FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Holistic Monitoring and Domain Specific Adaptation in Composite Services

DISSERTATION

zur Erlangung des akademischen Grades

Doktor/in der technischen Wissenschaften

eingereicht von

Oliver Moser
Matrikelnummer 0126303

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar

Diese Dissertation haben begutachtet:

(Univ.Prof. Dr. Schahram Dustdar)	(Univ.Prof. Dr. Frank Leymann)

Wien, 01.05.2012

(Oliver Moser)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Holistic Monitoring and Domain Specific Adaptation in Composite Services

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor/in der technischen Wissenschaften

by

Oliver Moser
Registration Number 0126303

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar

The dissertation has been reviewed by:

(Univ.Prof. Dr. Schahram Dustdar)	(Univ.Prof. Dr. Frank Leymann)

Wien, 01.05.2012

(Oliver Moser)

Erklärung zur Verfassung der Arbeit

Oliver Moser
Medwedweg 11/4.22, 1110 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 Verfasserin)

Danksagung

Der Abschluss eines Doktoratsstudiums ist mit Sicherheit ein persönlicher Meilenstein. Da an der Erreichung dieses Meilensteins nicht nur meine Wenigkeit beteiligt war, möchte ich an dieser Stelle einigen Personen meinen Dank aussprechen.

Zunächst gilt mein Dank meinem Betreuer Prof. Schahram Dustdar. Prof. Dustdar hat mir viel Inspiration mit auf den Weg zum Doktorat gegeben, und hatte wertvolle Kommentar zum Inhalt dieser Dissertation.

Ebenso möchte ich mich bei Prof. Frank Leymann bedanken, der trotz seines überbuchten Kalenders Zeit gefunden hat, die Zweitbegutachtung der Arbeit zu übernehmen. An dieser Stelle möchte ich auch Stefan Mitterhofer nicht unerwähnt lassen. Er hat dankenswerter Weise die Arbeit Korrektur gelesen und wichtigen Input für die Korrekturphase geliefert.

Ganz besonders möchte ich mich bei Florian Rosenberg bedanken. Die Zusammenarbeit mit ihm hat mich um viele Ideen reicher gemacht. Weiters hat Florian wichtige Beiträge zu den Publikationen geliefert, auf Basis derer diese Dissertation entstanden ist.

Auch meinen Eltern möchte ich herzlich danken. Ohne die Tugenden, die ich durch meinen Vater Helmut und meine Mutter Walpurga vermittelt bekommen habe, würde ich heute diese Zeilen nicht schreiben. Selbstverständlich möchte ich auch meiner Schwester Kathrin danken, die gerade selbst vor ihrem Studienabschluss steht. Sieh diese Zeilen als Motivation an, das Ziel ist in Sicht!

Der meiste Dank gilt allerdings Birgit Klein-Reiter. Birgit hat mich in allen Situationen und Lebenslagen unterstützt und bekräftigt, selbst dann, wenn die gemeinsame Zeit dadurch beschränkt wurde. Durch ihre verständnisvolle Art, aber auch durch ihren Humor, hat Birgit es mir ermöglicht, mein Doktoratsstudium neben einer Vollzeitbeschäftigung zum Abschluss zu bringen. Meine liebe Birgit – Schön, dass es dich gibt, und dass Du der Mensch an meiner Seite bist!

Abstract

Service orientation emerged as an ubiquitous paradigm to tackle large scale software systems. In this regard, service oriented systems represent application functionalities and responsibilities by the means of loosely coupled, computational elements, called services. Services can be combined in a structured fashion, resulting in composite services that are used to align enterprise business processes to information technology. Hence, composite services represent a paramount concern in today's enterprise landscapes. Quality of Service (QoS) provides a means to quantify certain performance and dependability related metrics that are crucial for the normal operation of composite services, in addition to various business related concerns. Therefore, monitoring QoS and business related concerns of composite services is an important challenge in service oriented systems. Moreover, such a QoS based quantification can be leveraged to dynamically adapt composite services to changing requirements. Runtime service selection represents one possibility for such dynamic adaptations. QoS can be used to determine slow and unreliable services that affect the overall performance. Consequently, if alternative service instances are available and provide better QoS, a bad performing service can be substituted with a better performing alternative.

