{v_I,v_D} \in \{0,1\}$. The assignment $p_{v_I,v_D} = 1$ decodes that runtime element $v_D$ is hosted on $v_I$, $p_{v_I,v_D} = 0$ stands for $v_D$ is not running on host $v_I$. This results in $|V_I| \cdot |V_D|$ binary variables, whose aggregation can be represented as a single vector $\mathbf{p} \in \{0,1\}^{|V_I|\cdot|V_D|}$.

To find out the optimal mapping of the dependency graph onto the infrastructure, we solve the following optimization problem, which can be classified as binary integer quadratic programming problem, based on the form of variable $\mathbf{p}$ and the function to minimize.

$$\min_{p} \sum_{e_I \in E_I} w_I(e_I) \cdot \sum_{e_D \in E_D} w_D(e_D) \cdot p_{v_I^1,v_D^1} \cdot p_{v_I^2,v_D^2} \tag{5.1}$$

Subject to

$$\forall c \in C: \; p_{h(c),c} = 1 \tag{5.2}$$

$$\forall v_D \in V_D: \sum_{v_I \in V_I} p_{v_I,v_D} = 1 \tag{5.3}$$

$$\forall v_I \in V_I: \sum_{v_D \in V_D} p_{v_I,v_D} \cdot c_D(v_D) \; \leq c_I(v_I) \tag{5.4}$$

The function to minimize (1) calculates for each edge $e_I = (v_I^1, v_I^2)$ in the infrastructure graph and each edge $e_D = (v_D^1, v_D^2)$ the weight that incurs if this particular dependency edge is mapped to this particular infrastructure edge. If both runtime elements ($v_D^1$ and $v_D^2$) are mapped to the same node no weight is added to the function, because all self-links have weight zero. The first equality constraint (2) fixes the mapping for every component $c \in C \subseteq V_D$ to the hosts they are statically assigned to, as defined in MONINA and represented by $h(c)$, where $sig(c) = (\bullet, \bullet, \bullet, h)$. We assume that components are bound to hosts. If there exist components that can be deployed on any host and do not have an assignment in MONINA, then this can be handled by simply omitting the respective constraint for this component. The second equality constraint (3) defines that each node from the dependency graph is mapped to exactly one node in the infrastructure graph. Finally, the inequality constraint (4) requires that for all hosts the summarized costs of all elements they are hosting is less than the respective capacity. The function $c_D : V_D \rightarrow \mathbb{R}_0^+$ represents the costs of executing a runtime element $v_D$, as defined in the MONINA description.

We use the Gurobi optimizer [60] for solving the optimization problem as described above. Runtime aspects of the currently implemented deployment module are discussed in Section 5.6.

### 5.5.4 *Deployment in Cloud Computing Environments*

The presented approach is suitable for continuous deployments in order to react to changes in the runtime environment. If new rules are added or communication characteristics change significantly we derive a new deployment strategy based on the existing structure. The goal of continuous (re-)deployment is to maintain a near-optimal component distribution while minimizing the changes to be performed. By moving as little components as possible, we minimize the cost of transferring component state information between machines.

This model of continuous optimization and re-deployment integrates perfectly with the concept of Cloud computing [8], which allows to dynamically allocate and release computing resources to implement elastically scaling applications. Cloud environments fulfill two prerequisites which are central to our approach. First, the Cloud provides the possibility to acquire a practically unbounded number of VM instances. In the optimization procedure of our approach, application components and monitoring queries are placed on hosts, and Cloud computing effectively removes any potential limits of the optimization procedure with regards to the number of hosts. Our approach considers this by dynamically adjusting the number of hosts available for deployment planning. Hosts are added to the solution space until a solution can be found that does not violate any placement constraints. Second, cost aspects are typically an integral part of (commercial) Cloud offerings, hence we can directly incorporate the computation and communication costs into our optimization model. In addition, many Cloud providers offer a convenient set of pre-configured software tools which simplify the implementation of our approach, including distributed messaging fabrics for de-centralized event transmission, host and network monitoring tools for obtaining the decision basis of our optimization, data storage services for persisting (event) data, and more.

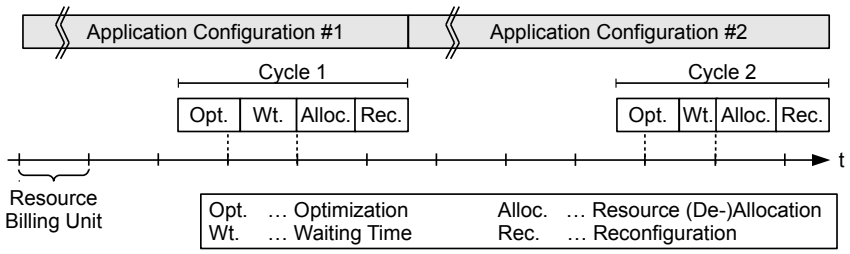



Figure 5.11: Cost-Efficient Optimization and Re-Configuration in Cloud Environments

Note, however, that the commercial nature of Cloud computing entails certain peculiarities, which should be taken into account for our approach. In particular, Cloud resources are typically subject to a billing cycle (e.g., VMs are billed in units of one hour), which requires that the optimization approach be adjusted in order to achieve optimal results. To address this issue, we suggest to perform adaptations in cycles, as illustrated in Figure 5.11. The figure shows a timeline which is split up into the billing units of computing resources, e.g., one hour (for simplification we assume that all resources are stopped/started simultaneously in each cycle). The grey bars at the top illustrate the current configuration (first #1, then #2) of the application whose deployment we strive to optimize. We assume that two adaptations are triggered over the duration of the

timeline, consisting of four main parts each: optimization procedure, waiting time, allocating and de-allocating of resources, and reconfiguration of the application based on the new resource allocations. The essential part is that the change in resource allocation should be aligned with the expiry time of the resource billing unit. If this alignment were not implemented, the unused resource utilization corresponding to the "waiting time" would be wasted from a cost perspective. Depending on the duration of the optimization algorithm (in relation to the resource billing unit), the duration of the waiting time should be minimized, in order to avoid changes in the environmental conditions which could potentially result in a different optimum at the time the adaptations get applied.

## 5.6 Implementation

In this section we discuss the implementation of the presented concepts. The developed prototype is available for download from the prototype web site[3]. As mentioned above, application specification using the MONINA language is implemented as Eclipse plugin. We use the Xtext [48] language development framework to model MONINA. The plugin offers convenience functions such as syntax highlighting, code completion and static analysis of system specifications to detect definition errors. Fig. 5.12 shows a screenshot of the editor, illustrating some of the implemented features. After specifying the relevant system structure, the MONINA plugin generates a set of configuration directives to be used with the runtime infrastructure.

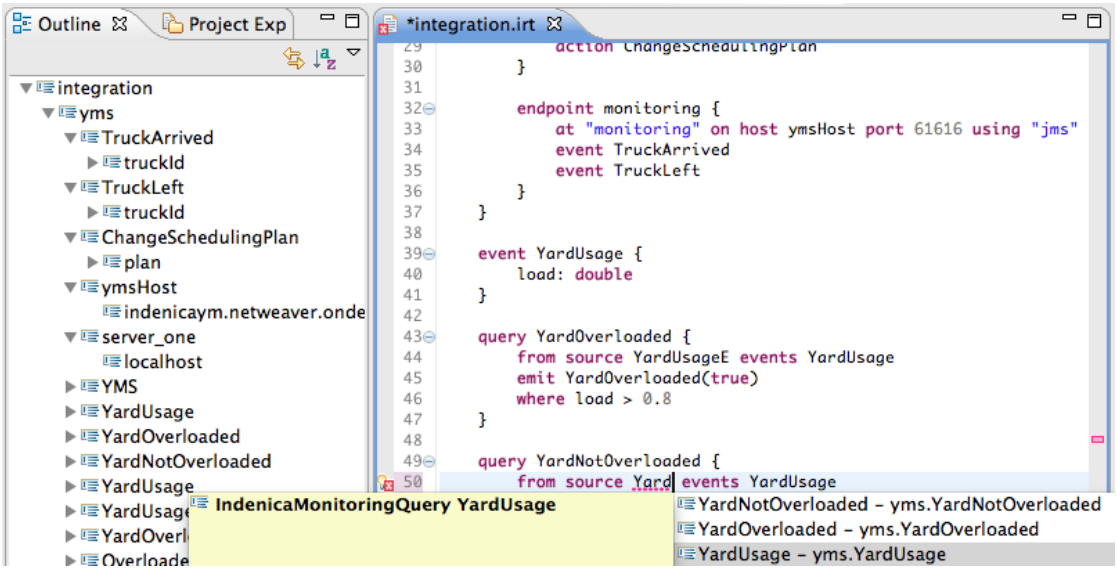



Figure 5.12: Sample Screenshot of MONINA Editor

The runtime infrastructure implements the architecture presented in Section 5.3. At the core of the runtime, a distributed messaging fabric (currently based on embedded ActiveMQ [53]

[3] http://indenicatuv.github.io/releases/

brokers) allows for extensible realization of distributed applications. The messaging fabric automatically establishes a communication mesh between application components deployed in the same network using multicast discovery. Components deployed in different networks need to know a single address per external network to establish connections to all relevant modules of the deployed app. When using a MONINA description for deployment, this information can be gathered from the contained `host` declarations and the deployment strategy. The messaging fabric furthermore establishes conventions for component discovery and management, such as common topic names and management endpoint addresses for infrastructure components to subscribe to. Furthermore, the messaging fabric allows components to register for communication paths or event streams they are interested in to create the runtime interaction structure. Messages are then delivered to components via the best available path (considering latency and bandwidth) to significantly reduce the introduced traffic overhead compared to solutions using centralized messaging middleware. Future versions of the framework will take additional factors, such as communication cost into account to enable more fine-grained control over the communication behavior of deployed applications.

The control interface is realized using the Apache Tuscany [56] Service Component Architecture (SCA) container to allow for easy integration of different interface technologies, such as SOAP and REST web services, JMS, RMI or the Common Object Request Broker Architecture (CORBA). Configuration directives from the MONINA application specification are used to establish connections to external components by registering monitoring event sinks and adaptation endpoints.

As described in the architecture in Section 5.3, the monitoring engines process event streams from integrated external components as well as queries running within the system to derive complex events representing relevant application state changes. In the current implementation, we use the Esper [51] CEP engine to perform event stream processing. Source events are received via the messaging fabric, processed using monitoring queries defined in the MONINA description, and derived events are handed back to the messaging infrastructure to make them available for further processing by other monitoring queries or adaptation rules.

The adaptation engines execute production rules on the current system state to influence its properties in order to maintain or achieve desired behavior. In the current implementation, we use the Drools Expert [83] rule engine to execute adaptation rules as specified in the MONINA system description. To establish a knowledge base from the available system state change events, a fact transformer component is used, translating a stream of state change events into a fact object representation that can be used in production rules. In the current implementation, fact transformation is handled using the Esper CEP engine, aggregating state changes into appropriate facts. Adaptation rules act on conditions about the state of facts in the knowledge base and can execute adaptation actions on integrated components. Actions to be executed are delivered to the according control interfaces by the messaging fabric. The control interface will then perform the actual execution of adaptation actions on the external component.

Control interface specifications, monitoring queries, fact transformation, and adaptation rules can be submitted to the system in multiple formats. MONINA descriptions are supported as a portable, technology-independent behavior specification, but component-native directives, such as raw EPL queries or Drools Rule Language (DRL) rules are supported by the currently imple-

mented monitoring and adaptation engines respectively.

Application deployment is carried out using the deployment component. It analyzes the supplied system specification, starts components on the available nodes according to computed deployment strategy, and deploys all necessary configuration artifacts such as endpoint definitions for external communication, monitoring queries, fact transformation rules, and adaptation rules. The deployment strategy is currently realized using the Gurobi [60] optimizer to solve the mapping problem discussed in Section 5.5. In the current version, a MONINA specification will be deployed according to cost and communication traffic estimations provided with the system description and can be redeployed based on interaction information gathered during runtime. In the future, we will extend the deployment strategy module to allow incremental deployments considering the costs of migrating existing components, rules, and queries.

Extensible interface design throughout the implemented framework allows for easy extension or replacement of components if required to avoid potential vendor lock-in.

## 5.7 Related Work

In this section we discuss important previous work related to event-based monitoring and adaptation, as well as optimized deployment of query operators in monitoring infrastructures. Although some of the seminal work dates back to the pre-Cloud era, we also emphasize the relevance of these approaches for Cloud-based monitoring.

### *5.7.1 Monitoring of QoS and SLAs*

Previous work on monitoring and adaptation of distributed heterogeneous systems is mainly concerned with establishing and monitoring SLAs and QoS policies. SLAs are typically composed of a number of Service Level Objectives (SLOs) [36] which correspond to the monitoring metrics, denoted facts, in our approach. The work by Comuzzi et al. [40] discusses a holistic SLA management approach. Whereas their work is strongly focused at the process for SLA establishment, we assume that the SLAs and the corresponding SLO metrics are known to the service provider. The MONINA language then facilitates the definition of raw facts (emitted by monitoring agents) and complex or derived facts (resulting from monitoring queries) to monitor the values of these SLOs. One of the core issues in service computing (and more recently Cloud computing) is the efficient generation of adaptation policies. The approach by Jung et al. [85] generates adaptation policies for multi-tier applications in consolidated server environments. The authors argue that online adaptation approaches based on, both, control theory and rule-based expert systems, have disadvantages. Hence, a hybrid approach is proposed which combines the best of both worlds. Their approach builds on queuing theoretic models for predicting system behavior, in order to automatically generate optimal system configurations. Our approach, on the other hand, abstracts from the type of monitoring data (whether predicted or actual values are used), and focuses on efficient definition and deployment of monitoring infrastructures. The work by Cardellini et al. [32] targets QoS-driven runtime adaptation of service oriented architectures. The presented approach does not, however, consider the efficient placement of the monitoring and adaptation rules themselves, but relies on decent initial placement or intervention by the operator.

Our work contributes an integrated approach which allows high-level definition of application topologies, which are then mapped to infrastructure graphs and deployed in Cloud environments.

### 5.7.2 Optimized Deployment of Monitoring Queries

The performance of monitoring infrastructures depends on the topology and data flow between query operators, hence efficient operator placement plays a key role. The work by Lakshmanan et al. [91] provides an overview of eight different operator placement algorithms, which are evaluated with respect to five core dimensions: node location (clustered/distributed), data rates (bursty/uniform), administrative domain (single/multiple), topology changes (dynamic/uniform), and queries (redundant/heterogeneous). Algorithms for efficient operator placement in widely-distributed systems are presented in [2]. Also the work by Pietzuch et al. [126] has influenced our work. Their approach performs operator placement using a stream-based overlay network for decentralized optimization decisions. A decentralized algorithm for near optimum operator placement in heterogeneous CEP systems is presented in [142]. The algorithm in [160] models the system load as a time series $X$ and computes the load correlation coefficient $\rho_{ij}$ for pairs of nodes $i$ and $j$. The optimization goal is to maximize the overall correlation, which has the effect that the load variance of the system is minimized. A comprehensive and fine-grained model of CPU capacity and system load is provided in [161]. The *feasible set* of stream data rates under a certain placement plan is constructed. Mathematically, the feasible set corresponds to the (nonnegative) space under *n node hyperplanes*, where $n$ is the number of nodes and the $i$-th hyperplane consists of all points that render node $i$ fully loaded.

### 5.7.3 Adaptation Rules and Objectives

The approach in [100] achieves optimization and adaptation of service compositions, which can arguably also be applied to the monitoring topology deployed in our approach. In contrast to [100], which takes a cost-centric viewpoint, in this work we target fast reactions, timely adaptations, and low network overhead. Adaptation rules based on the ECA [4] scheme are a popular technique used to control systems. However, for some complex systems the enumeration of all conditions, e.g., all possible types of failures, is often impracticable. Also, the actions to recover the system can become too tedious to be specified manually. Automated planning allows to automatically compute plans on top of a knowledge base following predefined objectives, and helps to enable goal-driven management of computer systems [138, 140].

### 5.7.4 Dynamic Reconfiguration and Redeployment

Facilities for dynamic reconfiguration and redeployment of monitoring infrastructures is at the heart of our approach. Srivastava et al. [148] present an approach for minimizing network usage and managing resource consumption in data acquisition networks by moving query operators. An elastic approach for optimal CEP query operator placement using cloud computing techniques is presented in [66]. As part of an optimization algorithm, the approach achieves a tradeoff between load distribution, duplicate event buffering and inter-node data traffic, also taking into account the

costs of migration. The work also tackles the technical challenge of migrating stateful operators between infrastructure nodes.

## 5.8 Summary

In this chapter we introduce an architecture and a domain-specific language that allow to integrate functionality provided by different components and to define monitoring and adaptation functionality. We assume that monitoring is carried out by complex-event processing queries, while adaptation is performed by condition action rules performed on top of a distributed knowledge base. However, our approach can be applied to other forms of control mechanisms with dependencies among functionality blocks. Furthermore, we discuss implementation characteristics of the currently realized prototype based on the proposed architecture.

In future work, we plan to integrate continuous deployment techniques, i.e., the capability to migrate elements at runtime to adapt according to more precise knowledge and changing environments. Furthermore, we aim to integrate the presented framework with current cloud management tools such as OpenStack Heat [119].

CHAPTER 6

# Non-intrusive Policy Optimization for Dependable and Adaptive Systems

The SOA paradigm facilitates the creation of distributed, composite applications. Services are usually simple to integrate, but often encapsulate complex and dynamic business logic with multiple variations and configurations. The fact that these services typically execute in a dynamic, unpredictable environment additionally complicates manageability and calls for adaptable management strategies. Current system control strategies mostly rely on static approaches, such as predefined policies. In this chapter we propose a novel technique to improve management policies for complex service-based systems during runtime. This allows systems to adapt to changing environments, to circumvent unforeseen events and errors, and to resolve incompatibilities of composed services. Our approach requires no knowledge about the internals of services or service platforms, but analyzes log output to realize adaptive policies in a non-intrusive and generic way. Experiments in our testbed show that the approach is highly effective in avoiding incompatibilities and reducing the impact of defects in service implementations.

## 6.1 Overview

The SOA [123] paradigm is a popular way to realize large-scale software systems integrating various services across multiple providers. A basic principle of SOA is that services are loosely coupled and implement business functionality as black boxes. Although services by definition are stateless and solely react to the provided input, service-based systems are generally depending on multiple configuration parameters and operate in dynamic, error-prone environments. All these characteristics render management of SOA systems a highly challenging task.

To ensure that such systems work properly, SOA governance is responsible for monitoring the performance of single services, runtime environments, and the overall system or service composition. This information is used either by human operators that reconfigure the system manually, or as input to policies that provide automated adaptation [147]. However, it is difficult to

anticipate each and every possible system state and provide an according policy rule. It is similarly problematic to foresee all effects of reconfiguration actions. Software bugs, for instance, can occur in every service implementation and runtime environment. Furthermore, the management policies or administrator actions may be in conflict, thus forcing the application into unwanted states or oscillation patterns [105]. In the end, human error can never be ruled out resulting in software defects [27], incorrect policies, or wrong adaptation actions, respectively.