This thesis addresses the challenges of managing monitoring and adaptation requirements of composite services. Our contributions are threefold. First, we introduce an adaptive QoS model to tackle short term requirements with regard to additional QoS attributes. In this model, we combine both domain-agnostic and domain-specific QoS attributes. Second, we propose an approach for managing holistic monitoring requirements in composite services. Our approach represents the message exchange within a composite service as a stream of events, and leverages event stream processing technology to address QoS and business related monitoring requirements. Additionally, we treat data that is created outside the scope of a composite service as first class citizens. This enables us to provide a holistic view on all monitoring data that is available in enterprise environments. Third, we present a method for runtime service selection that leverages our adaptive QoS model. For specifying the service selection strategy, we propose a domain-specific language. This language can be used to specify selector implementations based on the available QoS attributes. A genuine case study from the telecommunication domain is used to derive the requirements for such a monitoring and service selection framework. Finally, we provide a proof of concept implementation of our approach for WS-BPEL. This prototype is further used for a performance and qualitative validation of our method.

Kurzfassung

Dienst-Orientierte Systeme teilen die Verantwortlichkeiten innerhalb eines Software Systems in lose gekoppelte und durch definierte Schnittstellen ansprechbare Entititäten, genannt *Dienste* (engl. *Services*). Diese Dienste stellen jeweils eine bestimmte Menge an zusammenhängenden Funktionalitäten dar. Dienste können strukturiert zusammengeschalten werden um sogenannte *Zusammengesetzte Dienste* (engl. *composite services*) zu erstellen, mit Hilfe derer unternehmensweite Geschäftsprozesse abgebildet und automatisiert werden können. Sowohl einzelne Dienste als auch Zusammengesetzte Diensten zeichnen sich durch ihre *Dienstgüte* (engl. *Quality of Service, QoS*) aus. Um diese Art der Quantifizierung zu bestimmen müssen unterschiedliche Parameter gemessen werden, beispielsweise die Antwortzeiten oder Verfügbarkeiten. Diese Parameter haben daher direkten Einfluss auf das Laufzeit Verhalten eines Dienstes. Im Zusammenhang mit Zusammengesetzten Diensten beeinflusst die Dienstgüte der einzelnen, zusammengeschaltenen Dienste auch die Dienstgüte des daraus resultierenden Zusammengesetzten Dienstes. Neben diesen technischen Dienstgüte Parametern müssen geschäfts-relevante Parameter berücksichtigt werden, beispielsweise die Anzahl an Vertragsaktivierungen innerhalb der Geschäftsstunden in einem Telekom Unternehmen. Es ist wichtig, Schwankungen in den Werten dieser Parameter frühzeitig zu erkennen und entsprechend zu behandeln. Eine Möglichkeit um mit diesen Schwankungen umzugehen ist die Ersetzung von Diensten mit suboptimaler Dienstgüte durch funktional äquivalente, alternative Dienste. Findet die Feststellung der hinsichtlich ihres Laufzeitverhaltens problematischen Dienste, ebenso wie deren Ersetzung durch funktional äquivalente Dienste, zur Ausführungszeit des Zusammengesetzten Dienstes statt, können potentielle Probleme verhindert und die Dienstgüte des Zusammengesetzten Dienstes positiv beeinflusst werden.

Die vorliegende Arbeit behandelt die Anforderungen an ein System, das die oben beschriebenen Messungen und Anpassungen an Zusammengesetzten Diensten zu deren Ausführungszeit vornehmen kann. Unsere Arbeit besteht im Wesentlichen aus drei Eckpfeilern: (1) einem erweiterbaren Modell um Dienstgüte von Zusammengesetzten Diensten darzustellen, (2) einem ereignisgesteuerten System zur Messung von Dienstgüte und geschäftsrelevanten Dienstparametern, und (3) einer Domänen-spezifischen Methode um Dienste eines Zusammengesetzten Dienstes zu dessen Ausführungszeit zu ersetzen. Die spezifischen Anforderungen an die Methoden werden an Hand eines Fallbeispiels aus dem Telekom Umfeld erarbeitet und validiert.