We present a novel approach for dependable and adaptive control of service-oriented systems. Our technique poses low requirements on the system, and can be integrated into any SOA in a minimally invasive way. We propose a novel technique that infers a MDP model by analyzing the effect of system configuration parameters. The MDP model is automatically constructed from log output emitted by the services. The MDP is used to derive optimized policies considering software bugs, incompatibilities, and environmental changes. The main contributions of this chapter are (1) a generic framework for dependable and adaptive control of SOA based on log analysis, (2) novel techniques for transforming typical SOA log data into an MDP representation, and (3) real world experiments to quantify the usefulness of this approach.

## 6.2 Scenario

In this section, we introduce a scenario that serves as the basis for discussion of our approach. Consider a SOA that provides an enterprise travel itinerary service, allowing employees to automatically manage flight, car and hotel reservations for their trips to customers. The application is implemented as a composition of multiple interacting services.

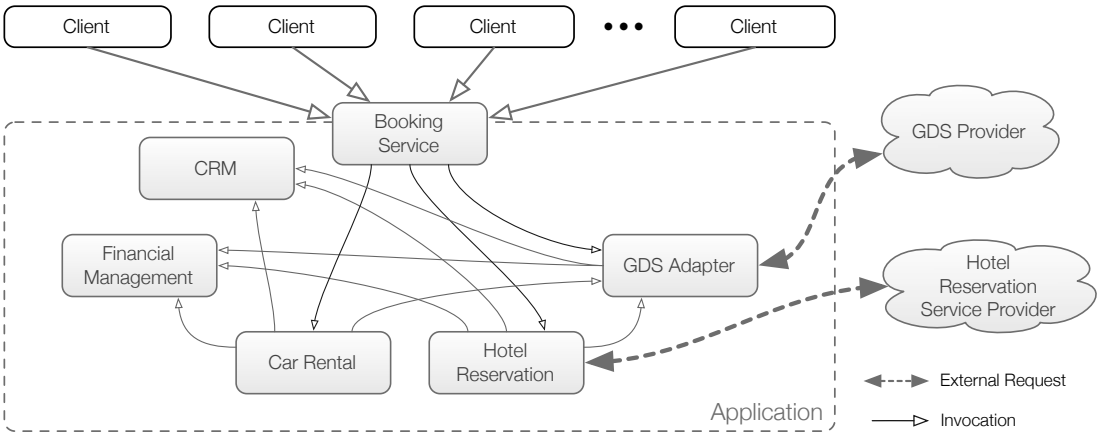



Figure 6.1: Architecture of the Travel Itinerary Application

A high-level overview of the services and interactions is shown in Figure 6.1. When an employee requests a new trip, the Global Distribution System (GDS) is requested to book a flight. To that end, the GDS adapter retrieves the customer address from the Customer Relationship Management (CRM) system, and uses external GDS Providers to find a suitable flight to an airport sufficiently close to the customer's site. At the same time the GDS needs to consider the available

budget, as retrieved from the Financial Management System (FMS). Next, the Hotel Reservation Service (HRS) contacts external HRS Providers to find a hotel close to the destination address. The Car Reservation Service (CRS) then issues a request to book a car for the specified period.

| **Parameter Type** | **Examples** |
|---|---|
| Application-Specific | Skip Budget Check |
| Dynamic Binding | Providers for GDS, HRS |
| QoS Criteria | Max. Response Time of GDS |
| Runtime Environment | Resource Limits |

Table 6.1: Parameters Defining the Travel Itinerary System Configuration

The system has various parameters that determine the current configuration state and influence the performance and functionality of the offered service (summarized in Table 6.1). Firstly, services expose application-specific properties that can be set explicitly (e.g., whether to skip a time-consuming budget check). Next, the composition depends on external services that are dynamically looked up and bound to at runtime (e.g., providers for GDS and HRS). Moreover, QoS properties are used to control the composition behavior (e.g., use only GDS providers with a certain response time). Finally, the runtime environment in which services execute influences the composition behavior (e.g., resource limits, request queue lengths). For reliable operation of a dependable system it is crucial to capture these parameters and to define management policies which aim at reconfiguring the application in response to undesired system states.

## 6.3 Adaptive Policy Optimization

In this section we present our approach for deriving optimized policies that lead to dependable and adaptive service-oriented systems. Figure 6.2 illustrates the assumptions we are posing on the system. Since the internal structure of the system is not important for our approach a generic SOA model can be assumed, consisting of a number of runtime environments, each hosting a number of services. We suppose that the system initially is either managed by an administrator or predefined policies, which are to be replaced by an optimized policy. The crucial prerequisite is that status information and management actions are observable, as illustrated by the magnifying glass.

We argue that the entity initially managing the system needs data about the system status and performed management actions anyways, hence such information is already available in some form. We do not assume that this information is produced centrally. Our approach is also applicable if each service and runtime environment produces the required data about status and reconfigurations individually. The status data needs to contain all information relevant to the system's performance and reliability. If, for instance, the response time of services matters then the status data should contain that information. However, there is no need for semantic annotations. Status information may be issued either at fixed intervals or triggered by events. It is sufficient, but not necessary, to emit only the indicators that have changed since the last output.

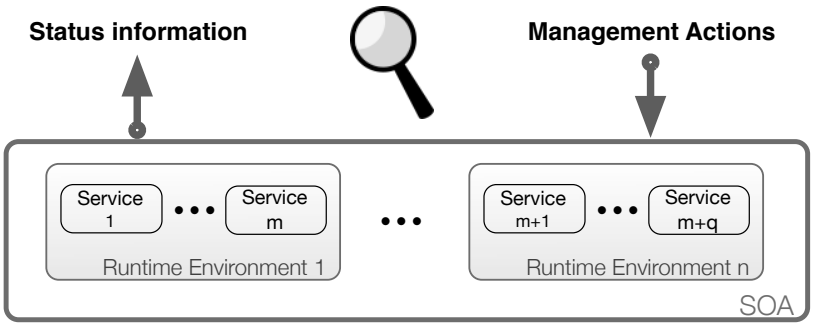



Figure 6.2: SBS without optimized policy. Information about the system status and management actions need to be observable in order to apply our approach.

At least if a performance indicator changes significantly, then there should be a status information update. We argue that all these requirements are met by virtually any SOA, which typically log the change of performance indicators and management actions.

In a nutshell, based on raw log data containing information about status updates of individual or compound components and management actions taken by the managing entity, we derive an optimized policy that takes over control, as shown in Figure 6.3.

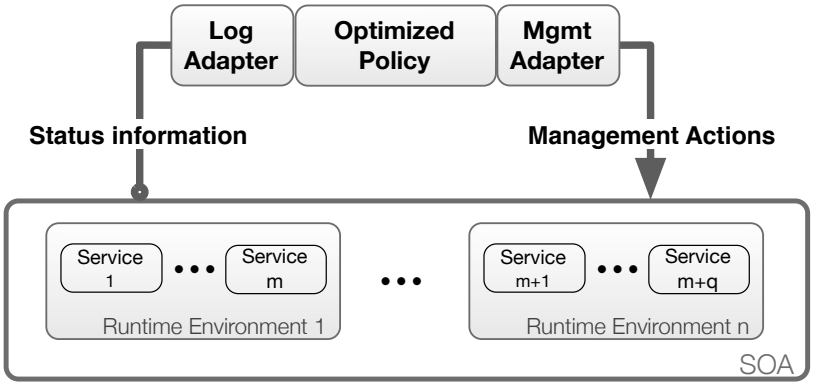



Figure 6.3: SOA with optimized policy, which takes system status data as input and outputs an optimized management action.

To be able to cover all kinds of systems with different models for providing status information and triggering management actions, we use a log adapter transforming status information into a canonical format and a management adapter responsible for executing a decision generated by the optimized policy.

Figure 6.4 shows the procedure of generating an optimized policy. Log data, containing information about relevant performance indicators as well as executed management actions, is

used as input to the log adapter, which transforms the information into a canonical representation. The log adapter is also used during runtime to preprocess the monitoring data for the optimized policy. The MDP Creator generates an MDP based on the canonical log data. Since MDPs are well known in Artificial Intelligence (AI) research, the Policy Creator can be based on already existing algorithms to finally create the optimized policy.

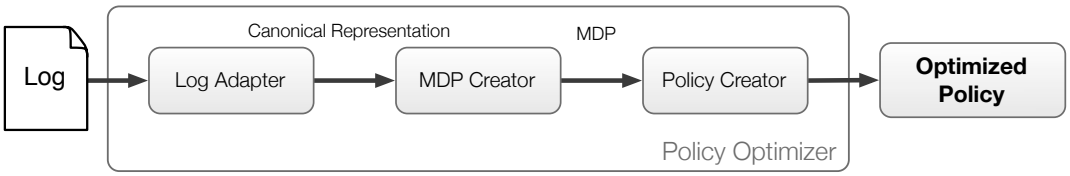



Figure 6.4: Functionality of the Policy Optimizer. It takes system information – usually in the form of logs – as input and outputs an optimized policy.

To the best of our knowledge there is no comparable approach transforming generic log data into an MDP. This transformation is not trivial since, for instance, there is no reward function contained in the log data, which, however, is an essential part of an MDP. The final optimized policy will mimic the successful management actions while avoiding the ones that lead to errors. It is able to absorb complex relationships, and root causes to effects that are not directly linked. In the following, we explain the three main components of the policy optimizer.

### *6.3.1 Log Adapter*

The log contains information about changes of performance indicators and management actions. However, we cannot assume, that logs from different services or runtime environments adhere to a common standardized format. The log adapter is responsible for the aggregation of all available logs from the system's components, and transforms the applications-specific formats into a canonical representation, as indicated by the arrow labeled ① in Figure 6.5. Apart from the Management Adapter, which provides a mapping from abstract management actions to concrete calls of components, the implementation of the log adapter is the only functionality that needs to be provided by the user in order to apply our approach. All other parts are generic and provided by our framework.

We propose the model shown in Figure 6.5 as canonical format for capturing relevant information contained in the log data. *StatusInformation* emitted from system components includes updated performance indicators, captured in *Parameter*, consisting of a generic name-value tuple.

Errors are identified as a special type of performance indicator in the log entries, and an according *Parameter* named '*ERR*' is set. When a configuration change is performed – either by the administrator or by the currently active management policy – the system emits a log message that is captured by *ManagementAction* in the model.

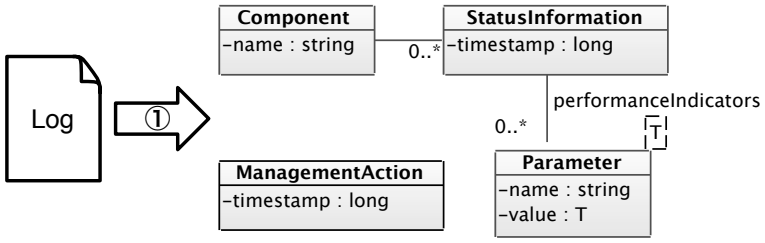



Figure 6.5: Functionality of the Log Adapter

### *6.3.2 MDP Creator*

In this section we show how the canonically represented log data are transformed into an MDP $(S, A, T, R)$.

**MDP States and Actions**

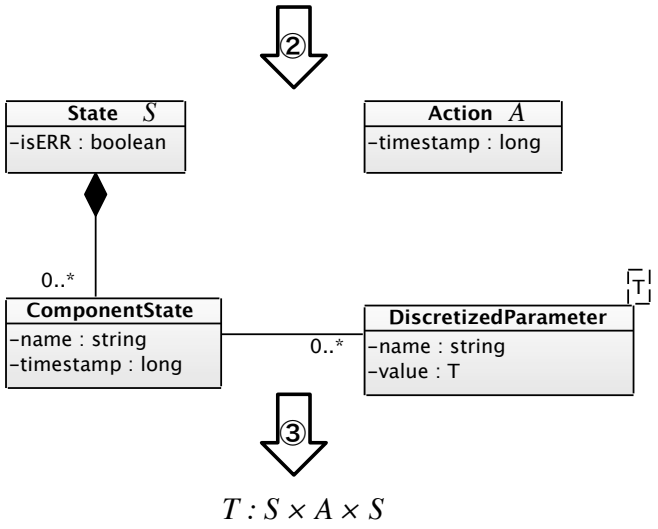



Figure 6.6: Generation of states, actions and transition model

The first step towards a complete representation as an MDP is the extraction of states $S$ and actions $A$ as shown by the arrow labeled ② in Figure 6.6. The captured *StatusInformation* updates for each component are aggregated to *ComponentState*s, and further to a full *State* representation for the system. This aggregation is based on either temporal proximity of the *timestamp*s or on a correlation by a specified *Parameter* attribute, such as request id. If a state contains a component state with at least one *DiscretizedParameter* signifying an error, the state is labeled as error state by setting *isERR*. States $S$ of the MDP must be finite. *Parameter*s, however, contain continuous values and could result in an infinite number of states. Therefore, we conduct an automatic

discretization of the observed values for each continuous *Parameter* (e.g., response-time), using a simple equal width interval method [44], resulting in *DiscretizedParameter*s. Our framework is designed to allow for the usage of different discretization methods, such as equal frequency binning or Holte's 1R [62], but we found that the simple equal width binning method performed reasonably well. Actions $A$ are constructed from each *ManagementAction* element. Additionally, a no-operation action (*NOP*) is added to the set of actions to allow the policy to remain in the current state, which allows to cover environment changes. Whenever performance indicators change without interference of policy or administrator action, the *NOP* action is used to represent external events not within control of our framework.

### MDP Transition Model

In this step, we extract the transition model $T : S \times A \times S$ from the representation generated so far, as shown by the arrow labeled ③ in Figure 6.6. The transition model $T(s,a,s')$ assigns the probability of reaching state $s'$ from state $s$ when performing action $a$. We derive the transition model by employing a modified Passive ADP Agent [135] algorithm. Algorithm 2 is invoked for each *State* and *Action* in chronological order, incrementally constructing a transition model from the observed data.

**Algorithm 2** Transition model learning

**Input:** $s'$: current state;
$a$: previously taken action

| | |
|---|---|
| $T(s,a,s')$ | a transition model |
| $N_{sa}$ | a table of frequencies for the state-action pairs, initially zero |
| $N_{s'\|sa}$ | a table of outcome frequencies given state-action pairs, initially zero |

1: **if** $s$ is not null **then**
2: increment $N_{sa}(s,a)$ and $N_{s'|sa}(s',s,a)$
3: **for all** $t$ such that $N_{s'|sa}(t,s,a)$ is nonzero **do**
4: $T(s,a,s') \leftarrow N_{s'|sa}(t,s,a)/N_{sa}(s,a)$
5: **end for**
6: **end if**

### MDP Reward Function

To complete the generation of the MDP $(S,A,T,R)$, we finally need to derive a reward function $R : S \to \mathbb{R}$ from the model. So far, the required data was in some way directly extractable from the log output. The reward function, however, is not readily available, as neither the logs, nor the models generated so far provide any reward signals.

We propose a novel approach to finding a reward function from preprocessed log data, based on the assumption that a majority of the actions taken by the initial managing entity are reasonable.

The basic idea is that if the initial manager performs some action $a$ in state $s_1$ which leads to state $s_2$, i.e., $s_1 \xrightarrow{a} s_2$, then we assume that state $s_2$ is more desirable than state $s_1$. In the case of contradictory transitions where there are a number of transitions $s_1 \xrightarrow{\bullet} s_2$ and $s_2 \xrightarrow{\bullet} s_1$ we assume that the majority of management actions was beneficial. In any case, failure conditions are to be avoided. Figure 6.7 graphically illustrates how we generate a reward function $R$ that

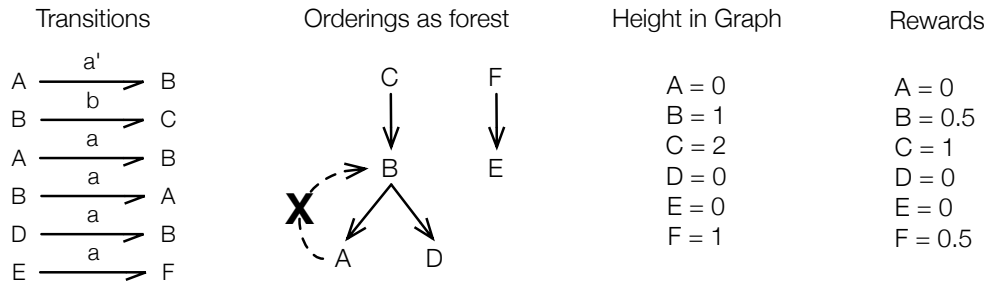



Figure 6.7: Illustrative example of our reward function

assigns rewards to states. The input is observed transitions $T^O \subseteq S \times A \times S$. As a first step a forest $F = (S,E)$ is created, where the nodes consist of states $S$ and edges $E$. Furthermore, the following condition holds: $(s_1,s_2) \in E \implies |\{s_1 \xrightarrow{\bullet} s_2\}| \leq |\{s_2 \xrightarrow{\bullet} s_1\}|$. The reward function $R : S \to [-1,1]$, which maps states to rewards, is defined as follows

$$R(s) = \begin{cases} \frac{height_F(s)}{height(subtree(s))}, & \text{if } s \neq ERR \\ -1, & \text{if } s = ERR \\ 0, & \text{otherwise,} \end{cases} \tag{6.1}$$

where $height_F(s)$ returns the length of the longest path from node $s$ to a leaf node in $F$, $subtree(s)$ return the tree $s$ belongs to, and $height(t)$ returns the height of tree $t$, i.e., the height of its root node.

The definition ensures that failure states give lower (negative) rewards than all non-failure states. Non-failure states give a higher reward if they have been favored by the initially managing entity. For a state without any occurrence in the log, 0 reward is given.

### *6.3.3 Policy Improvement*

The output of the MDP creator is a system description in the form of an MDP. In the policy improvement step, well-known decision making algorithms can be applied to optimize the management policy. We have incorporated both Policy Iteration [20] for adapting a policy to avoid error states in an environment, that is not expected to change significantly, as well as the Q-learning [157] algorithm, able to iteratively adjust to changing environments.

Finally, to utilize the optimized policy, it is deployed, replacing the initial policy. The policy optimization can be arbitrarily repeated. This allows to take additionally gathered data into account to refine the policy and react to changes in the environment.

## 6.4 Evaluation

To evaluate our approach, we have created a simulation testbed, allowing for quick and easy specification of complex composite applications, their runtime properties, as well as the initial management policies. The simulation testbed is implemented using Ruby[1]. It provides a DSL for the definition of the service behavior, e.g., interaction with other services, and processing cost. Furthermore, it allows for the specification of configuration variants and parameters that can be dynamically changed during runtime. The listing in Figure 6.8 shows a simplified definition of the HRS's "find hotel" method in the "search using all external providers" configuration. It adds a new configuration variant to the HRS's "find hotel" method, which invokes 3 partner services CRM, FMS, and GDS. Furthermore, to model processing time and interaction with external services, the 'cost' value defines the mean of the normally distributed invocation time.

```
add("hrs#find_hotel").add_config("all#regular") do
  invoke "crm#get_customer_address"
  invoke "fm#get_budget"
  invoke "gds#get_flight_information"
  cost 4 # estimated cost of external requests
end
```

Figure 6.8: Service Method Definition: HRS "find hotel", demonstrating basic capabilities of the developed simulation testbed, i.e., invocation of other services and their methods, simulation of external processing costs, and the definition of configuration variants.