The question of whether computers can think is like the question of whether submarines can swim. (Edsger Wybe Dijkstra, 1984)

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

In 2010, Finland enshrined the legal right to broadband internet access for all citizens [132, 30]. This law guarantees a one-megabit internet connection to every person in Finland. By putting broadband internet access on the same level as electricity and water supply, Finland is clearly emphasizing on the importance of broadband internet access in modern society. Considering that internet technologies have superseded many traditional media, enable new ways of communication, and support citizens in their governmental concerns, the reasoning behind this move becomes obvious.

From an economical point of view, such developments impose high demands on various industry sectors, most evident on internet service providers (ISP). However, not only ISPs, but also providers of internet applications, such as Amazon, Google or Facebook, are facing increasing demands. These enterprises are servicing thousands of users per second. Moreover, not only the number of concurrent users is constantly increasing, but also the user expectations that have to be met. Many commercial users leverage social media to represent their company via Facebook or sell goods via Amazon Marketplace. Hence, the unavailability of such applications can have a tremendous impact on a business, ranging from bad reputation to substantial monetary loss.

From a technical perspective, there are several aspects that need consideration when designing systems that support such large scale, mission critical enterprise applications. One aspect is the ever increasing system complexity. A common way to tackle this application complexity is by the means of modularization. David Parnas' important paper *On the Criteria to Be Used in Decomposing Systems into Modules* [127] defines modularizations as *partial system descriptions* where a *module is considered to be a responsibility assignment*. Putting it another way, the fundamental idea behind modularization is to dissect application functionality into more manageable entities that serve a certain purpose.

In recent years, a novel methodology for building large scale software systems gained increased adoption. *Service orientation* reifies the concept of modularization by the means of *services* - computational elements that serve a well defined purpose. Such services provide several aspects that are crucial when building large scale, distributed and mission critical applications, often interacting with a variety of heterogeneous systems [121]. Amongst other principles, ser-

vices promote loose coupling [88], interaction via well defined service contracts [158], reusability and platform independence [147]. From a conceptual point of view, service orientation is a design paradigm to create *service oriented computing* (SOC) systems, and is manifested in an architectural style called *service oriented architecture* (SOA) [53]. SOA has a high adoption rate in enterprise landscapes [7], since it enables enterprises to expose their core business logic via services, which can be consumed both inter-organizationally as well as by external business partners.

Typically, contemporary enterprise applications rely on a number of inter-organizational and mission critical business processes [74]. Service orientation provides the means to model such process-driven applications by combining services in a structured fashion. This service orchestration delivers *composite services* that effectively align information technology to enterprise business processes. Composite services allow enterprises to reuse common building blocks of business logic that are represented by services. Furthermore, it enables timely reaction to restructurings of business processes due to the loosely coupled nature of service oriented systems. The composition of services is usually controlled by the means of a central entity, referred to as the *composition engine*. Related technologies, such as the Web service technology stack for implementing services, as well as WS-BPEL, the de-facto standard for implementing composite services based on Web services technology, are widely accepted in both industry and academia. Taken together, the concept of service orientation, and in particular, leveraging its benefits by the means of composite services, is an integral part of today's information technology landscapes.

1.1 Problem Statement

As outlined above, service orientation supports enterprises in tackling rapidly changing market requirements by the means of composite services. However, adopting service orientation as a design paradigm alone cannot solve all issues apparent in inter-organizational settings. A key challenge for a successful SOA adoption, and particularly important for service composition infrastructures, is to build mechanisms for supporting composite services with self-adaptivity and dependability [48, 167]. A self-adaptive and dependable composite service is capable of reacting to changes in the environment [118] and has the ability to deliver services that can justifiably be trusted [19].

Quality of Service (QoS) is an important aspect of modeling and managing self-adaptive and dependable composite services and their individual services [140]. Dependability and performance properties of Web services can be categorized and measured by using QoS attributes from different providers [100, 97]. In the service composition context, the QoS of the overall composite service is particularly important. However, the QoS of the individual services usually determines the QoS of the overall composition [85]. Thus, a single service with bad QoS may cause deteriorating performance or failure of the composite service. Additional to these technical performance and dependability indicators, certain business related data might be relevant for Key Performance Indicator (KPI) progression or Service Level Agreement (SLA) fulfillment [154].

The first step towards leveraging such a QoS based quantification of composite services is to provide the means for *monitoring* the aforementioned attributes. A number of monitoring

systems have been proposed, however they are mainly tailored for a particular composition or workflow engine [161, 144, 70, 22, 33, 32, 86]. Thus, these systems are usually tightly coupled or directly integrated in the composition engine without considering other subsystems that live outside the composition engine but still influence the composite service execution. The lack of an integrated monitoring mechanism leads to a fragmentation of monitoring information across different subsystems. It does not provide a holistic view on all monitoring data that can be leveraged to get detailed information on the operational state at any point in time. Additionally, most monitoring approaches do not cover monitoring scenarios where requirements span across several unrelated composite services. Taken together, these shortcomings call for an integrated and flexible monitoring system for service composition environments.

Along with the ability to rate the operational state of composite services, such QoS based quantifications can be leveraged to provide *adaptations* in service compositions. Considering the case where a single service that takes part in a service composition causes failure of the whole composite service, the unresponsive service can be identified using its QoS data. A replacement or rebinding of this faulty service with an available and better performing alternative service can be used to improve the overall QoS of the composite service. However, a major problem with existing approaches is that current composition infrastructures only provide limited support for defining and handling even simple QoS-based adaptations at runtime [103, 164, 85, 84, 27, 40, 56, 107]. Another example for such adaptations is a change of a QoS attribute or the introduction of a new QoS attribute during the lifetime of a composite service. Handling such adaptations requires (a) mechanisms to define which service should be selected based on the desired QoS attributes and (b) appropriate runtime capabilities that define how the newly adapted composite service can be executed. However, most of these adaptations currently require a system engineer to manually re-engineer and redeploy the composite service which implies a disruption of the provisioning process.

1.2 Research Questions and Key Principles

The problem statement given above outlines two considerable issues that arise in service composition infrastructures. We can derive the following two research questions from Section 1.1:

1.2.1 Research Questions

Q 1. *What is a method for managing holistic monitoring requirements in heterogeneous composite service infrastructures?*

- What is a method to reason about non-functional aspects of composite services?
- What is an approach to integrate platform and technology agnostic monitoring capabilities in existing composite services?
- What is an appropriate abstraction to model both QoS and business related requirements?
- What is the performance penalty introduced by a system supporting these monitoring capabilities?

Q 2. *How can self-adaptation features be integrated into composite service infrastructures?*

- What is a method to provide self-adaptation for composite services that improves performance and dependability?
- What is the right amount of coupling between the supporting system and the composite service infrastructure to provide efficient, yet generic self-adaptation?
- What is a method to make the features of the supporting system available to non-technical users?

The final chapter will review these research questions and will examine in how far the methods presented in this work provide the means to answer them.

1.2.2 Key Principles

Along with the research questions listed above, the work presented in this thesis was driven by several *key principles*. These principles will be referenced hereinafter by their respective token (e.g., **[P1]**). Adhering to these principles had several effects on both the theory of operation of our approach as well as the implementation of the prototype framework. Additionally, these principles also affect the stakeholders of the proposed system in several ways. Therefore, an analysis of the key principles in the stakeholder context is provided in Chapter 8. Please note that in the following list of key principles, the term *system capabilities* refers to the monitoring and self-adaptation features that were discussed above.

[P1]: Non-Intrusiveness The system design shall require a minimum amount of changes in existing systems to leverage the system capabilities. Additionally, a potential performance overhead introduced by the system capabilities should be as low as possible.

[P2]: Platform and Technology Agnostic The system design shall not be tailored for a particular service composition platform or technology. Instead, the system design shall solely rely on the assumption that the service interaction within a composite service is realized by the means of a message exchange.

[P3]: Targeted at Domain Experts The system design shall enable the usage of the system capabilities by domain experts (cf. [64]). In particular, no programming skills should be required to leverage the system capabilities.

[P4]: Delivered as a Service The system design shall expose the system capabilities by the means of a service. This design principle shall (1) enable third parties to integrate the system capabilities into their infrastructure and (2) promote loose coupling within the design of the system itself.