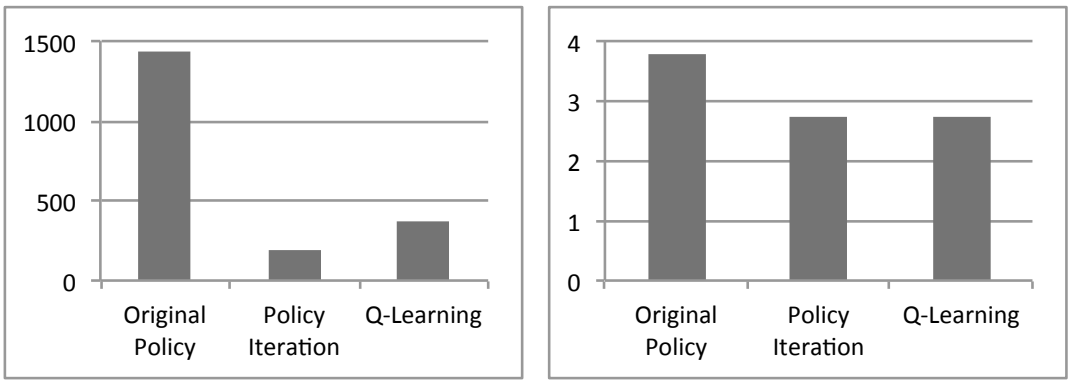
The simulation testbed furthermore allows for the specification of initial management policies for the created services using a similar DSL. Additionally, a management interface is provided, allowing to replace the initial policy with the optimized one. We also provide for several different user interaction patterns to simulate varying numbers of users with different behaviors.

The policy optimization framework is implemented as a Java library. For convenience, we provide an exemplary log adapter[2], compatible with the simulation testbed log, and conforming to the specification. The optimization algorithms, i.e., policy iteration and Q-Learning, are optimized implementations of the algorithms presented in [135].

We implemented the scenario application as described in Section 6.2, with multiple concurrent clients sending requests to the Booking Service. The initial policy is designed to degrade the quality of services for faster processing times if load rises above a certain threshold. In the interaction of the HRS and GDS services there manifests a hidden incompatibility in one certain configuration constellation. In that scenario, the HRS is configured to only consider major airports for finding hotels which increases the probability of empty results for the travel itinerary in combination with the GDS, configured to look for the cheapest flights, which might use smaller airports. This situation is perceived as an error and mitigated by the HRS querying all available

[1] http://ruby-lang.org/

[2] Exemplary log4j configuration directives along with helper methods conforming to the implemented format are available at https://gist.github.com/1197839




(a) Total Errors (b) Processing Time (s)

Figure 6.9: Evaluation result summary. Our approach was able to significantly improve the initial management policy using both, policy iteration and Q-Learning. Error occurrence was reduced by more than 70%, and average processing time decreased by over 27%.

partner services incurring additional processing overhead. This special case is not addressed in the otherwise useful initial policy. In general, as argued before, it is very difficult to anticipate all possible failure scenarios, which calls for adaptive management policies as proposed here.

The policy optimization is performed after a bootstrapping phase which is needed to collect log data. This phase is completed if 3000 requests have been processed. Each single service invocation triggers the output of at least one status update information. The requests are issued in a sawtooth pattern to simulate varying load patterns. The evaluation period, in which the performance of policies is assessed, consists of an equal amount of requests following the same pattern. The evaluation was performed on a machine with a 2.4GHz quad-core Intel Xeon E5620 CPU with 12MB shared L3 cache, 16GB RAM, running Ubuntu 10.04 LTS.

The results in Figure 6.9 show, that we are able to reduce the occurrence of errors by over 70% using the Q-Learning policy improvement, and by more than 80% using the policy obtained through policy iteration. Furthermore, the average request processing time is reduced by over 27% due to the reduced impact of the performance penalty incurred when errors are encountered. The worse performance of the Q-Learning algorithm with regard to total errors can be attributed to the slower convergence of this algorithm for the given problem. However, Q-Learning is more suitable for iterative, online policy improvement.

Figure 6.10 presents detailed evaluation results. The section ‘Queue Length’ shows the system’s processing queue and illustrates that the chosen request pattern induces significant stress on the application. The three ‘Processing Time’ sections show the time it took the system to process each single request. The optimized policies maintain appropriate processing times, and allow for a degradation in processing time to avoid errors. The last section, ‘Cumulative Errors’, shows the aggregated number of errors encountered during the evaluation. In our scenario, both optimized policies outperform the original management strategy.

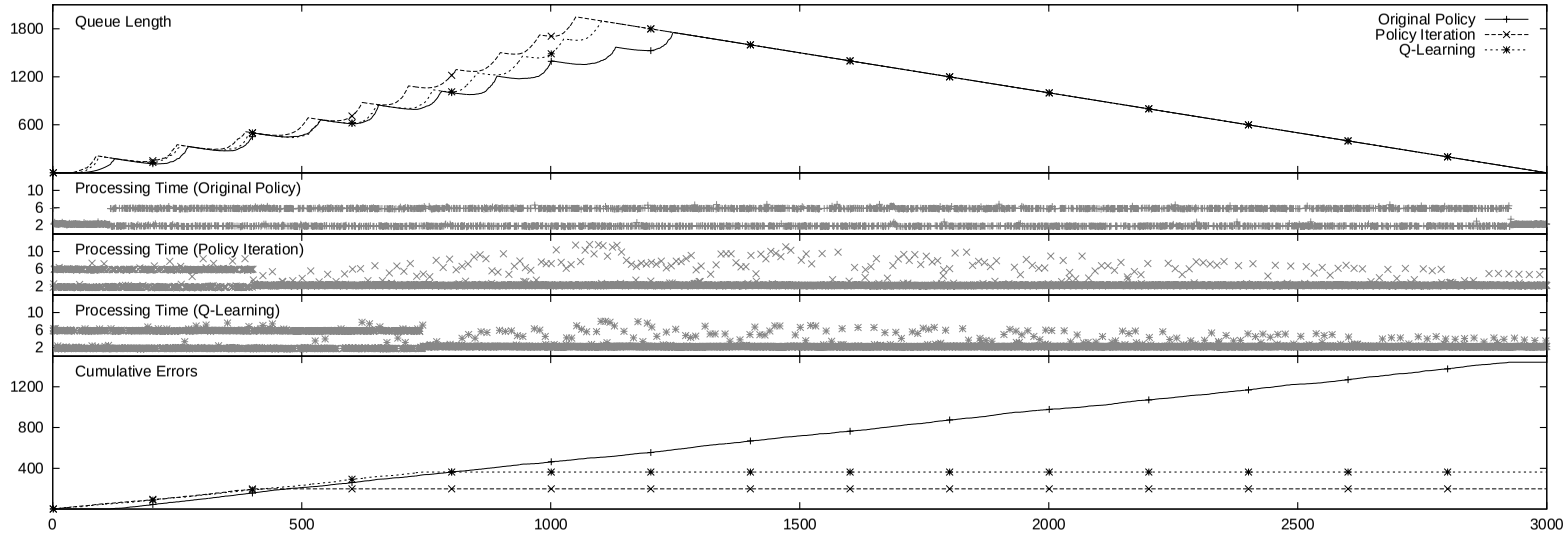



Figure 6.10: Results of the conducted experiment. The x-axis represents the progress of the evaluation based on the number of request processed. ‘Queue Length’ shows the number of requests pending for processing. The three ‘Processing Time’ panels depict the time it took for individual requests to be processed for each of the three evaluated policies. ‘Cumulative Errors’ shows the aggregated number of errors occurred during the evaluation run.

## 6.5 Related Work

Autonomic and policy-based management, fault tolerance, and self-healing systems research has received a lot of attention in the past and continues to do so until today. Recently, these areas are becoming more and more relevant for SOA environments, as unaided optimization of configuration, collaboration, and error mitigation strategies, are essential for the successful implementation of loosely-coupled SBSs.

An approach to autonomic SLA-based management of distributed systems is presented in [3], proposing a hierarchical architecture of autonomic managers using a traditional MAPE cycle, each responsible for certain non-functional concerns of an application according to a predefined policy. Similarly, [96] presents an autonomic framework for preventing SLA violations. The approach presented in [140] applies automated planning algorithms for system reconfiguration based on user-defined objectives.

Several approaches have been presented in the area of self-healing web service integration and composition (e.g., [33, 41, 42]), as well as self-healing BPEL processes (e.g., [14, 15, 113]). The presented techniques are concerned with optimizing the behavior of integrated and/or composed services and business processes, using static a priori policies, and, contrary to our approach, assume in-depth knowledge of the services to be managed.

An architecture for self-manageable cloud services is presented in [26]. Similar to our approach, services provide management interfaces to allow for the control by the autonomic manager. However, the presented solution requires for the autonomic manager to know the service capabilities ahead of time.

A notable method for policy-driven autonomic management using reinforcement learning techniques is presented in [11, 12]. The approach allows for optimization of runtime behavior of managed applications by analyzing and deconstructing the provided management policies, utilizing a complex management architecture. In contrast to our approach, managed applications and components must be completely controlled using the proposed framework, policies enforced by internal mechanisms cannot be taken into account.

## 6.6 Summary

In this chapter we present a novel approach for optimizing control policies for SOA, leading to dependable and adaptive service-oriented systems. The approach makes minimal assumptions about the structure and capabilities of the system. We present a new technique to transform log data into a Markov Decision Process representation, which is used to generate an improved control policy that takes into account dynamics of the environment and software defects. This is done at runtime without need for human intervention. Experiments conducted in a testbed consisting of real Web services show that the adaptive policies are capable of mitigating the effects of defects and incompatibilities between collaborating components.

As future work we plan to integrate service level objectives, as well as request payload data, into the policy generation in addition to log data. We will also investigate how our approach can be applied to Web service compositions and business process optimization. Another future research direction includes to consider not only the service level but also the resource level, i.e.,

to control the mapping of services to resources. The techniques presented in this paper allow to manage relatively complex SBAs. However, if large-scale highly complex systems are to be controlled, algorithms for complexity reduction, such as principal component analysis, could be employed. Active learning promises better exploration and faster learning rates for Q-learning.

CHAPTER 7

# Identifying Incompatible Service Implementations

We study fault localization techniques for identification of incompatible configurations and implementations in SBAs. Practice has shown that standardized interfaces alone do not guarantee compatibility of services originating from different partners. Hence, dynamic runtime instantiations of such SBAs pose a great challenge to reliability and dependability. The aim of this work is to monitor and analyze successful and faulty executions in SBAs, in order to detect incompatible configurations at runtime. We propose an approach using pooled decision trees for localization of faulty service parameter and binding configurations, explicitly addressing transient and changing fault conditions. The presented fault localization technique works on a per-request basis and is able to take individual service inputs into account. Considering not only the service configuration but also the service input data as parameters for the fault localization algorithm increases the computational complexity by an order of magnitude. Hence, our performance evaluation is targeted at large-scale SBAs and illustrates the feasibility and decent scalability of the approach.

## 7.1 Introduction

Distributed and mission-critical enterprise applications are becoming more and more reliant on external services, provided by suppliers, customers or other members of Service Value Networks (SVNs) [23]. In many industries, the technical interfaces of these services are governed by industry standards, specified by bodies such as the TMF, the Association for Retail Technology Standards (ARTS) or the International Air Transport Association (IATA). Hence, integration of services provided by different partners into a single SBA becomes feasible. Additionally, as oftentimes a multitude of potential partners are providing implementations of the same standardized interfaces, SBAs are enabled to dynamically switch providers at runtime, i.e., select the most suitable implementation of a given standardized interface based on current requirements.

Unfortunately, practice has shown that standardized interfaces alone do not guarantee compatibility of services originating from different partners. Many industry standards are prone to underspecification, while others simply allow multiple alternative (and incompatible) implementations to co-exist. Additionally, and particularly for younger specifications, not every vendor can be trusted to interpret each standard text in the same way. Consequently, there are practical cases where SBAs, which should work correctly in theory, fail to function because of unexpected incompatibilities of service implementations chosen at runtime. Note that this does not necessarily mean that any single one of the chosen service implementations is faulty in itself – it merely means that two or more chosen service implementations do not work in conjunction (even though both may work perfectly in combination with other services).

In this chapter, we present a machine learning driven approach to identify such incompatibilities of industry standard implementations. We analyze runtime event logs emitted by the SBA using decision tree techniques and principal component analysis, with the goal of suggesting combinations of service implementations that should not be used in conjunction. Decision trees are a white-box machine learning approach that allow to extract incompatibility rules from the constructed tree [144]. Our approach takes into account not only the actual service implementations themselves, but also the received input and the produced output data of implementations. Furthermore, we quantify the benefits of our approach based on a numerical evaluation.

## 7.2 Scenario

We base the discussion of our approach on the scenario introduced in Section 3.2, and elaborate relevant parts in more detail. As introduced in Section 3.2, the eTOM [154] is a widely adopted industry standard for implementation of business processes promoted by the TMF, and our scenario is condensed from the TMF's Case Study Handbook [153] as well as two eTOM-related IBM publications on practical application of SOA in such systems [52, 58].

### *7.2.1 Service Delivery the eTOM Way*

Figure 7.1 depicts the service delivery process in BPMN. The process consists of six activities (denoted $i_1, \ldots, i_6$). We refer to these activities as *interfaces* or *abstract services*. Each abstract service activity has a set of sub-activities which we denote as *concrete service implementations* (denoted $c_1, \ldots, c_{14}$ in the figure). At runtime the process selects and executes one concrete service for each service interface. The data flow between the service interfaces of the scenario process is illustrated in Figure 7.2.

The process is initiated by the abstract service $i_1$ (Handle Customer Order) which is offered in two variants for standard and premium users. Depending on the order input, the process then configures a particular service (ADSL, IPTV or VoIP). The third abstract service selects one among three partner providers to allocate resources required for service delivery. Telecommunication services are typically associated with QoS attributes, which are fine-tuned by abstract service $i_4$. For instance, this activity configures parameters in the ADSL device or sets the location Uniform Resource Identifier (URI) of IPTV endpoints, in correspondence with QoS requirements. If a

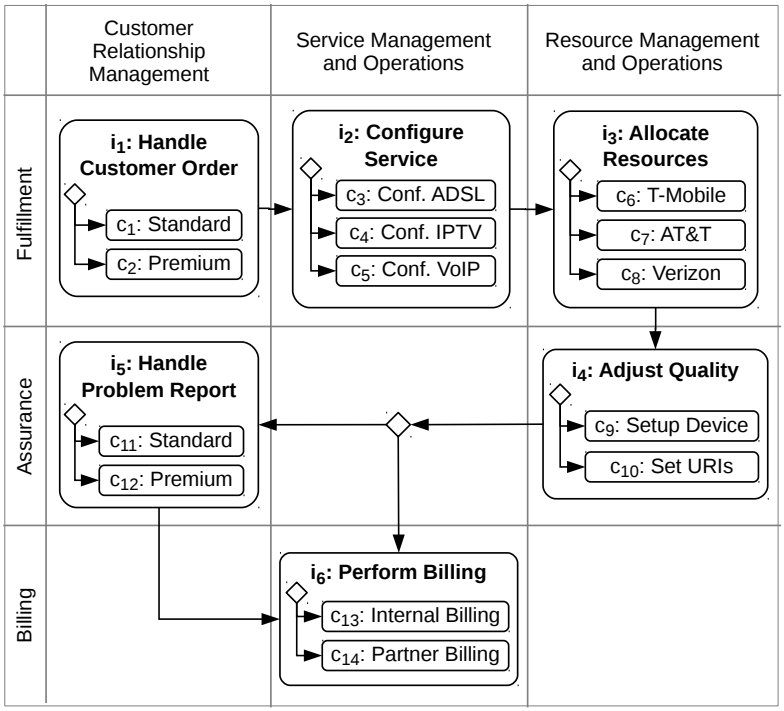



Figure 7.1: Service-Based Scenario Application Architecture, based on [52] and [58]

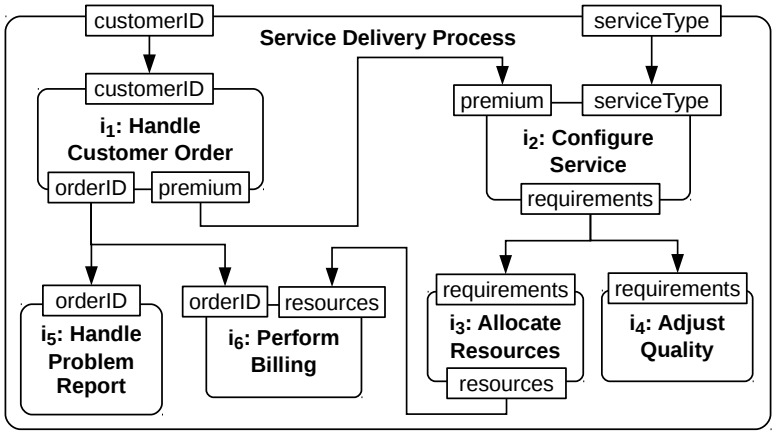



Figure 7.2: Data Flow in the Scenario Process

problem is detected at runtime, the optional reporting service is executed in activity $i_5$. Finally, the process terminates after storing billing information, either for paying partner providers or for internal accounting if the service was delivered in-house. Besides regular termination, the process may also be interrupted by exceptions at any stage of execution (not depicted in Figure 7.1). We assume that the information whether the execution has terminated regularly or exceptionally is available for each instance of the process.

One defining characteristic of eTOM and the presented scenario is process decomposition, which means that business processes are modeled at different levels of abstraction, from the high-level business goals view down to the technical implementation level. In our scenario this is illustrated by the distinction between abstract and concrete services, though in fact the number of abstraction levels can be higher than two.

### *7.2.2 Challenges for Reliable Service Delivery*

The scenario outlined in Section 7.2 entails challenges to reliability that are typically encountered in service-based applications. Interface standardization (such as Multi-Technology Operations System Interface (MTOSI) in the case of our scenario) per se does not guarantee compatibility of services originating from different partners. The interactions among services contain complex dependencies and data flows. The number of variations, i.e., possible instantiations of the process, grows exponentially with the combination of concrete services as well as the provided user input. Hence, comprehensive upfront verification and validation in terms of integration testing is not always feasible and can only cover a certain percentage of the possible instantiations. Therefore, in addition to rigorous testing methods, reliable operation of business-critical SBAs requires proactive monitoring to analyze and avoid incompatible configurations at runtime.

## 7.3 Fault Localization Approach

This section discusses our novel fault localization technique. Section 7.3.1 establishes a notion for the model of SBAs. Sections 7.3.2 and 7.3.3 discuss preprocessing and machine learning techniques used to learn rules which describe the reasons for faults based on the collected model data.

### *7.3.1 System Model*

We establish a generalized model that forms the basis for the concepts presented in the work. The core model artifacts are summarized in Table 7.1 and briefly discussed in the following.

A SBA consists of a set of industry standard service interfaces $I = \{i_1, \ldots, i_n\}$ and a set of implementations $C = \{c_1, \ldots, c_m\}$. The mapping between interface and implementation is defined by the function $c : I \to \mathscr{P}(C)$, where $\mathscr{P}(C)$ denotes the power set of $C$. The domain of possible input parameters $P$, each defined by name ($N$) and domain of possible data values ($D$) is represented by $P = N \times D$. Function $p : I \to \mathscr{P}(P)$ returns all inputs required by an interface, and $d : P \to D$ returns the value domain for a given parameter. The set $F \subseteq I \times I$ defines data flows as pairs of interfaces $(i_x, i_y)$, where the output of $i_x$ becomes the input of $i_y$. Transitive data flows spanning more than two services can be derived from $F$. Moreover, we define $T = \langle t_1, \ldots, t_k \rangle$

as the sequence of logged execution traces $t_x : K \to V$ in chronological order, mapping the set of keys $K = I \cup (I \times N)$ to values $V = C \cup D$; interfaces $I$ map to implementations $C$, whereas parameter names $I \times N$ map to parameter domains $D$. Finally, the function $r : \{1,\ldots,k\} \to \{success, fault\}$ is used to express the result of a trace $t_x, x \in \{1,\ldots,k\}$, i.e., whether the trace represents a successful or failed execution of the SBA.

| **Symbol** | **Description** |
|---|---|
| $I = \{i_1,...,i_n\}$ | Set of industry standard interfaces defined by the SBA. Example: $I = \{i_1,...,i_6\}$ |
| $C = \{c_1,...,c_m\}$ | Set of available concrete implementations to interfaces. Example: $C = \{c_1,...,c_{14}\}$ |
| $c : I \to \mathscr{P}(C)$ | Function that returns all concrete candidate implementations for an interface. Example: $c(i_2) = \{i_3, i_4, i_5\}$ |
| $P = [N \times D]$ | Domain of service input parameters. Each input parameter is defined by a name ($N$) and a domain of possible data values ($D$). Example: $P = \{('premium', \{true, false\}), ('serviceType', String), \ldots\}$ |
| $p : I \to \mathscr{P}(P)$ | Function that returns all input parameters for an interface. Example: $p(i_1) = \{('customerID', String)\}$ |
| $F \subseteq I \times I$ | Set of direct data flows (dependencies) between two services. Example: $F = \{(i_1,i_2),(i_2,i_3),(i_2,i_4),\ldots\}$ |
| $t_x : K \to V$, $K = I \cup (I \times N)$, $V = S \cup D$, $x \in \{1,\ldots,k\}$ | Log trace representing one execution of the SBA. The function maps from a set of keys ($K$) to values ($V$). In particular, interfaces ($I$) map to implementations ($S$), and parameter names ($I \times N$) map to parameter values ($D$). Example: $t_1$: $a_1 \mapsto c_2$, $i_2 \mapsto c_3$, ..., $(i_1, 'customerID') \mapsto 'joe123'$, $(i_2, 'premium') \mapsto true$, ... |
| $T = \langle t_1,...,t_k \rangle$ | Sequence of logged execution traces. |
| $r : \{1,\ldots,k\} \mapsto \{success, fault\}$ | Function that determines for an integer $x \in \{1,\ldots,k\}$ whether the execution represented by the trace $t_x$ was successful or has failed. |
| $E_S \subseteq \mathscr{P}(\mathscr{P}(I \to C))$ | Incompatible assignment. If the implementations in $E$ are used in combination, a fault occurs at runtime. Example: $E_C = \{\{(i_1 \mapsto c_2)\}, \{(i_2 \mapsto c_4), (i_3 \mapsto c_8)\}\}$. |

*continued on next page*

*continued from previous page*

| Symbol | Description |
|---|---|
| $E_P \subseteq \mathscr{P}(\mathscr{P}($ $(I \to C) \cup$ $((I \times N) \mapsto D)))$ | Incompatible assignment with specific input data. Example: $E_P = \{\{(i_1 \mapsto c_2)\}, \{((i_2,'premium') \mapsto false), (i_3 \mapsto c_8)\}\}$ |

Table 7.1: Description of Variables

Summarizing the model, the core idea of our approach is to analyze log traces of SBA executions for fault localization. We consider two classes of properties as part of the traces: 1) runtime binding of interfaces to concrete implementations, 2) service input parameters, i.e., data provided by the user to the application as well as data flowing between services.

### *7.3.2 Trace Data Preparation*

Table 7.2 lists an excerpt of six exemplary traces for the scenario application. The table contains multiple rows which represent the traces ($t_1, \ldots, t_6$); the columns contain the bindings for the service interfaces ($i_1, i_2, i_3, \ldots$), the input parameter values ($t_x(\ldots)$), and the success result of the trace ($r(x)$). Two exemplary parameters for a customer service are in the Table: $t_x(i_1,'custID')$ denotes the customer identifier provided to some interface $i_1$, and the parameter $t_x(i_2,'premium')$ tells the service interface $i_2$ whether it is dealing with a regular customer or a high-paying premium customer.

We follow the typical machine learning terminology and denote the column titles as *attributes* and the rows starting from the second row as *instances*. The first attribute ($t_x$) is the instance identifier, and $r(x)$ is denoted *class attribute*.

| $\mathbf{t_x}$ | $\mathbf{i_1}$ | $\mathbf{i_2}$ | $\mathbf{i_3}$ | .. | $\mathbf{t_x(i_1,'custID')}$ | $\mathbf{t_x(i_2,'premium')}$ | .. | $\mathbf{r(x)}$ |
|---|---|---|---|---|---|---|---|---|
| $\mathbf{t_1}$ | $c_1$ | $c_3$ | $c_7$ | .. | $'joe123'$ | $false$ | .. | $success$ |
| $\mathbf{t_2}$ | $c_2$ | $c_4$ | $c_6$ | .. | $'aliceXY'$ | $true$ | .. | $success$ |
| $\mathbf{t_3}$ | $c_1$ | $c_5$ | $c_8$ | .. | $'joe123'$ | $false$ | .. | $fault$ |
| $\mathbf{t_4}$ | $c_2$ | $c_5$ | $c_8$ | .. | $'bob456'$ | $true$ | .. | $success$ |
| $\mathbf{t_5}$ | $c_2$ | $c_4$ | $c_7$ | .. | $'aliceXY'$ | $true$ | .. | $success$ |
| $\mathbf{t_6}$ | $c_1$ | $c_4$ | $c_8$ | .. | $'lindaABC'$ | $false$ | .. | $fault$ |
| .. | | | | | | | | |

Table 7.2: Example Traces for Scenario Application

The number of attributes and combinations of attribute values can grow very large. To estimate the number of possible traces for a medium sized application, consider an SBA using 10 interfaces ($|I| = 10$), 3 candidate implementations per interface ($|c(i_x)| = 3 \ \forall i_x \in I$), 3 input parameters per service ($|p(i_x)| = 3 \ \forall i_x \in I$), and 100 possible data values per parameters ($|d| = 100 \ \forall i_x \in I, (n,d) \in p(i_x)$). The total number of possible execution traces in this SBA is

$3^{10} * 100^{3^{10}} = 5.9049 * 10^{64}$. Efficient localization of faults in such large problem spaces evidently poses a huge algorithmic challenge. Even more problematically, the problem space becomes infinite if the service parameters use non-finite data domains (e.g., *String*). The first step towards feasible fault analysis is to reduce the problem space to the most relevant information. We propose a two-step approach:

1. Identifying (ir)relevant attributes: The first manual preprocessing step is to decide, based on domain knowledge about the SBA, which attributes are relevant for fault localization. For instance, in an e-commerce scenario we can assume that a unique customer identifier (*custID*) does not have a direct influence on whether the execution succeeds or fails. Per default, all attributes are deemed relevant, but removing part of the attributes from the execution traces helps to reduce the search space.

2. Partitioning of data domains: Research on software testing and dependability has shown that faults in programs are often not solely incurred by a single input value, but usually depend on a range of values with common characteristics [158]. Partition testing strategies therefore divide the domain of values into multiple sub-domains and treat all values within a sub-domain as equal. As a simple example, consider a service parameter with type *Integer* (i.e., $\{-2^{31}, \ldots, +2^{31}-1\}$), a valid partitioning would be to treat negative/positive values and zero as separate sub-domains: $\{\{-2^{31}, \ldots, -1\}, \{0\}, \{1, \ldots, +2^{31}-1\}\}$. If explicit knowledge about suitable partitioning is available, input value domains can be partitioned manually as part of the preprocessing. However, efficient methods have been proposed to automatize this procedure (e.g., [38]).

### *7.3.3 Learning Rules from Decision Trees*

Using the preprocessed trace data, we strive to identify the attribute values or combinations of attribute values that are likely responsible for faults in the application. For this purpose, we utilize decision trees [128], a popular technique in machine learning. It has the advantage that the decision making of the resulting trees can be easily comprehended; their knowledge can be distilled for the purpose of fault localization. Also, decision tree training with state of the art algorithms like C4.5 results in comparably fast learning speeds, compared to other machine learning approaches.

Figure 7.3 illustrates decision trees based on the example traces in Table 7.2. The figure shows two variants of the same tree which classifies non-premium services from Provider 3 ($t_x(i_3) = c_8$). The inner nodes are decision nodes which divide the traces search space, and the leaf nodes indicate the trace results. The left-hand side of the figure shows a regular decision tree where each decision node splits according to the possible values of an attribute. The right-hand side shows the same tree with binary split (i.e., each decision node has two outgoing edges).

The decision tree with binary split is used to automatically derive incompatible attribute values. The basic procedure is to loop over all $fault$ leaf nodes and create a combination of attribute assignments along the path from the leaf to the root node. The detailed algorithm is presented in Algorithm 3. For each $fault$ leaf node, a set $E_{temp}$ is constructed which contains the conditions that are true along the path. The total set of all such condition combinations is denoted $E_I$. Our approach exploits the simple structure of decision trees for extracting incompatibility rules; other

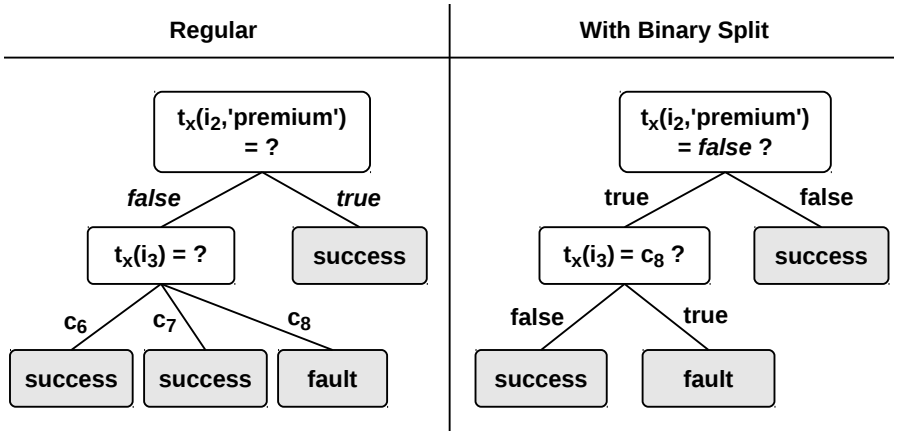



Figure 7.3: Exemplary Decision Tree in Two Variants

**Algorithm 3** Obtaining Incompatibility Rules from Decision Tree

```
1:  E_I ← ∅
2:  for all fault leaf nodes as n do
3:      path ← path of nodes from n to root node
4:      E_temp ← ∅
5:      for all decision node along path as d do
6:          c ← condition of d
7:          if c is true along path then
8:              E_temp ← E_temp ∪ c
9:          end if
10:     end for
11:     E_I ← E_I ∪ E_temp
12: end for
13: for all E_x, E_y ∈ E_I do
14:     if E_x is covered by E_y then
15:         E_I ← E_I \ E_x
16:     end if
17: end for
```

popular classification models (e.g., neural networks) have much more complex internal structures which make it harder to extract the principal attributes responsible for the output [144].

### *7.3.4 Coping with Transient Faults*

So far, we have shown how trace data can be collected, transformed into a decision tree, and used for obtaining rules which describe which configurations have led to a fault. The assumption so far was that faults are deterministic and static. However, in real-life systems which are influenced by various external factors, we have to be able to cope with temporary and changing faults. Our

approach is hence tailored to react to such irregularities in dynamically changing environments.

A temporary fault manifests itself in the log data as a trace $t \in T$ whose result $r(t)$ is supposed to be *success*, but the actual result is $r(t) = fault$. Such temporary faults can lead to a situation of contradicting instances in the data set. Two trace instances $t_1, t_2 \in T$ contradict each other if all attributes are equal except for the class attribute: $\{(k,v) \mid (k,v) \in t_1\} = \{(k,v) \mid (k,v) \in t_2\}$, $r(t_1) \neq r(t_2)$.

Fortunately, state-of-the-art decision tree induction algorithms are able to cope with such temporary faults which are considered as noise in the training data (e.g., [1]). If the reasons for faults within an SBA change permanently, we need a mechanism to let the machine learning algorithms forget old traces and train new decision trees based on fresh data. Before discussing strategies for maintaining multiple decision trees, we first briefly elaborate on how the accuracy of an existing classification model is tested over time.

### 7.3.5 Assessing the Accuracy of Decision Trees

Let $D$ be the set of decision trees used for obtaining fault combination rules. We use the function $rc : (D \times \{1,\ldots,k\}) \rightarrow \{success, fault\}$, where $k$ is the highest trace index, to express how a decision tree classifies a certain trace. Over a subset $T_d \subseteq T$ of the traces classified by a decision tree $d$, we assess its accuracy using established measures true positives ($TP$), true negatives ($TN$), false positives ($FP$), and false negatives ($FN$) [10]:

- True Positives: $TP(T_d) = \{t_x \in T_d \mid rc(d,x) = fault \wedge r(x) = fault\}$
- True Negatives: $TN(T_d) = \{t_x \in T_d \mid rc(d,x) = success \wedge r(x) = success\}$
- False Positives: $FP(T_d) = \{t_x \in T_d \mid rc(d,x) = fault \wedge r(x) = success\}$
- False Negatives: $FN(T_d) = \{t_x \in T_d \mid rc(d,x) = success \wedge r(x) = fault\}$

From the four basic measures we obtain further metrics to assess the quality of a decision tree. The *precision* expresses how many of the traces identified as faults were actually faults ($TP/(TP+FP)$). *Recall* expresses how many of the faults were actually identified as such ($TP/(TP+FN)$). Finally, the *F1 score* [64] integrates precision and recall into a single value (harmonic mean):

$$F1 = 2 * \frac{precision \cdot recall}{precision + recall}$$

### 7.3.6 Maintaining a Pool of Decision Trees

In the following, we discuss our approach to cope with changing fault conditions over time, based on a sample execution of the system model introduced in Section 7.3.1.

Figure 7.4 illustrates a representative sequence of execution traces $\{t_1, t_2, t_3, \ldots\}$; time progresses from the left-hand side to the right-hand side of the figure. In the top of the figure the trace results $r(t_x)$ are printed, where "S" represents *success* and "F" represents $fault$. As the traces arrive with progressing time we utilize deduction algorithms to learn decision trees from

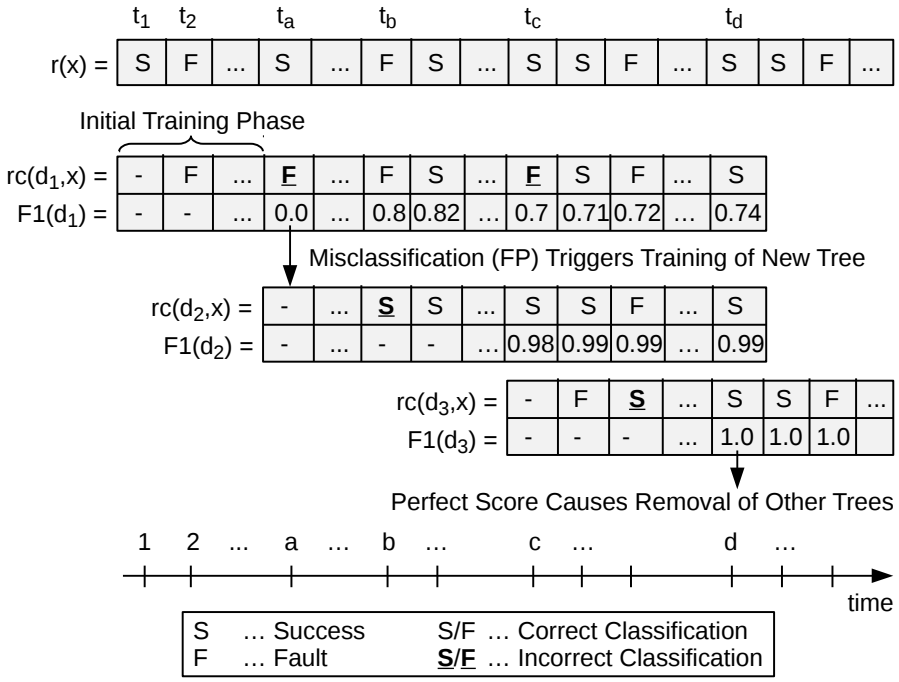



Figure 7.4: Maintaining Multiple Trees to Cope with Changing Faults

the data. At time point 1, the decision tree $d_1$ is initialized and starts the training phase. The learning algorithm has an initial training phase which is required to collect a sufficient amount of data to generate rules that pass the required statistical confidence level. After the initial training phase the quality of the decision tree rules is assessed by classifying new incoming traces. In Figure 7.4 correct classifications are printed in normal text, while incorrect classifications are printed in bold underlined font.

We have marked four particularly interesting time points $(a, b, c, d)$ in Figure 7.4, which we discuss in the following.

1. At time $a$ the tree $d_1$ misclassifies the trace $t_a$ as a false positive. This triggers the parallel training of a new decision tree $d_2$ based on the traces starting with $t_a$.

2. A false negative by $d_2$ occurs at time $b$. However, since this happens during the initial training phase of $d_2$, we simply regard the trace $t_b$ as useful information for the learner and add it to the training set. No further action is required.

3. Time point $c$ contains another false positive misclassification of $d_1$. In the meantime, $F1(d_1)$ had risen due to some correct classifications, but now the score is pushed down to 0.7. Again, as in time point $a$, the generation of a new tree $d_3$ is triggered.

4. At time $d$ the environment seems to have stabilized and decision tree $d_3$ reached a state with perfect classification ($F1(d_3) = 1$). At this point, the remaining decision trees are

rejected. The old trees are still stored for reference, but are not trained with further data to save computing power.

## 7.4 Implementation

Our prototype implementation of the presented fault localization approach is implemented in Java. We utilize the open-source machine learning framework Weka [108]. Weka contains an implementation of the popular *C4.5* decision tree deduction algorithm [129], denoted *J48 classifier* in Weka. C4.5 has been applied successfully in many application areas and is known for its good performance characteristics.