1.3 Contributions

The research that has been carried out during the doctoral studies produced several scientific contributions, which represent the cornerstones of this thesis. Figure 1 gives a high level overview of these contributions. Please note that the *Service Composition Environment*, which is shown in light grey in Figure 1, is not part of the contributions, but represents an arbitrary service composition technology, such as a WS-BPEL compliant composition engine. Such a composition engine is enhanced with the monitoring and adaptation features that are described in the following.

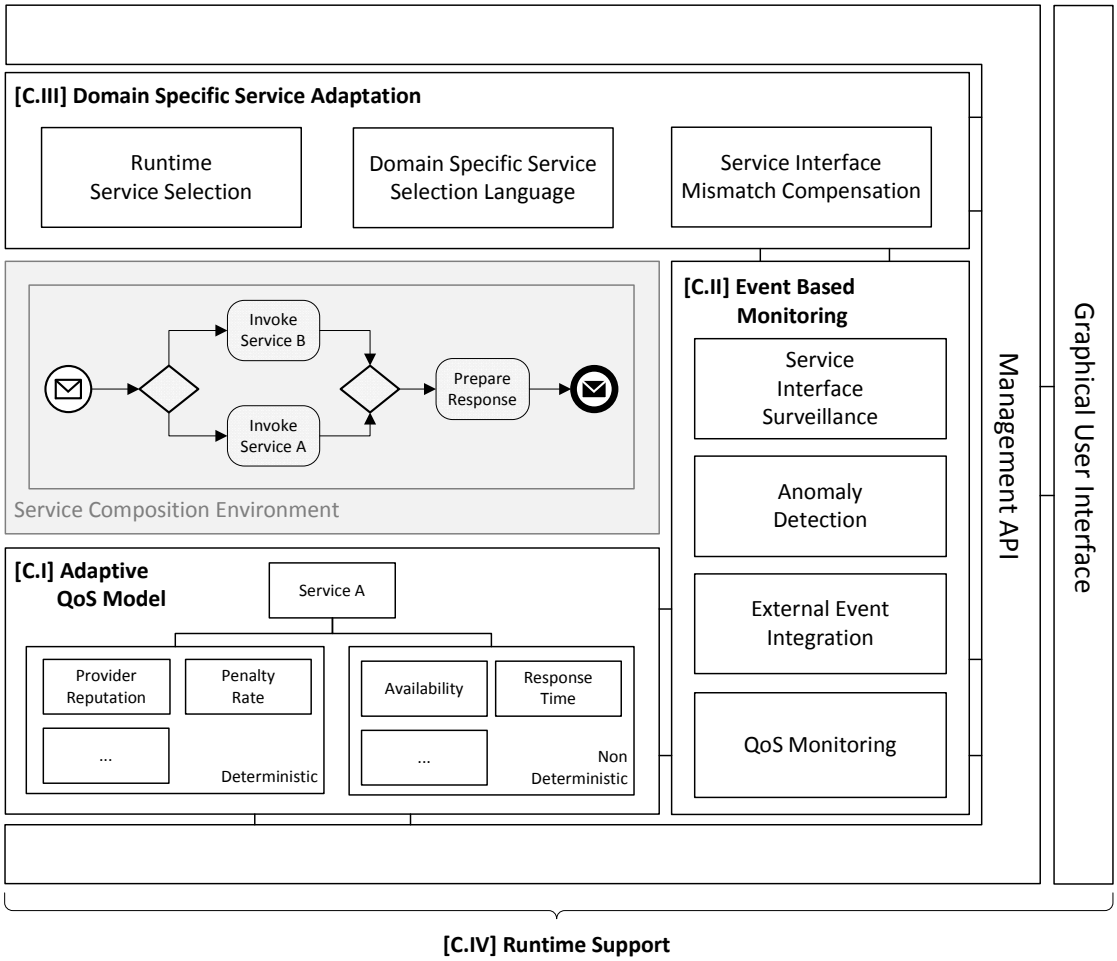


Figure 1: Contributions of this Thesis

[C.I] Adaptive QoS Model The first contribution of this thesis is an adaptive QoS model. Our model categorizes QoS attributes using the dimensions *determinism* and *applicability*. Values of deterministic QoS attributes are known in advance, whereas values of non-deterministic QoS