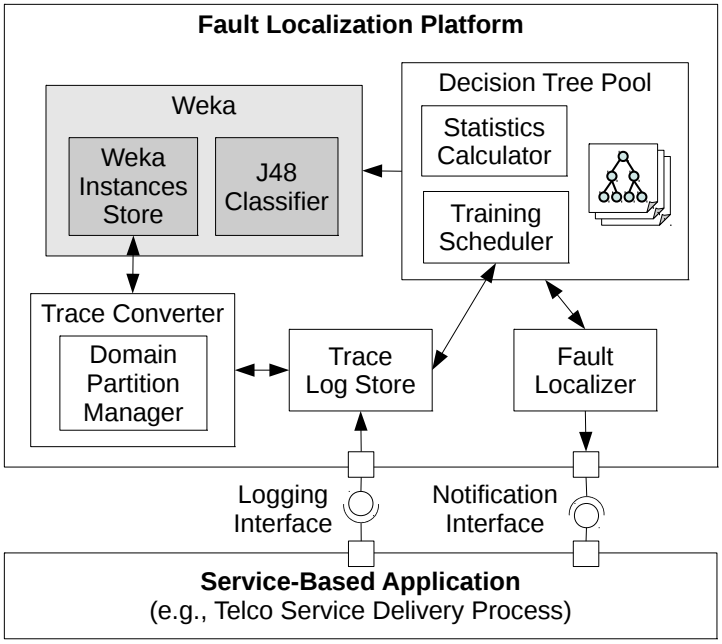



Figure 7.5: Prototype Implementation Architecture

Figure 7.5 outlines the architecture of the Fault Localization Platform with the core components. Third-party components (Weka) are depicted with light grey background color. The service-based application submits its log traces (service bindings plus input messages) to the Logging Interface and provides a Notification Interface to receive fault localization updates. The Trace Log Store receives trace data and forwards them to the Trace Converter. The Domain Partition Manager maintains the customizable value partitions for input messages. For instance, if a trace contains an integer input parameter $x = -173$ and the chosen domain partition for $x$ is $\{negative, zero, positive\}$ then the Trace Converter transforms the input to $x = negative$. The transformed traces are put to the Weka Instances Store. The Decision Tree Pool utilizes the Weka J48 Classifier to maintain the set of trees. The Statistics Calculator determines quality measures

for the learned classifiers, and the Training Scheduler triggers the adaptation of the tree pool to changing environments.

## 7.5 Evaluation

In the following, we evaluate different aspects of our proposed fault localization approach.

### *7.5.1 Evaluation Setup*

The test traces are generated randomly, with assumed uniform distribution of the underlying random generator.

| ID | $\lvert\mathbf{I}\rvert$ | $\lvert\mathbf{c(i)}\rvert$, $\mathbf{i} \in \mathbf{I}$ | $\lvert\mathbf{p(i)}\rvert$, $\mathbf{p(i)} \in \mathbf{P}$ | $\lvert\mathbf{d}\rvert$, $\mathbf{i} \in \mathbf{I}$ | $\lvert\mathbf{e}\rvert$, $\mathbf{e} \in \mathbf{E_I}$ | **Fault Probability** |
|---|---|---|---|---|---|---|
| $S1$ | 5 | 5 | 10 | 20 | $\{1\}$ | $4 * 10^{-2}$ |
| $S2$ | 5 | 5 | 10 | 20 | $\{2\}$ | $2 * 10^{-3}$ |
| $S3$ | 5 | 5 | 10 | 20 | $\{3\}$ | $1 * 10^{-4}$ |
| $S4$ | 5 | 5 | 10 | 20 | $\{3,3,3\}$ | $3 * 10^{-4}$ |
| $S5$ | 10 | 10 | 10 | 100 | $\{3,4\}$ | $1.001 * 10^{-6}$ |
| $S6$ | 10 | 10 | 10 | 100 | $\{4\}$ | $1 * 10^{-12}$ |

Table 7.3: Fault Probabilities for Exemplary SBA Model Sizes

Table 7.3 shows six different SBA instances with corresponding parameter settings that are considered for evaluation. $|I|$ denotes the number of service interfaces, $|c(i)|$ is the number of concrete implementations of each interface $i \in I$, $|p(i)|$ represents the number of input parameters per interface, $|d(p)|$ is the domain size for a parameter $p \in P$, and $|E_I|$ is the number of injected incompatibilities that cause the faults at runtime. The table also lists for each setting the probability that a fault occurs in a random execution.

All tests have been performed on machines with two Intel Xeon E5620 quad-core CPUs, 32 GB RAM, and running Ubuntu Linux 11.10 with kernel version 3.0.0-16.

### *7.5.2 Training Duration*

First, we evaluate how many fault traces are required by the J48 classifier to pass the threshold for reliable fault detection. The scenario SBAs $S3, S2, S1$ (cf. Table 7.3) were used in Figure 7.6, 20 iterations of the test were executed, and the figure contains three boxes representing the range of minimum and maximum values. As shown in Figure 7.6, the number of traces required to successfully detect a faulty configuration depends mostly on the complexity (i.e., probability) of the fault with regard to the total scenario size.

A single fault in the configuration $S1$ was on average detected after observing between 90 and 190 traces. If we multiply these values with the fault probability of $4 * 10^{-2}$, we get a range

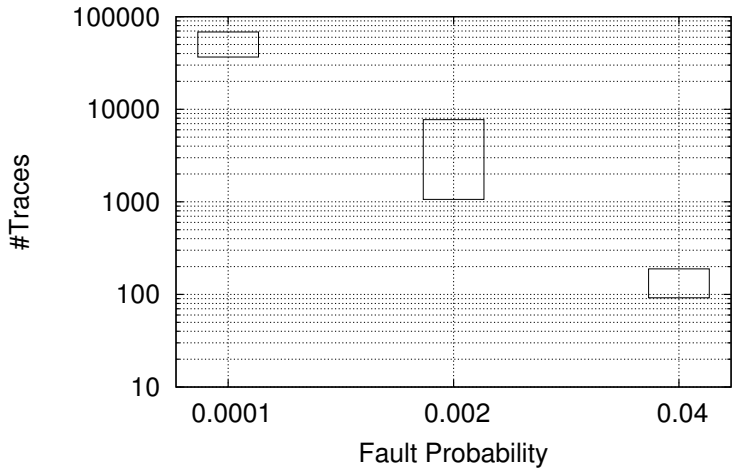



Figure 7.6: Number of Traces Required to Detect Faults of Different Probabilities

of 4 to 8 fault traces required for the localization. Also with more complex (unlikely) faults the relative figures do not appear to change considerably. With a fault probability of $2 * 10^{-3}$ and $1 * 10^{-4}$ the faults are detected after observing 3/16 and 4/7 minimum/maximum fault traces, respectively. The data suggest that there is a strong relationship between the number of required fault traces and the fault probability.

### *7.5.3 Transient Faults*

As discussed in Section 7.3.6, our fault localization approach is designed to cope with changing environments, which is evaluated here. Figure 7.7 shows the performance in the presence of changing faults. The evaluation setup is as follows: Initially a fault combination $FC1$ (e.g., $\langle t_x(i_2,'premium') = false, i_3 = c_8 \rangle$) is active. At trace 33000, the implementation that causes the fault $FC1$ is repaired, but the fix introduces a new fault $FC2$ that is fixed at trace 66000. At trace 66000, another fault $FC3$ occurs, and an attempted fix at trace 88000 introduces an additional fault $FC4$, while $FC3$ remains active. At trace 121000, both $FC3$ and $FC4$ are fixed, but two new faults $FC5$ and $FC6$ are introduced to the system. The occurrence probability for each of the fault combinations ($FC1 - FC6$) is set to $2 * 10^{-3}$ (corresponding to scenario setting $S2$ in Table 7.3).

This scenario is designed to mimic a realistic situation, but serves mainly to highlight several aspects of our solution. After about 4000 observed execution traces the localizer provides a first guess as to the cause of the fault, but the classification is not yet correct. After around 5200 observed execution traces, the localizer was able to analyze enough error traces to provide an accurate localization result. Note that at that time, only about 6 error traces have been observed, yet the algorithm already produces a correct result. At trace 33000, the previously detected fault $FC1$ disappears and is replaced by $FC2$. Due to the pool of decision trees maintained by our localizer, $FC2$ can again be accurately localized roughly 6000 traces later. Similarly, after $FC2$

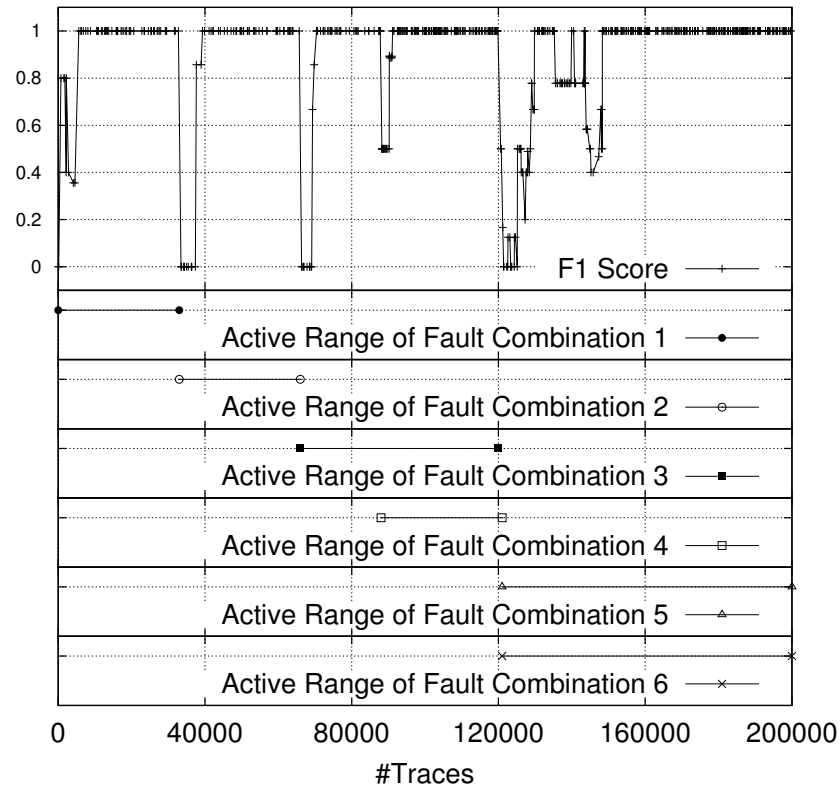



Figure 7.7: Fault Localization Accuracy for Dynamic Environment with Transient Faults

disappears, $FC3$ is localized roughly 5000 traces after its introduction.

The decision tree pool allows for the effective localization of new faults introduced to the system at any time. At trace 88000 in Figure 7.7, $FC4$ is introduced, and can again be accurately localized after observing around 5000 traces. $FC3$ and $FC4$ disappear at trace 121000 and are replaced by simultaneously occurring errors $FC5$ and $FC6$. This situation is more challenging for our approach, as seen in the rightmost 80000 traces in Figure 7.7. The spikes between trace 121000 and 150000 represent different localization attempts that are later invalidated by contradicting execution traces. Finally, however, the localization stabilizes and both faults $FC5$ and $FC6$ are accurately detected.

We also evaluated the performance of our approach using different noise levels in the trace logs. Figure 7.8 analyzes how the F1 score develops with increasing noise ratio. The figure contains four lines, one each for the scenario settings $S1 - S4$. To ensure that the algorithm actually obtained enough traces for fault localization, we executed the localization run after 200000 observed traces.

### *7.5.4 Runtime Considerations*

Due to the nature of the tackled problem, as well as the usage of C4.5 decision trees to generate rules, there are some practical limitations to the number of traces and scenario sizes that can

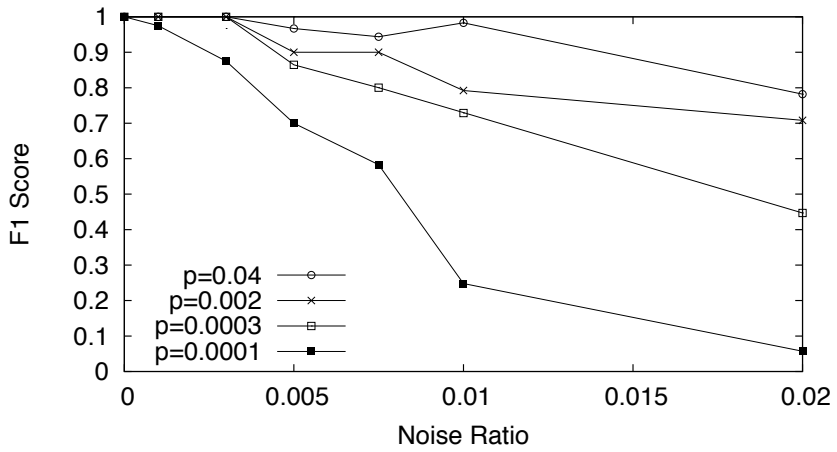



Figure 7.8: Noise Resilience. Our approach maintains reasonable accuracy in the presence of noisy data.

be analyzed using our approach within a reasonable time. In the following, we provide insights into the runtime performance in different configurations and discuss strategies for fine-tuning the performance.

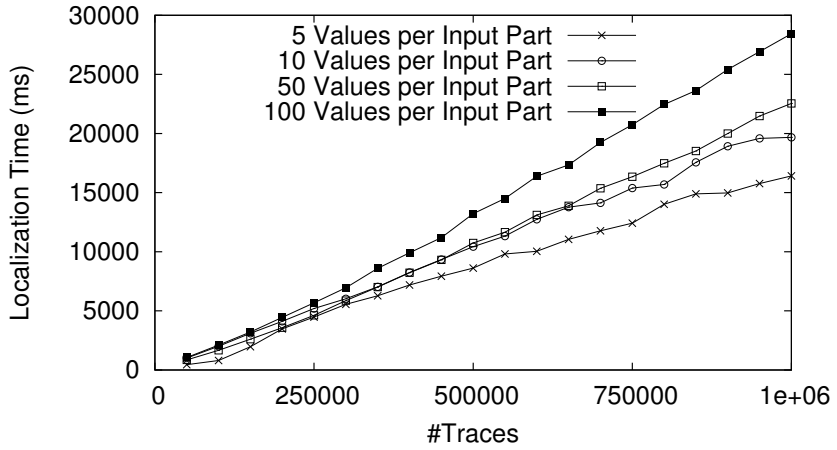



Figure 7.9: Localization time for different trace window sizes in the scenario *S*5 for input sizes $|d| = \{5, 10, 50, 100\}$.

Figure 7.9 shows the time needed for to localize faults for various trace window sizes for the base scenario *S*5, for input sizes $|d| = \{5, 10, 50, 100\}$. The figure illustrates that the time needed for a single localization run increases roughly linearly with increasing window sizes. Larger trace windows allow the algorithm to find more complex faults. If fast localization results are needed, the window size must be kept adequately small, at the cost of the system not being able to localize

faults above a certain complexity.

Furthermore, the frequency of localization runs must be considered when implementing our approach in systems with very frequent incoming traces (in the area of hundreds or thousands of traces per second). Evidently, there is a natural limit to the number of traces that can be processed per time unit. Figure 7.10 shows the localization speed as number of traces processed per second compared to different fault localization intervals (i.e., number of traces after which fault localization is triggered periodically) for different window sizes ($|T|$, i.e., number of considered traces).

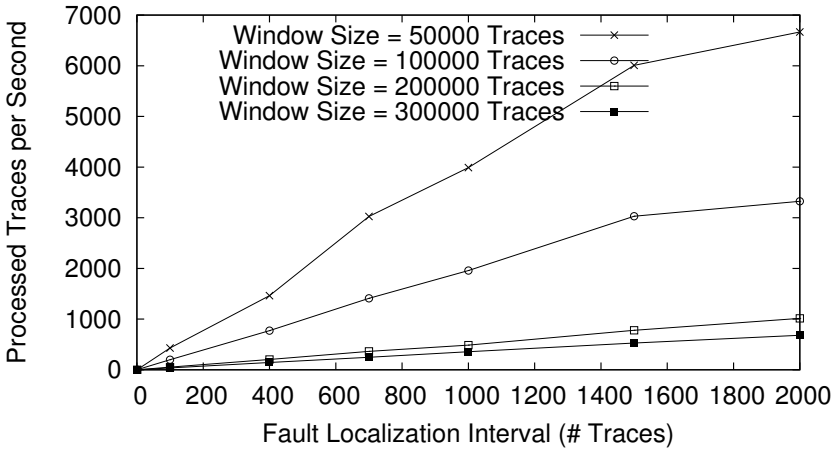



Figure 7.10: Localization performance in traces per second for different fault localization intervals and window sizes, using scenario *S*5

The data in Figure 7.10 can be seen as a performance benchmark for the machine(s) on which the fault localization is executed. Executing this test on different machines will result in different performance footprints, which serves as a decision support for configuring window size and localization interval. For instance, if our application produces 1500 traces per second (i.e., processes 1500 requests per second), a localization interval greater than 400 should be used. Currently, the selection happens manually, but as part of our future work we investigate means to fine-tune this configuration automatically.

## 7.6 Related Work

In this section we discuss existing approaches related to reliability, fault detection, and fault localization in SBAs and distributed systems in general.

### *7.6.1 Software Testing*

Our work is related to the broad field of software testing where a plethora of approaches for fault localization have been proposed. Generally, software testing is the process of executing a program

or systems with the intent of finding errors [116]. Testing approaches are often divided into white- and black-box testing. In white-box (or logic-driven) testing the internals of the software under test are visible to the tester. Black-box (input/output-driven) testing has to get along with no information about internal structure. Our problem formulation faces a black-box model in which we can observe the system behavior but have no details about the internals. Formal verification of software is an alternative to testing that is often employed in highly safety-critical environments.

Canfora et al. [30] provide an extensive overview of testing services and SBAs. The seminal work by Narayanan and McIlraith [117] was among the first to perform automated simulation and verification based on a semantic model of Web services. Another related approach has been presented in [67], which performs upfront integration testing with different combinations of concrete service implementations. Due to the huge search space, even in medium sized SBAs, their test case generation approach is not able to consider the service input and output data, whereas the efficient fault localization algorithms used in this work allow us to do so. Concluding, in software testing a system is actively executed to find problems; in this work, however, we do not control the software but we monitor its execution to localize faults and fault reasons at runtime.

### *7.6.2 Software Fault Localization*

Software fault localization helps to identify bugs in software on the source code level. Oftentimes a two-phase procedure is applied: 1) finding suspicious code that may contain bugs and 2) examining the code and deciding whether it contains bugs with the goal of fixing them. Research mainly focused on the former, the identification of suspicious code parts with prioritization based on its likelihood of containing bugs [39, 101, 159]. The seminal paper by Hutchins et al. [71] introduces an evaluation environment suitable for fault localization (often referred to as the *Siemens suite*), consisting of seven base programs (in different versions) that have been seeded with faults on the source code level. Fundamental research on statistical bug isolation is presented in [101]. Decision branches are modeled as predicates, and conditional probabilities are used to compute the likelihood that a failure occurs in a certain branch.

Renieres et al. [132] present a fault localization technique for identifying suspicious lines of a program's source code. Based on the existence of a faulty run of the program and many correct runs they select the correct run that is most similar to the faulty one. Proximity is defined based on the program spectra. Then, traces of the two runs are compared and suspicious program lines are reported. This general approach is very common in software fault localization. Arguing that traditional trace proximity (literal comparison of traces) is insufficient as faults can be triggered in various ways, Liu and Han [103] introduce *R-Proximity* which regards two traces as similar if they appear to have roughly the same fault location. Guo et al. [59] propose a different similarity metric based on control flow. The metric takes into account the sequence of statement rather than just the unordered set. Our work differs from traditional software fault localization in that we do not analyze program code but only observe the runtime behavior of services. We also assume that the environment or service implementations may change during runtime, in contrast to the analysis of static code. The work in [84] assists humans in localizing software faults by visualizing test information and highlighting suspicious code statements with different color intensity. The empirical study conducted shows that single faults are evidently easier to find

for humans than complex fault combinations, which strengthens the motivation for automated machine learning based fault localization, as studied here.