attributes require constant observation. Considering the applicability dimension, we categorize a QoS attribute to be either *domain agnostic* or *domain specific*. Hereby, we are able to constrain the applicability of a particular QoS attribute to a certain domain, such as the telecommunications domain, or make it available to the public domain. Another important feature is the *extensibility* of our QoS model and the *runtime adaptivity* of the QoS model implementation. This allows us to extend the QoS model with additionally required QoS attributes without imposing a downtime on the system. This adaptive QoS model represents an integral part of our framework and is leveraged by the other contributions described below.

[C.II] Holistic Composite Service Monitoring The second contribution is an *Holistic Monitoring* method for composite services. Our approach interprets the message exchange within a composite service as a stream of events. We leverage event stream processing methods to query, filter and aggregate the various events. This allows us to tackle a large variety of monitoring requirements, ranging from simple *QoS Monitoring*, such as observing the response times of a particular service, to highly complex monitoring requirements. Such complex monitoring requirements include temporal and causal dependencies between messages, monitoring requirements that span across several composite services as well as anomaly and outlier detection. Additionally, events provided by external systems can be easily integrated into our event processing infrastructure. This allows us to provide a holistic view on all monitoring data that is available in heterogeneous information system landscapes. Finally, our monitoring approach provides *Service Interface Surveillance*. Our system constantly monitors the interface descriptions of captured services. If a modification to a service interface is detected, a revision of this interface description is stored. This enables us to compare service interface revisions with each other and reason about their compatibility.

[C.III] Domain Specific Service Adaptation The third contribution of this thesis provides *Domain Specific Service Adaptation* for composite services. Adaptation is provided in terms of a *Runtime Service Selection* component. During the execution of a composite service, this component dynamically decides whether the invocation of a service should be routed to the service endpoint originally defined in the composite service specification (*original service*), or if an *alternative service* should be invoked instead. This decision is made upon (1) the QoS data provided by the monitoring component described above and (2) a selection strategy that is defined in a *Domain Specific Service Selection Language*. This domain specific language is small and easy to learn, yet comprehensive enough to model advanced service selection preferences that include conditional selection. The language is tightly integrated with the adaptive QoS model, which means that any extension to the QoS model is immediately available in the language. Finally, we provide a method for *Service Interface Mismatch Compensation*. This enables our system to resolve service interface incompatibilities between an original service and its alternative services, as well as tackle incompatibilities of different original service revisions.

[C.IV] Runtime Support Together with the aforementioned scientific contributions, we provide a proof of concept implementation in terms of the *VieDAME Framework*. The framework offers a REST based *Management API* to administer the components described above. A *Graph-*

ical User Interface, which is built on top of the Management API, supports end users with administrative tasks, such as extending the QoS model or defining service selection rules. Finally, this proof of concept implementation is used to show the feasibility of our approach in the evaluation sections. Currently, the implementation supports several major Open Source composition engines, which further emphasizes the practical usefulness of our framework.

1.4 Thesis Organization

The following briefly discusses the structure of this thesis as well as the conventions that are used for symbols, figures and text formatting.

1.4.1 Chapter Structure

The remainder of this thesis is organized as follows. Background information and a motivating example are discussed in Chapter 2 and Chapter 3. The scientific contributions of this thesis are presented in Chapter 4, Chapter 5 and Chapter 6. The proof of concept implementation is subject of Chapter 7 and evaluated in Chapter 8. Related work is discussed in Chapter 9. The thesis is concluded in Chapter 10.

Chapter 2 provides general background information on paradigms and concepts related to service oriented computing, autonomic computing as well as event stream processing.

Chapter 3 introduces a motivating example. This case study is referenced throughout the remainder of this thesis to motivate and exemplify our approach.

Chapter 4 discusses an adaptive QoS model for composite services. This model provides the foundation for the monitoring and adaptation approaches that are discussed in Chapter 5 and Chapter 6.