### *7.6.3 Monitoring and Fault Detection*

Monitoring and fault detection are key challenges for implementing reliable distributed systems. Fault detectors are a general concept in distributed systems and aim at identifying faulty components. In asynchronous systems it is in fact impossible to implement a perfect fault detector [34], because faults cannot be distinguished with certainty from lost or delayed messages. Heartbeat messages can be used for probabilistic detection of faulty components; in this case a monitored component or service has the responsibility to send heartbeats to a remote entity. The fault detector presented in [139] considers the heartbeat inter-arrival times and allows for a computation of a component's faulty behavior probability based on past behavior. Steinder and Sethi [149] study fault localization in communication systems using belief networks. The approach is noise resilient and able to handle spurious events, but if fault conditions change permanently, updates in the belief network are arguably slower than using pooled decision trees. Moreover, their results indicate that fault localization time has exponential growth in the number of network nodes, whereas our centralized approach scales near-linearly in the number of traces. Lin et al. [102] describes a middleware architecture called *LIama* that advocates a service bus that can be installed on existing service-based infrastructures. It collects and monitors service execution data which enable to incorporate fault detection mechanisms using the data. Such a service bus can be used to collect the data necessary for our analysis. The major body of research in the area of monitoring and fault detection in SBAs deals with topics like SLAs [100] and service compositions rather than compatibility issues [123].

### *7.6.4 Fault Analysis and Adaptation*

Fault analysis derives knowledge from faults that have been experienced. Adaptation tries to leverage this knowledge to reconfigure the system to overcome faults. Oftentimes, domain-specific knowledge is required to efficiently analyze faults and their origins (e.g., [69]). Zhou et al. [166] have proposed GAUL, a problem analysis technique for unstructured system logs. Their approach is based on enterprise storage systems, whereas we focus on dynamic service-based applications. GAUL uses a fuzzy match algorithm based on string similarity metrics to associate problem occurrences with log output lines. The aim of GAUL differs from our approach since we assume the existence of structured log files and focus on the localization of faulty configuration parameters. Control of SOAs mostly relies on static approaches, such as predefined policies [125]. Techniques from artificial intelligence can be used to improve management policies for SBAs during runtime. For instance, Markov decision processes represent a possible way for modeling the decision-making problems that arise in controlling SBAs.

## 7.7 Summary

In this chapter we describe a fault localization technique that is able to identify which combinations of service bindings and input data cause problems in SBAs. The analysis is based on

log traces, which accumulate during runtime of the SBA. A decision tree learning algorithm is employed to construct a tree from which we extract rules, describing which configurations are likely to lead to faults. For providing a fine-grained analysis we do not only consider the service bindings but also data on message level. This allows to find incompatibilities that go beyond "service A has incompatibility issues with service B" leading to rules of the form "service A has incompatibility issues with service B for messages of type C". Such rules can help to safely use partial functionality of services. We present extensions to our basic approach that help to cope with dynamic environments and changing fault patterns. We have conducted experiments based on scenario traces of realistic size. The results provide evidence that the employed approach leads to successful fault localization for dynamically changing conditions, and is able to cope with the large amounts of data that accumulate by considering fine-grained data on message level.

CHAPTER 8

# Conclusion and Future Research

In this chapter we summarize the main results of this thesis. In Section 8.1 we focus on the core outcomes of the conducted work and how the state of the art in research was advanced as part of this work. Then, the research questions posited in Section 1.2 are revisited and critically analyzed in Section 8.2. Finally, Section 8.3 discusses ongoing trends and open topics in related research areas for future research to build on the contributions presented in this work.

## 8.1 Summary of Contributions

In this thesis, we have presented novel methods for coping with the challenges of evolution and adaptation of SBAs in cloud computing environments in a structured and predictable way. We take a holistic viewpoint and align our discussion with the phases of the software development lifecycle, ranging from managing software evolution in a structured way at design time, reliable specification and deployment of complex adaptation infrastructures at deployment time, to improving policy and fault management at runtime.

The presented evolution lifecycle model and framework allow for the structured evolution and adaptation of SBAs throughout all phases of the software development lifecycle by capturing and documenting relevant information as it becomes available, enabling traceability of design decisions and their evolutionary changes, from abstract descriptions of intents to concrete and actionable application components, their dependencies and interactions within the system. Moreover, the model and accompanying strategies allow for unified handling of change requests in any lifecycle phase, and facilitate the propagation of necessary changes between phases in a controlled manner. To facilitate effective modeling and optimized deployment of application runtime management infrastructures in cloud environments, we have introduced a method for provider-managed adaptation that enables customers to leverage provider experience in managing complex distributed systems without requiring large upfront investments. A novel DSL called MONINA was created to model application structure, monitoring queries, and adaptation rules, along with strategies for optimized deployment and increased maintainability, allowing

complex SBAs to effectively and efficiently react to changes in their environment without operators needing to implement custom management infrastructure. To further improve created application management policies and increase dependability at runtime, we have presented approaches based on machine learning techniques to incrementally improve adaptation policies without explicit domain knowledge, along with a novel technique for automated identification of service implementation incompatibilities using pooled decision trees.

The results of our investigations were evaluated based on multiple case studies and showed that our approaches can significantly contribute to facilitate structured evolution of SBSs and increase system robustness by autonomically improving adaptation policies. For cloud providers, our work allows the creation of new revenue streams by offering managed adaptation solutions, as well as increased resource efficiency by leveraging additional information gained from application management policies.

## 8.2 Research Questions Revisited

The research questions introduced in Section 1.2 guided the work in this thesis. In this section, we revisit these questions and summarize how they have been answered within the context of our work, along with a discussion of the limitations of the presented solution.

*Research Question I:*
*How can software evolution and adaptation be explicitly incorporated in cloud application design and management?*

We demonstrated that explicit documentation and structured realization of evolutionary changes throughout the software development process enables partial automation of far-reaching changes in SBAs with significantly reduced human intervention, along with increased transparency owing to seamless traceability of design decisions. As part of our solution, we introduced a DSL and accompanying adaptation infrastructure allowing for optimized deployment and management of adaptation concerns suitable for a wide range of business domains. However, our approaches currently rely on software designers explicitly modeling evolution and adaptation concerns using the proposed methods. To allow for a higher degree of automation, it is necessary to also gather as much information as possible from existing tools that are used in the development process, such as software and bug repositories, requirements models, and software architecture models. Furthermore, the presented adaptation infrastructure currently relies on traditional ECA rules, whereas the evolution model would allow for more expressive, goal-based adaptation strategies, which have not been studied in-depth in this work.

*Research Question II:*
*How can explicit cloud application design and management be autonomously improved in the face of changes in their environment?*

The presented policy improvement approach can autonomously improve application management policies without specific domain knowledge. While the approach can effectively improve

management policies and reduce the occurrence of failures, the resulting policies are only suitable for consumption by an autonomic manager, but meaningful rules for human consumption (and subsequent improvement of explicit management policies) are not created in the current work, due to the nature of the used machine learning techniques. In contrast, the introduced fault detection approach can reliably detect service implementation incompatibilities and create rules that are meaningful for human operators. While the technique can reliably detect and create rules for incompatible combinations of service implementations and parameters, the effectiveness depends on proper partitioning of the source attributes to reduce the state space to a size feasible for our approach. Currently, this is a manual step that must be performed by domain experts.

## 8.3 Future Work

In this work we presented different aspects to enable controlled evolution and adaptation of SBSs in cloud computing environments. However, based on the discussion in Section 8.2, it is apparent that a number of important challenges were out of scope for this thesis. In the following, we outline some challenges and possibilities for future research.

- It is expected that future research will build on the evolution lifecycle and adaptation model created in this work to derive a comprehensive software development process for cloud applications. This development process will be tailored to the fundamental properties of cloud computing environments, such as resource, cost, and quality elasticity [46], and will enable the creation and evolution of cloud-native applications in an agile way.

- In future work we envision tighter mutual integration of the proposed approaches for fault detection and policy improvement to automatically assert previously unknown fault states when they occur, as well as improve the modeled rules and augment system control policies, along with automated testing [70] to identify incompatible configurations of activated artifacts.

- The approach for specifying and deploying application adaptation infrastructures should be extended to integrate continuous deployment techniques, i.e., the capability to migrate elements at runtime to adapt according to more precise knowledge and changing environments. Furthermore, the presented framework should be integrated with and adopted by current cloud management tools, such as OpenStack Heat [119].

- With regard to the presented policy improvement framework, we expect that future work will integrate service level objectives, as well as request payload data, into the policy generation in addition to log data. Another future research direction includes to consider not only the service level but also the resource level, i.e., to control the mapping of services to resources. The techniques presented in this work allow to manage relatively complex service-oriented systems. However, if large-scale highly complex systems are to be controlled, algorithms for complexity reduction, such as principal component analysis, could be employed.

# Bibliography


[1] David W. Aha. Tolerating noisy, irrelevant and novel attributes in instance-based learning algorithms. *International Journal of Man-Machine Studies*, 36(2):267–287, 1992. ISSN 0020-7373. doi:10.1016/0020-7373(92)90018-G.

[2] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the 30th International Conference on Very large data bases*, VLDB '04, pages 456–467. VLDB Endowment, 2004. ISBN 0120884690. acmid:1316730.

[3] Marco Aldinucci, Marco Danelutto, and Peter Kilpatrick. Autonomic management of non-functional concerns in distributed & parallel application programming. In *IEEE International Symposium on Parallel Distributed Processing, 2009*, IPDPS '09., pages 1–12, 2009. doi:10.1109/IPDPS.2009.5161034.

[4] E. Emanuel Almeida, Jonathan E. Luntz, and Dawn M. Tilbury. Event-condition-action systems for reconfigurable logic control. *Automation Science and Engineering, IEEE Transactions on*, 4(2):167–181, 2007. doi:10.1109/TASE.2006.880857.

[5] Amazon Web Services, Inc. CloudFormation, 2014. http://aws.amazon.com/cloudformation. [Online; accessed January 17, 2014].

[6] Amazon Web Services, Inc. CloudWatch, 2014. http://aws.amazon.com/cloudwatch. [Online; accessed January 17, 2014].

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html.

[8] Michael Armbrust, Ion Stoica, Matei Zaharia, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, and Ariel Rabkin. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, April 2010. doi:10.1145/1721654.1721672.

[9] CFEngine AS. CFEngine, 2014. http://cfengine.com. [Online; accessed January 17, 2014].

[10] Ricardo Baeza-Yates and Ribeiro-Neto Berthier. *Modern information retrieval*. Addison-Wesley, 1999. ISBN 978-0201398298.

[11] Raphael M. Bahati and Michael A. Bauer. Modelling reinforcement learning in policy-driven autonomic management. *International Journal On Advances in Intelligent Systems*, 1(1):54–79, 2008. ISSN 1942-2679.

[12] Raphael M. Bahati, Michael A. Bauer, and Elvis M. Vieira. Policy-driven autonomic management of multi-component systems. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '07, pages 137–151, New York, NY, USA, 2007. ACM. doi:10.1145/1321211.1321226.

[13] Olivier Barais, Anne Françoise Meur, Laurence Duchien, and Julia Lawall. Software architecture evolution. In *Software Evolution* Mens and Demeyer [111], pages 233–262. ISBN 978-3-540-76439-7. doi:10.1007/978-3-540-76440-3_10.

[14] Luciano Baresi and Sam Guinea. Dynamo and Self-Healing BPEL Compositions. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 69–70, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2892-9. doi:10.1109/ICSECOMPANION.2007.31.

[15] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–20, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-798-8. doi:10.1145/1294904.1294906.

[16] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2nd edition, 2003. ISBN 978-0321154958.

[17] Mokhtar S. Bazaraa, Hanif D. Sherali, and C. M.R Shetty. *Nonlinear Programming: Theory and Algorithms*. Wiley-Interscience, 3rd edition, 2006. ISBN 978-0-4714-8600-8.

[18] Wesley Beary. fog – the ruby cloud services library, 2014. http://fog.io. [Online; accessed January 17, 2014].

[19] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, October 1999. ISBN 978-0201616415.

[20] Richard E. Bellman. *Dynamic programming*. Dover Books on Computer Science. Dover Publications, 2003. ISBN 978-0486428093.

[21] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, New York, NY, USA, 2000. ACM. doi:10.1145/336512.336534.

[22] Keith H. Bennett, Václav T. Rajlich, and Norman Wilde. Software evolution and the staged model of the software lifecycle. *Advances in Computers*, 56:1–54, 2002. doi:10.1016/S0065-2458(02)80003-1.

[23] Benjamin Blau, Jan Kramer, Tobias Conte, and Clemens van Dinther. Service value networks. In *Proceedings of the IEEE Conference on Commerce and Enterprise Computing*, CEC '09, Washington, DC, USA, 2009. IEEE Computer Society. doi:10.1109/CEC.2009.64.

[24] Jason Bloomberg. *The Agile Architecture Revolution*. How Cloud Computing, REST-Based SOA, and Mobile Computing Are Changing Enterprise IT. John Wiley & Sons, January 2013. ISBN 9781118421994.

[25] Robert Bohn, John Messina, Fang Liu, Jin Tong, and Jian Mao. NIST cloud computing reference architecture. In *Proceedings of the 2011 IEEE World Congress on Services*, SERVICES '11, pages 594–596, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/SERVICES.2011.105.

[26] Ivona Brandic. Towards self-manageable cloud services. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, COMPSAC '09, Washington, DC, USA, 2009. IEEE Computer Society. doi:10.1109/COMPSAC.2009.126.

[27] Aaron B. Brown and David A. Patterson. To err is human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY*, EASY '01, 2001. http://roc.cs.berkeley.edu/papers/easy01.pdf.

[28] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Software: Practice and Experience*, 36(11-12):1257–1284, August 2006. doi:10.1002/spe.767.

[29] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599–616, June 2009. doi:10.1016/j.future.2008.12.001.

[30] Gerardo Canfora and Massimiliano Di Penta. Testing services and service-centric systems: challenges and opportunities. *IT Professional*, 8(2):10–17, 2006. ISSN 1520-9202. doi:10.1109/MITP.2006.51.

[31] Canonical, Inc. Juju, 2014. http://juju.ubuntu.com/. [Online; accessed January 17, 2014].

[32] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Francesco Lo Presti, and Raffaela Mirandola. Qos-driven runtime adaptation of service oriented architectures. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09. ACM, 2009. doi:10.1145/1595696.1595718.

[33] K.S. May Chan and Judith Bishop. The design of a self-healing composition cycle for web services. In *Proceedings of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '09, pages 20–27, Washington, DC, USA, 2009. IEEE Computer Society. doi:10.1109/SEAMS.2009.5069070.

[34] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. doi:10.1145/226643.226647.

[35] Fangzhe Chang, Ramesh Viswanathan, and Tom L. Wood. Placement in clouds for application-level latency requirements. In *Proceedings of the 5th International Conference on Cloud Computing*, CLOUD '12, pages 327–335. IEEE, 2012. doi:10.1109/CLOUD.2012.91.

[36] Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic, and Akhil Sahai. Sla decomposition: Translating service level objectives to system level thresholds. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, pages 3–3. IEEE, 2007. doi:10.1109/ICAC.2007.36.

[37] Betty H C Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In *Lecture Notes in Computer Science*, pages 1–26. Springer, Berlin, Heidelberg, 2009. ISBN 978-3-642-02161-9. doi:10.1007/978-3-642-02161-9_1.

[38] Michal R. Chmielewski and Jerzy W. Grzymala-Busse. Global discretization of continuous attributes as preprocessing for machine learning. *International Journal of Approximate Reasoning*, 15(4):319 – 331, 1996. doi:10.1016/S0888-613X(96)00074-6.

[39] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM. doi:10.1145/1062455.1062522.

[40] Marco Comuzzi, Constantinos Kotsokalis, George Spanoudakis, and Ramin Yahyapour. Establishing and monitoring SLAs in complex service based systems. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '09, pages 783–790, Washington, DC, USA, 2009. IEEE Computer Society. doi:10.1109/ICWS.2009.47.

[41] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. SHIWS: A self-healing integrator for web services. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, pages 55–56, Washington, DC, USA, 2007. IEEE Computer Society. doi:10.1109/ICSECOMPANION.2007.66.

[42] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. Designing self-adaptive service-oriented applications. In *Proceedings of the 4th International Conference on Autonomic Computing*, ICAC '07, page 16. IEEE Computer Society, June 2007. doi:10.1109/ICAC.2007.13.

[43] Marios D. Dikaiakos, Asterios Katsifodimos, and George Pallis. Minersoft: Software retrieval in grid and cloud computing infrastructures. *Transactions on Internet Technology*, 12(1), June 2012. doi:10.1145/2220352.2220354.

[44] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *Proceedings of the 12th International Conference on Machine Learning*, pages 194–202. Morgan Kaufmann Publishers, Inc., 1995.

[45] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, January 2005. ISSN 1741-1114.

[46] Schahram Dustdar, Yike Guo, Benjamin Satzger, and Hong-Linh Truong. Principles of elastic processes. *Internet Computing, IEEE*, 15(5):66–71, 2011. doi:10.1109/MIC.2011.121.

[47] Marco D'Ambros, Harald Gall, Michele Lanza, and Martin Pinzger. Analysing software repositories to understand software evolution. In *Software Evolution* Mens and Demeyer [111], pages 37–67. ISBN 978-3-540-76440-3. doi:10.1007/978-3-540-76440-3_3.

[48] Eclipse Foundation. Xtext Documentation, 2014. http://www.eclipse.org/Xtext/documentation.html. [Online; accessed January 17, 2014].

[49] Thomas Erl. *Service-Oriented Architecture*. Concepts, Technology, and Design. Prentice Hall, August 2005. ISBN 9780132715829.

[50] Thomas Erl, Richardo Puttini, and Zaigham Mahmood. *Cloud Computing: Concepts, Technology & Architecture*. The Prentice Hall Service Technology Series from Thomas Erl. Prentice Hall, 2013. ISBN 9780133387520.

[51] EsperTech. Esper Reference Documentation, 2014. http://esper.codehaus.org/esper/documentation/documentation.html. [Online; accessed January 17, 2014].

[52] Marc Fiammante. Dynamic SOA and BPM: From simplified integration to dynamic processes. In *Dynamic SOA and BPM: Best Practices for Business Process Management and SOA Agility*. IBM Press, 2009.

[53] Apache Software Foundation. ActiveMQ, 2014. http://activemq.apache.org. [Online; accessed January 17, 2014].

[54] Apache Software Foundation. jclouds, 2014. http://jclouds.apache.org. [Online; accessed January 17, 2014].

[55] Apache Software Foundation. Libcloud python library, 2014. http://libcloud.apache.org. [Online; accessed January 17, 2014].

[56] Apache Software Foundation. Tuscany SCA, 2014. http://tuscany.apache.org. [Online; accessed January 17, 2014].

[57] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, October 2004. doi:10.1109/MC.2004.175.

[58] Scott M. Glen and Jens Andexer. A practical application of SOA, October 2007. https://web.archive.org/web/20090226230835/http://www.ibm.com/developerworks/webservices/library/ws-soa-practical/. [Online; accessed January 17, 2014].

[59] Liang Guo, Abhik Roychoudhury, and Tao Wang. Accurately choosing execution runs for software fault localization. In *Proceedings of the 15th International Conference on Compiler Construction*, CC '06, pages 80–95. Springer, 2006. doi:10.1007/11688839_7.

[60] Gurobi Optimization, Inc. Gurobi optimizer reference manual, 2014. http://www.gurobi.com/resources/documentation. [Online; accessed January 17, 2014].

[61] Reiko Heckel, Rui Correia, Carlos Matos, Mohammad El-Ramly, Georgios Koutsoukos, and Luis Andrade. Architectural Transformations: From Legacy to Three-Tier and Services. In *Software Evolution* Mens and Demeyer [111], pages 139–170–170. ISBN 978-3-540-76440-3. doi:10.1007/978-3-540-76440-3_7.

[62] Robert C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11(1):63–90, 1993. doi:10.1023/A:1022631118932.

[63] Paul Horn. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. IBM, Armonk, NY, USA, 2001. https://web.archive.org/web/20050310235031/http://www-1.ibm.com/industries/government/doc/content/bin/auto.pdf.

[64] George Hripcsak and Adam S. Rothschild. Agreement, the f-measure, and reliability in information retrieval. *Journal of the American Medical Informatics Association*, 12(3):296–298, 2005. doi:10.1197/jamia.M1733.

[65] Markus C Huebscher and Julie A McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, 40(3), August 2008. doi:10.1145/1380584.1380585.

[66] Waldemar Hummer, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Dynamic migration of processing elements for optimized query execution in event-based systems. In *On the Move to Meaningful Internet Systems*, OTM '11, pages 451–468, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-25106-1_3.

[67] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Test coverage of data-centric dynamic compositions in service-based systems. In *Proceedings of the Fourth International Conference on Software Testing, Verification and Validation*,

ICST '11, pages 40–49, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/ICST.2011.55.

[68] Waldemar Hummer, Benjamin Satzger, Philipp Leitner, Christian Inzinger, and Schahram Dustdar. Distributed continuous queries over web service event streams. In *Proceedings of the 7th International Conference on Next Generation Web Services Practices*, NWeSP '11, pages 176–181, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/NWeSP.2011.6088173.

[69] Waldemar Hummer, Christian Inzinger, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Deriving a unified fault taxonomy for event-based systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 167–178, New York, NY, USA, 2012. ACM. doi:10.1145/2335484.2335504.

[70] Waldemar Hummer, Orna Raz, Onn Shehory, Philipp Leitner, and Schahram Dustdar. Testing of data-centric and event-based dynamic service compositions. *Software Testing, Verification and Reliability*, 23(6):465–497, 2013. doi:10.1002/stvr.1493.

[71] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society. ISBN 0-8186-5855-X. acmid:257766.

[72] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Identifying incompatible implementations of industry standard service interfaces for dependable service-based applications. Technical Report TUV-1841-2012-1, Vienna University of Technology, 2012.

[73] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Towards identifying root causes of faults in service-based applications. In *Proceedings of the 31st IEEE International Symposium on Reliable Distributed Systems*, SRDS '12, pages 404–405, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/SRDS.2012.78.

[74] Christian Inzinger, Benjamin Satzger, Waldemar Hummer, and Schahram Dustdar. Specification and deployment of distributed monitoring and adaptation infrastructures. In *Proceedings of the International Workshop on Performance Assessment and Auditing in Service Computing, co-located with ICSOC '12*, PAASC '12, pages 167–178, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-37804-1_18.

[75] Christian Inzinger, Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Non-intrusive policy optimization for dependable and adaptive service-oriented systems. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 504–510, New York, NY, USA, 2012. ACM. doi:10.1145/2245276.2245373.

[76] Christian Inzinger, Waldemar Hummer, Ioanna Lytra, Philipp Leitner, Huy Tran, Uwe Zdun, and Schahram Dustdar. Decisions, models, and monitoring – A lifecycle model for the evolution of service-based systems. In *Proceedings of the 17th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '13, pages 185–194, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/EDOC.2013.29.

[77] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Identifying incompatible service implementations using pooled decision trees. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 485–492, New York, NY, USA, 2013. ACM. doi:10.1145/2480362.2480456.

[78] Christian Inzinger, Benjamin Satzger, Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Model-based adaptation of cloud computing applications. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD '13), Special Track on Model-driven Software Adaptation*, MODA '13, pages 351–355. SciTePress, 2013. doi:10.5220/0004381803510355.

[79] Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems. *Software: Practice and Experience*, 2014. doi:10.1002/spe.2254. (to appear).

[80] Christian Inzinger, Stefan Nastic, Sanjin Sehic, Michael Vögler, Fei Li, and Schahram Dustdar. MADCAT – A methodology for architecture and deployment of cloud application topologies. In *Proceedings of the 8th International Symposium on Service-Oriented System Engineering*, SOSE '14, Washington, DC, USA, 2014. IEEE Computer Society. (to appear).

[81] Florian Irmert, Thomas Fischer, and Klaus Meyer-Wegener. Runtime adaptation in a service-oriented component model. In *Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '08, pages 97–104, New York, NY, USA, 2008. ACM. doi:10.1145/1370018.1370036.

[82] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, WICSA '05, pages 109–120, Washington, DC, USA, 2005. IEEE Computer Society. doi:10.1109/WICSA.2005.61.

[83] JBoss Drools team. Drools Expert User Guide, 2014. http://docs.jboss.org/drools/release/5.5.0.Final/drools-expert-docs/html_single/index.html. [Online; accessed January 17, 2014].

[84] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM. doi:10.1145/581339.581397.

[85] Gueyoung Jung, Kaustubh R. Joshi, Matti A. Hiltunen, Richard D. Schlichting, and Calton Pu. Generating adaptation policies for multi-tier applications in consolidated server environments. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '08, pages 23–32, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/ICAC.2008.21.

[86] Gueyoung Jung, Matti A. Hiltunen, Kaustubh R. Joshi, Richard D. Schlichting, and Calton Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, ICDCS '10, pages 62–73, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/ICDCS.2010.88.

[87] Leslie Pack Kaelbing, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. doi:10.1613/jair.301.

[88] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. doi:10.1109/MC.2003.1160055.

[89] Hyunjoo Kim, Yaakoub el Khamra, Shantenu Jha, and Manish Parashar. Exploring application and infrastructure adaptation on hybrid grid-cloud infrastructure. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 402–412, New York, NY, USA, 2010. ACM. doi:10.1145/1851476.1851536.

[90] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional, 3rd edition, 2004. ISBN 9780321197702.

[91] Geetika T. Lakshmanan, Ying Li, and Rob Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, 2008. doi:10.1109/MIC.2008.129.

[92] George Lawton. Developing software online with platform-as-a-service technology. *Computer*, 41(6):13–15, June 2008. doi:10.1109/MC.2008.185.

[93] Meir M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, January 1979. doi:10.1016/0164-1212(79)90022-0.

[94] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. doi:10.1109/PROC.1980.11805.

[95] Meir M. Lehman, Juan F. Ramil, P D Wernick, D E Perry, and W M Turski. Metrics and laws of software evolution-the nineties view. In *Proceedings of the Fourth International Software Metrics Symposium*, METRIC '97, pages 20–32, Washington, DC, USA, 1997. IEEE Computer Society. doi:10.1109/METRIC.1997.637156.

[96] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. Monitoring, prediction and prevention of sla violations in composite services. In *Proceedings of the 2010 IEEE International Conference on Web Services*, ICWS '10, pages 369–376, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/ICWS.2010.21.

[97] Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. Cost-efficient and application sla-aware client side request scheduling in an infrastructure-as-a-service cloud. In *Proceedings of the 5th IEEE International Conference on Cloud Computing*, CLOUD '12, pages 213–220, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/CLOUD.2012.21.

[98] Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *Proceedings of the 5th IEEE International Conference on Service-Oriented Computing and Applications*, SOCA '12, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/SOCA.2012.6449437.

[99] Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger, and Schahram Dustdar. CloudScale – a novel middleware for building transparently scaling cloud applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 434–440, New York, NY, USA, 2012. ACM. doi:10.1145/2245276.2245360.

[100] Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Cost-based optimization of service compositions. *IEEE Transactions on Services Computing*, 6(2):239–251, 2013. doi:10.1109/TSC.2011.53.

[101] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Cconference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM. doi:10.1145/1064978.1065014.

[102] Kwei-Jay Lin, Mark Panahi, Yue Zhang, Jing Zhang, and Soo-Ho Chang. Building accountability middleware to support dependable SOA. *Internet Computing*, 13(2), 2009. doi:10.1109/MIC.2009.28.

[103] Chao Liu and Jiawei Han. Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '06, pages 46–56, New York, NY, USA, 2006. ACM. doi:10.1145/1181775.1181782.

[104] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. NIST cloud computing reference architecture. *NIST Special Publication*, 500-292, 2011.

[105] Emil C. Lupu and Morris Sloman. Conflicts in policy-based distributed systems management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999. doi:10.1109/32.824414.

[106] Ioanna Lytra, Stefan Sobernig, and Uwe Zdun. Architectural decision making for service-based platform integration: A qualitative multi-method study. In *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, WICSA-ECSA '12, pages 111–120, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/WICSA-ECSA.212.19.

[107] Ioanna Lytra, Huy Tran, and Uwe Zdun. Constraint-based consistency checking between design decisions and component models for supporting software architecture evolution. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 287–296, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/CSMR.2012.36.

[108] Machine Learning Group at the University of Waikato. Weka 3: Data Mining Software in Java, 2014. http://www.cs.waikato.ac.nz/ml/weka/. [Online; accessed January 17, 2014].

[109] Peter Mell and Timothy Grance. The NIST definition of cloud computing. *NIST Special Publication*, 800-145, 2011.

[110] Tom Mens. Introduction and roadmap: History and challenges of software evolution. In *Software Evolution* Mens and Demeyer [111], pages 1–11. ISBN 978-3-540-76440-3. doi:10.1007/978-3-540-76440-3_1.

[111] Tom Mens and Serge Demeyer. *Software Evolution*. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-76440-3.

[112] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. End-to-end support for QoS-aware service selection, binding, and mediation in VRESCo. *IEEE Transactions on Services Computing*, 3(3):193–205, 2010. doi:10.1109/TSC.2010.20.

[113] Stefano Modafferi, Enrico Mussi, and Barbara Pernici. SH-BPEL: a self-healing plug-in for Ws-BPEL engines. In *Proceedings of the 1st workshop on Middleware for Service Oriented Computing*, MW4SOC '06, pages 48–53, New York, NY, USA, 2006. ACM. doi:10.1145/1169091.1169099.

[114] Monitis. Cloud monitoring, 2014. http://www.monitis.com/cloud-monitoring. [Online; accessed January 17, 2014].

[115] Gero Mühl, Ludger Fiege, and Peter R. Pietzuch. *Distributed event-based systems*. Springer, 2006. ISBN 978-3-5403-2651-9.

[116] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. Wiley, 3rd edition, 2011. ISBN 1-118-03196-2.

[117] Srini Narayanan and Sheila McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pages 77–88, New York, NY, USA, 2002. ACM. doi:10.1145/511446.511457.

[118] OASIS. Topology and orchestration specification for cloud applications TC, 2014. https://www.oasis-open.org/committees/tosca/. [Online; accessed January 17, 2014].

[119] OpenStack Foundation. OpenStack Heat, 2014. https://wiki.openstack.org/wiki/Heat. [Online; accessed January 17, 2014].

[120] OpsCode, Inc. Chef, 2014. http://opscode.com/chef. [Online; accessed January 17, 2014].

[121] Stephen R. Palmer and John M. Felsing. *A Practical Guide to Feature-Driven Development*. The Coad Series. Prentice Hall PTR, 2002. ISBN 9780130676153.

[122] Michael P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12, 2003. doi:10.1109/WISE.2003.1254461.

[123] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007. doi:10.1109/MC.2007.400.

[124] Xin Peng, Bihuan Chen, Yijun Yu, and Wenyun Zhao. Self-tuning of software systems through goal-based feedback loop control. In *Proceedings of the 18th IEEE International Requirements Engineering Conference*, RE '10, pages 104–107, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/RE.2010.22.

[125] Tan Phan, Jun Han, Jean-Guy Schneider, Tim Ebringer, and Tony Rogers. A survey of policy-based management approaches for service oriented systems. In *Proceedings of the 19th Australian Conference on Software Engineering*, ASWEC '08, pages 392–401, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/ASWEC.2008.4483228.

[126] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 49–61, Washington, DC, USA, 2006. IEEE Computer Society. doi:10.1109/ICDE.2006.105.

[127] PuppetLabs, Inc. Puppet, 2014. http://puppetlabs.org/. [Online; accessed January 17, 2014].

[128] John Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986. doi:10.1007/BF00116251.

[129] John Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., 1993. ISBN 978-1-5586-0238-0.

[130] Václav T. Rajlich. Changing the paradigm of software engineering. *Communications of the ACM*, 49(8):67–70, 2006. doi:10.1145/1145287.1145289.

[131] Václav T. Rajlich and Keith H. Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 2000. doi:10.1109/2.869374.

[132] Manos Renieres and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE '03, pages 30–39, New York, NY, USA, 2003. IEEE Computer Society. doi:10.1109/ASE.2003.1240292.

[133] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC*, pages 44–51, Los Alamitos, CA, USA, November 2009. IEEE. doi:10.1109/NCM.2009.218.

[134] Winston W. Royce. Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th International Conference on Software Engineering (Reprint)*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press. ISBN 0-89791-216-0. acmid:41801. Originally published in Proceedings of IEEE WESTCON, August 1970.

[135] Stuart Jonathan Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2010. ISBN 978-0-1360-4259-4.

[136] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14:1–14:42, May 2009. doi:10.1145/1516533.1516538.

[137] Farshad A. Samimi, Philip K. McKinley, S. Masoud Sadjadi, Chiping Tang, Jonathan K. Shapiro, and Zhinan Zhou. Service clouds: Distributed infrastructure for adaptive communication services. *IEEE Transactions on Network and Service Management*, 4(2):84–95, 2007. doi:10.1109/TNSM.2007.070901.

[138] Benjamin Satzger and Oliver Kramer. Goal distance estimation for automated planning using neural networks and support vector machines. *Natural Computing*, 12(1):87–100, 2013. doi:10.1007/s11047-012-9332-y.

[139] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. A new adaptive accrual failure detector for dependable distributed systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 551–555, New York, NY, USA, 2007. ACM. doi:10.1145/1244002.1244129.

[140] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler, and Theo Ungerer. Using automated planning for trusted self-organising organic computing systems. In *Proceedings of the 5th International Conference on Autonomic and Trusted Computing*, ATC '08, pages 60–72, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-69295-9_7.

[141] Benjamin Satzger, Waldemar Hummer, Christian Inzinger, Philipp Leitner, and Schahram Dustdar. Winds of change: From vendor lock-in to the meta cloud. *IEEE Internet Computing*, 17(1):69–73, 2013. doi:10.1109/MIC.2013.19.

[142] Björn Schilling, Boris Koldehofe, and Kurt Rothermel. Efficient and distributed rule placement in heavy constraint-driven event systems. In *Proceedings of the 13th IEEE International Conference on High Performance Computing and Communications*, HPCC '11, pages 355–364, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/HPCC.2011.53.

[143] Ingo Schnabel and Markus Pizka. Goal-driven software development. In *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, SEW '06, pages 59–65, Washington, DC, USA, 2006. IEEE Computer Society. doi:10.1109/SEW.2006.21.

[144] Toby Segaran. *Programming Collective Intelligence*. O'Reilly Media, 2007. ISBN 978-0-5965-2932-1.

[145] Liwei Shen, Xin Peng, and Wenyun Zhao. Quality-driven self-adaptation: Bridging the gap between requirements and runtime architecture by design decision. In *Proceedings of the 36th Annual IEEE Computer Software and Applications Conference*, COMPSAC '12, pages 185–194, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/COMPSAC.2012.29.

[146] James Skene, D Davide Lamanna, and Wolfgang Emmerich. Precise service level agreements. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04. IEEE Computer Society, May 2004. acmid:998675.999422.

[147] Morris Sloman. Policy driven management for distributed systems. *Journal of Network and Systems Management*, 2(4):333–360, 1994. doi:10.1007/BF02283186.

[148] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for in-network stream query processing. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 250–258, New York, NY, USA, 2005. ACM. doi:10.1145/1065167.1065199.

[149] Małgorzata Steinder and Adarshpal S. Sethi. Probabilistic fault localization in communication systems using belief networks. *IEEE/ACM Transactions on Networking*, 12(5): 809–822, 2004. doi:10.1109/TNET.2004.836121.

[150] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the 7th International Conference on Machine Learning*, pages 216–224, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc. ISBN 1-55860-141-4.

[151] Hirotaka Takeuchi and Ikujiro Nonaka. The new new product development game. *Harvard business review*, 64(1):137–146, 1986.

[152] Gerald Tesauro, Nicholas K Jong, Rajarshi Das, and Mohamed N Bennani. On the use of hybrid reinforcement learning for autonomic resource allocation. *Cluster Computing*, 10 (3):287–299, 2007. doi:10.1007/s10586-007-0035-6.

[153] TM Forum. Case study handbook, December 2009.

[154] TM Forum. eTOM Business Process Framework, 2014. http://www.tmforum.org/BusinessProcessFramework/1647/home.html. [Online; accessed January 17, 2014].

[155] Huy Tran, Uwe Zdun, and Schahram Dustdar. View-based and model-driven approach for reducing the development complexity in process-driven SOA. In *Proceedings of the 1st International Working Conference on Business Process and Services Computing*, BPSC '07, pages 105–124, Bonn, 2007. GI. ISBN 978-3-88579-210-9.

[156] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer, Berlin, Heidelberg, April 2011. ISBN 978-3-6421-9344-6.

[157] Christopher J.C.H. Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4): 279–292, 1992. doi:10.1007/BF00992698.

[158] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991. doi:10.1109/32.83906.

[159] W. Eric Wong and Vidroha Debroy. Software fault localization. *Part of the IEEE Reliability Society 2009 Annual Technology Report*, 2009. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.172.202.

[160] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 791–802, Washington, DC, USA, 2005. IEEE Computer Society. doi:10.1109/ICDE.2005.53.

[161] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd international conference on Very large data bases*, VLDB '06, pages 775–786. VLDB Endowment, 2006. acmid:1164194.

[162] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Proceedings of the Grid Computing Environments Workshop*, GCE '08, pages 1–10, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/GCE.2008.4738443.

[163] Chuanzhen Zang and Yushun Fan. Complex event processing in enterprise information systems based on RFID. *Enterprise Information Systems*, 1(1):3–23, 2007. doi:10.1080/17517570601092127.

[164] Ji Zhang and Betty H.C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 371–380, New York, NY, USA, 2006. ACM. doi:10.1145/1134285.1134337.

[165] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010. doi:10.1007/s13174-010-0007-6.

[166] Pin Zhou, Binny Gill, Wendy Belluomini, and Avani Wildani. GAUL: Gestalt analysis of unstructured logs for diagnosing recurring problems in large enterprise storage systems. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 148–159, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/SRDS.2010.25.

[167] Olaf Zimmermann, Uwe Zdun, Thomas Gschwind, and Frank Leymann. Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. In *Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture*, WICSA '08, pages 157–166, Washington, DC, USA, 2008. IEEE Computer Society. doi:10.1109/WICSA.2008.19.

[168] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. Predicting bugs from history. In *Software Evolution* Mens and Demeyer [111], pages 69–88. ISBN 978-3-540-76440-3. doi:10.1007/978-3-540-76440-3_4.

APPENDIX A

# Glossary

**ADSL** Asymmetric Digital Subscriber Line.
**AI** Artificial Intelligence.
**AMQP** Advanced Message Queueing Protocol.
**AOP** Aspect-Oriented Programming.
**API** Application Programming Interface.
**ARTS** Association for Retail Technology Standards, http://nrf-arts.org/.

**BPMN** Business Process Modeling Notation.

**CEP** Complex Event Processing.
**CM** Configuration Management.
**CORBA** Common Object Request Broker Architecture.
**CRM** Customer Relationship Management.
**CRS** Car Reservation Service.

**DaaS** Data as a Service.
**DRL** Drools Rule Language, http://docs.jboss.org/drools/.
**DSL** Domain-Specific Language.

**EBNF** Extended Backus-Naur Form.
**EC2** Elastic Compute Cloud.
**ECA** Event-Condition-Action.
**EPL** Esper Event Processing Language, http://esper.codehaus.org/.
**eTOM** enhanced Telecom Operations Map.

**FDD** Feature Driven Development.
**FMS** Financial Management System.
**FQDN** Fully-Qualified Domain Name.

**GDS** Global Distribution System.
**GUI** Graphical User Interface.

**HRS** Hotel Reservation Service.

**IaaS** Infrastructure as a Service.
**IATA** International Air Transport Association, http://www.iata.org/.
**IPTV** Internet Protocol television.

**JMS** Java Messaging Service.

**MAPE** Monitor, Analyze, Plan, Execute.
**MDP** Markov Decision Process.
**MOM** Message-Oriented Middleware.
**MONINA** MONitoring, INtegration, and Adaptation.
**MTOSI** Multi-Technology Operations System Interface.

**PaaS** Platform as a Service.

**QoS** Quality of Service.

**RDBMS** Relational Database Management System.
**RDS** Relational Database Service.
**REST** REpresentational State Transfer.
**RMI** Remote Method Invocation.
**RUP** Rational Unified Process.

**S3** Simple Storage Service.
**SaaS** Software as a Service.
**SBA** Service-Based Application.
**SBS** Service-Based System.
**SCA** Service Component Architecture.
**SLA** Service Level Agreement.
**SLO** Service Level Objective.
**SOA** Service Oriented Architecture.
**SOAP** Simple Object Access Protocol (originally).
**SQS** Simple Queue Service.
**SVN** Service Value Network.

**TMF** TM Forum, http://tmforum.com/.

**URI** Uniform Resource Identifier.

**VbMF** View-based Modeling Framework.
**VM** Virtual Machine.
**VoIP** Voice over IP.
**VSP** Virtual Service Platform.

**XP** Extreme Programming.

APPENDIX B

# MONINA Language Grammar

In the following, we show the complete EBNF specification for the MONINA language, as implemented in the current version of the MONINA Editor Eclipse plugin. Documentation on how to install the plugin is available at http://indenicatuv.github.io/releases. The plugin source code can be inspected at https://github.com/inz/monina, and the runtime components used by the plugin for deploying the specified application management infrastructures is available at https://github.com/inz/indenica-runtime-core.

⟨*monina-root*⟩ ::= ⟨*abstr-elem*⟩*

⟨*abstr-elem*⟩ ::= ⟨*pkg-decl*⟩
| ⟨*import*⟩
| ⟨*component*⟩
| ⟨*event*⟩
| ⟨*action*⟩
| ⟨*query*⟩
| ⟨*fact*⟩
| ⟨*host*⟩
| ⟨*rule*⟩

⟨*pkg-decl*⟩ ::= '`package`' ⟨*qualified-name*⟩ '`{`' ⟨*abstr-elem*⟩* '`}`'

⟨*import*⟩ ::= '`import`' ⟨*qualified-name-w*⟩

⟨*component*⟩ ::= '`component`' ⟨*ID*⟩
'`{`' ⟨*comp-metadata*⟩ ⟨*refs*⟩* ⟨*host-ref*⟩? ⟨*endpoint*⟩* '`}`'

⟨*comp-metadata*⟩ ::= ('`vendor`' ⟨*STRING*⟩)?
('`version`' ⟨*STRING*⟩)?
('`description`' ⟨*STRING*⟩)?

⟨*endpoint*⟩ ::= ‘`endpoint`’ ⟨*ID*⟩? ‘`{`’ ‘`at`’ ⟨*endpoint-addr*⟩ ⟨*refs*⟩* ‘`}`’

⟨*endpoint-addr*⟩ ::= ⟨*STRING*⟩
(‘`on`’ ⟨*host-ref*⟩)?
(⟨*host-port*⟩)?
(‘`using`’ ⟨*STRING*⟩)?
(‘`with`’ ⟨*STRING*⟩)?

⟨*refs*⟩ ::= ⟨*event-ref*⟩ | ⟨*action-ref*⟩

⟨*host-ref*⟩ ::= ‘`host`’ ⟨*qualified-name*⟩

⟨*event-ref*⟩ ::= ⟨*qualified-name*⟩ ⟨*freq*⟩?

⟨*action-ref*⟩ ::= ‘`action`’ ⟨*qualified-name*⟩

⟨*event*⟩ ::= ‘`event`’ ⟨*qualified-name*⟩ ‘`{`’ ⟨*e-attr*⟩* ‘`}`’

⟨*action*⟩ ::= ‘`action`’ ⟨*qualified-name*⟩ ‘`{`’ ⟨*e-attr*⟩* ‘`}`’

⟨*e-attr*⟩ ::= ⟨*ID*⟩ ‘`:`’ ⟨*qualified-name*⟩

⟨*freq*⟩ ::= ‘`every`’ ⟨*Number*⟩ ⟨*t-unit*⟩ | ⟨*Number*⟩ ‘`Hz`’

⟨*host*⟩ ::= ‘`host`’ ⟨*ID*⟩ ‘`{`’ ⟨*host-addr*⟩? ⟨*host-port*⟩? ⟨*host-capacity*⟩? ‘`}`’

⟨*host-addr*⟩ ::= ‘`address`’ ⟨*STRING*⟩

⟨*host-port*⟩ ::= ‘`port`’ ⟨*INT*⟩

⟨*host-capacity*⟩ ::= ‘`capacity`’ ⟨*INT*⟩

⟨*query*⟩ ::= ⟨*simple-query*⟩ | ⟨*esper-query*⟩

⟨*simple-query*⟩ ::= ‘`query`’ ⟨*ID*⟩ ‘`{`’
(⟨*source-decl*⟩ | ⟨*emit-decl*⟩)*
⟨*window-decl*⟩? ⟨*cond-decl*⟩?
⟨*cost*⟩? ⟨*io-ratio*⟩?
‘`}`’

⟨*esper-query*⟩ ::= ‘`equery`’ ⟨*ID*⟩ ‘`{`’ ⟨*source-decl*⟩+ ⟨*STRING*⟩ ‘`}`’

⟨*source-decl*⟩ ::= ‘`from`’ ⟨*source*⟩ (‘`,`’ ⟨*source*⟩)*

⟨*source*⟩ ::= (‘`source`’ | ‘`sources`’) ⟨*qualified-name*⟩ (‘`,`’ ⟨*qualified-name*⟩)*
(‘`event`’ | ‘`events`’) ⟨*qualified-name*⟩ (‘`,`’ ⟨*qualified-name*⟩)*
(‘`as`’ ⟨*ID*⟩)?

⟨*emit-decl*⟩ ::= ‘`emit`’ ⟨*qualified-name*⟩
‘`(`’ ⟨*attr-emit-decl*⟩ (‘`,`’ ⟨*attr-emit-decl*⟩)* ‘`)`’

⟨*attr-emit-decl*⟩ ::= ⟨*cond-expr*⟩ (‘`as`’ ⟨*qualified-name*⟩)?

⟨*window-decl*⟩ ::= ‘`window`’ ⟨*window-expr*⟩

⟨*window-expr*⟩ ::= ⟨*batch-window*⟩ | ⟨*time-window*⟩

⟨*batch-window*⟩ ::= ⟨*INT*⟩ (‘`event`’ | ‘`events`’)?

⟨*time-window*⟩ ::= ⟨*INT*⟩ ⟨*time-unit*⟩

⟨*time-unit*⟩ ::= ‘`s`’ | ‘`sec`’ | ‘`second`’ | ‘`seconds`’
| ‘`m`’ | ‘`min`’ | ‘`minute`’ | ‘`minutes`’
| ‘`h`’ | ‘`hour`’ | ‘`hours`’
| ‘`d`’ | ‘`day`’ | ‘`days`’
| ‘`M`’ | ‘`month`’ | ‘`months`’
| ‘`y`’ | ‘`year`’ | ‘`years`’

⟨*cond-decl*⟩ ::= ‘`where`’ ⟨*cond-expr*⟩

⟨*cost*⟩ ::= ‘`cost`’ ⟨*Number*⟩

⟨*io-ratio*⟩ ::= ‘`ratio`’ ⟨*Number*⟩

⟨*fact*⟩ ::= ⟨*ID*⟩? ‘`{`’ ⟨*source-decl*⟩ ⟨*part-key*⟩? ‘`}`’

⟨*part-key*⟩ ::= ‘`by`’ ⟨*qualified-name*⟩

⟨*rule*⟩ ::= ⟨*simple-rule*⟩ | ⟨*drools-rule*⟩

⟨*simple-rule*⟩ ::= ‘`rule`’ ⟨*ID*⟩ ‘`{`’ ⟨*rule-source*⟩+ ⟨*stmt*⟩+ ⟨*cost*⟩? ‘`}`’

⟨*drools-rule*⟩ ::= ‘`drule`’ ⟨*ID*⟩ ‘`{`’ ⟨*rule-source*⟩+ ⟨*STRING*⟩ ‘`}`’

⟨*rule-source*⟩ ::= ‘`from`’ ⟨*qualified-name*⟩ (‘`as`’ ⟨*ID*⟩)?

⟨*stmt*⟩ ::= ‘`when`’ ⟨*cond-expr*⟩ ‘`then`’ ⟨*action-expr*⟩

⟨*action-expr*⟩ ::= ⟨*qualified-name*⟩ ⟨*qualified-name*⟩
(‘`(`’ ⟨*attr-emit-decl*⟩ (‘`,`’ ⟨*attr-emit-decl*⟩)* ‘`)`’)?

⟨*cond-expr*⟩ ::= ⟨*cond-or-expr*⟩

⟨*cond-or-expr*⟩ ::= ⟨*cond-and-expr*⟩ (⟨*or-op*⟩ ⟨*cond-and-expr*⟩)*

⟨*or-op*⟩ ::= ‘`||`’ | ‘`or`’ | ‘`OR`’

⟨*cond-and-expr*⟩ ::= ⟨*eq-expr*⟩ (⟨*and-op*⟩ ⟨*eq-expr*⟩)*

⟨*and-op*⟩ ::= '`&&`' | '`and`' | '`AND`'

⟨*eq-expr*⟩ ::= ⟨*rel-expr*⟩ (⟨*eq-op*⟩ ⟨*rel-expr*⟩)*

⟨*eq-op*⟩ ::= '=' | '`!=`'

⟨*rel-expr*⟩ ::= ⟨*add-expr*⟩ (⟨*rel-op*⟩ ⟨*add-expr*⟩)*

⟨*rel-op*⟩ ::= '<' | '<=' | '>=' | '>'

⟨*add-expr*⟩ ::= ⟨*mult-expr*⟩ (⟨*add-op*⟩ ⟨*mult-expr*⟩)*

⟨*add-op*⟩ ::= '+' | '−'

⟨*mult-expr*⟩ ::= ⟨*unary-expr*⟩ (⟨*mult-op*⟩ ⟨*unary-expr*⟩)*

⟨*mult-op*⟩ ::= '`*`' | '`/`' | '`%`'

⟨*unary-expr*⟩ ::= ⟨*unary-op*⟩? ⟨*primary-expr*⟩

⟨*unary-op*⟩ ::= '−' | '`not`' | '`!`'

⟨*primary-expr*⟩ ::= ⟨*par-expr*⟩ | ⟨*literal*⟩ | ⟨*feature-call*⟩

⟨*par-expr*⟩ ::= '`(`' ⟨*cond-expr*⟩ '`)`'

⟨*feature-call*⟩ ::= ⟨*qualified-name*⟩

⟨*literal*⟩ ::= ⟨*bool-lit*⟩ | ⟨*number-lit*⟩ | ⟨*null-lit*⟩ | ⟨*string-lit*⟩

⟨*bool-lit*⟩ ::= '`false`' | '`true`'

⟨*number-lit*⟩ ::= ⟨*Number*⟩

⟨*null-lit*⟩ ::= '`null`'

⟨*string-lit*⟩ ::= ⟨*STRING*⟩

⟨*qualified-name*⟩ ::= ⟨*ID*⟩ ('`.`' ⟨*ID*⟩)*

⟨*qualified-name-w*⟩ ::= ⟨*qualified-name*⟩ '`.*`'?

⟨*ID*⟩ ::= ('`a`'..'`z`'|'`A`'..'`Z`'|'`_`') ('`a`'..'`z`'|'`A`'..'`Z`'|'`_`'|'`0`'..'`9`')*

⟨*INT*⟩ ::= '`0`'..'`9`' ('`0`'..'`9`')*

⟨*Number*⟩ ::= ⟨*INT*⟩ ('`.`' ⟨*INT*⟩)?

APPENDIX C

# Curriculum Vitae

Christian Inzinger
Gruschaplatz 2/9
1140 Wien, Austria

Born May 20, 1982
Email inzinger@dsg.tuwien.ac.at
Web dsg.tuwien.ac.at/staff/inzinger

## Experience

| | |
|---|---|
| Researcher at the Distributed Systems Group<br>Vienna University of Technology<br>http://dsg.tuwien.ac.at/ | since 2011 |
| CTO, Blackwhale GmbH | 2008–2010 |
| Consultant, Freelance<br>PVM Data Services GmbH (http://pvmoil.com)<br>convisio GmbH (http://convisio.at)<br>ilogs GmbH (http://ilogs.com) | since 2004 |
| CTO, Nexovis GmbH (http://nexovis.com/) | 2002–2004 |
| Software Engineer, LKH Villach (http://lkh-vil.or.at/) | 2000–2001 |

## Education

| | |
|---|---|
| Ph.D. in Computer Science at the Distributed Systems Group,<br>Vienna University of Technology | 2014 |

Dipl.Ing. (M.Sc.) in Software Engineering & Internet Computing, Vienna University of Technology — 2010

Bakk.techn. (B.Sc.) in Software & Information Engineering, Vienna University of Technology — 2006

## Publications

- Christian Inzinger, Stefan Nastic, Sanjin Sehic, Michael Vögler, Fei Li, and Schahram Dustdar. MADCAT – A methodology for architecture and deployment of cloud application topologies. In *Proceedings of the 8th International Symposium on Service-Oriented System Engineering*, SOSE '14, Washington, DC, USA, 2014. IEEE Computer Society. (to appear)

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems. *Software: Practice and Experience*, 2014. doi:10.1002/spe.2254. (to appear)

- Christian Inzinger, Waldemar Hummer, Ioanna Lytra, Philipp Leitner, Huy Tran, Uwe Zdun, and Schahram Dustdar. Decisions, models, and monitoring – A lifecycle model for the evolution of service-based systems. In *Proceedings of the 17th IEEE International Enterprise Distributed Object Computing Conference*, EDOC '13, pages 185–194, Washington, DC, USA, 2013. IEEE Computer Society. doi:10.1109/EDOC.2013.29

- Christian Inzinger, Benjamin Satzger, Philipp Leitner, Waldemar Hummer, and Schahram Dustdar. Model-based adaptation of cloud computing applications. In *Proceedings of the International Conference on Model-Driven Engineering and Software Development (MODELSWARD '13), Special Track on Model-driven Software Adaptation*, MODA '13, pages 351–355. SciTePress, 2013. doi:10.5220/0004381803510355

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Identifying incompatible service implementations using pooled decision trees. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 485–492, New York, NY, USA, 2013. ACM. doi:10.1145/2480362.2480456

- Benjamin Satzger, Waldemar Hummer, Christian Inzinger, Philipp Leitner, and Schahram Dustdar. Winds of change: From vendor lock-in to the meta cloud. *IEEE Internet Computing*, 17(1):69–73, 2013. doi:10.1109/MIC.2013.19

- Philipp Leitner, Christian Inzinger, Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. Application-level performance monitoring of cloud services based on the complex event processing paradigm. In *Proceedings of the 5th IEEE International Conference on Service-Oriented Computing and Applications*, SOCA '12, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/SOCA.2012.6449437

- Christian Inzinger, Benjamin Satzger, Waldemar Hummer, and Schahram Dustdar. Specification and deployment of distributed monitoring and adaptation infrastructures. In *Proceedings of the International Workshop on Performance Assessment and Auditing in Service Computing, co-located with ICSOC '12*, PAASC '12, pages 167–178, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-37804-1_18

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Towards identifying root causes of faults in service-based applications. In *Proceedings of the 31st IEEE International Symposium on Reliable Distributed Systems*, SRDS '12, pages 404–405, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/SRDS.2012.78

- Waldemar Hummer, Christian Inzinger, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. Deriving a unified fault taxonomy for event-based systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 167–178, New York, NY, USA, 2012. ACM. doi:10.1145/2335484.2335504

- Philipp Leitner, Waldemar Hummer, Benjamin Satzger, Christian Inzinger, and Schahram Dustdar. Cost-efficient and application sla-aware client side request scheduling in an infrastructure-as-a-service cloud. In *Proceedings of the 5th IEEE International Conference on Cloud Computing*, CLOUD '12, pages 213–220, Washington, DC, USA, 2012. IEEE Computer Society. doi:10.1109/CLOUD.2012.21

- Christian Inzinger, Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Non-intrusive policy optimization for dependable and adaptive service-oriented systems. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 504–510, New York, NY, USA, 2012. ACM. doi:10.1145/2245276.2245373

- Christian Inzinger, Waldemar Hummer, Benjamin Satzger, Philipp Leitner, and Schahram Dustdar. Identifying incompatible implementations of industry standard service interfaces for dependable service-based applications. Technical Report TUV-1841-2012-1, Vienna University of Technology, 2012

- Philipp Leitner, Benjamin Satzger, Waldemar Hummer, Christian Inzinger, and Schahram Dustdar. CloudScale – a novel middleware for building transparently scaling cloud applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 434–440, New York, NY, USA, 2012. ACM. doi:10.1145/2245276.2245360

- Waldemar Hummer, Benjamin Satzger, Philipp Leitner, Christian Inzinger, and Schahram Dustdar. Distributed continuous queries over web service event streams. In *Proceedings of the 7th International Conference on Next Generation Web Services Practices*, NWeSP '11, pages 176–181, Washington, DC, USA, 2011. IEEE Computer Society. doi:10.1109/NWeSP.2011.6088173