Chapter 5 presents an holistic monitoring method for composite services. The theory of operation, as well as architectural aspects of our approach are discussed in detail.

Chapter 6 examines an approach for runtime adaptation of composite services by the means of service selection. Analog to Chapter 5, the theory of operation and the architecture of the proposed method are presented in detail.

Chapter 7 delivers a compact overview of the proof of concept implementation of the presented monitoring and adaptation method.

Chapter 8 evaluates the proof of concept implementation with regard to system performance using both micro and macro benchmarks. An WS-BPEL implementation of the case study provides the basis for an end-to-end evaluation. Finally, we provide an in-depth qualitative discussion of system features and explicitly highlight the strengths and limitations of our approach.

Chapter 9 gives an overview of related work in the areas of monitoring and adaptation of composite services.

Chapter 10 concludes this thesis. It reviews the research questions with regard to the contributions of this thesis and provides some final remarks. Lastly, future work that is out of scope for this thesis is briefly presented.

1.4.2 Symbol, Figure and Style Conventions

This thesis adheres to certain symbol and style conventions that we will define in the following. These conventions should promote a clear understanding and reduce the possibility for ambiguities.

Symbol Conventions

Autonomic Computing Certain components of the contributions that we present in this thesis reference concepts related to the autonomic computing (AC) area, with a particular emphasis on the MAPE-K architecture. Along with textual references, we will use several symbols in our figures to mark a component related to a certain phase of the MAPE-K architecture. Figure 2 depicts the mapping of MAPE-K phases to their related symbols, while Figure 3 exemplifies these symbols. The MAPE-K phases are referenced by a rhombus labeled with the according starting letter. Figure 3a shows a component that covers the Monitor, Analyze and Plan phase of the MAPE-K architecture.

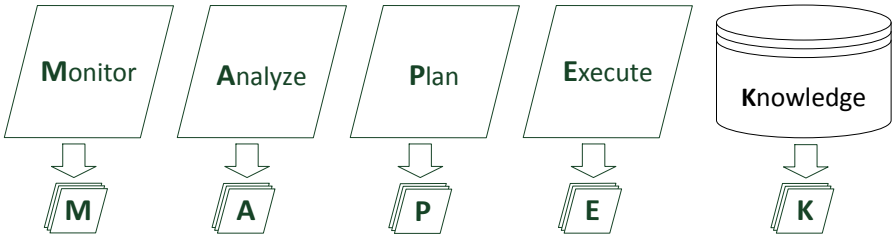
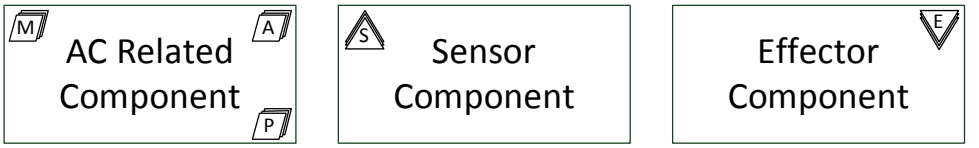


Figure 2: Symbols used to identify MAPE-K related components

On the other hand, components that represent Sensors or Effectors are labeled with a triangle. Figure 3b shows a Sensor component, while Figure 3c shows an Effector related component.



(a) AC Related Component (b) Sensor Related Component (c) Effector Related Component

Figure 3: Autonomic Computing Related Symbols

Message and Event Symbols Messages and events are an integral part of the contributions of this thesis. Therefore, we decided to use dedicated symbols to depict messages and events in our figures. Figure 4 displays the corresponding symbols, with Figure 4a exemplifying the message symbols and Figure 4b showing the symbol used for representing an event.

(a) Message Symbol (b) Event Symbol

Figure 4: Message and Event Related Symbols

Figure Conventions

With the exception of class diagrams, this thesis does not use UML [114] to realize figures. UML can be seen as the de-facto standard for the specification of system design related assets, such as use case specifications, sequence diagrams and component diagrams. However, the shortcomings of UML have been the subject of extensive discussion [34], and we share part of this opinion. We believe that it is difficult to create UML figures that provide a concise, yet comprehensive view on system architecture, in particular when considering "big pictures". Nevertheless, our figures adhere to the convention that dependencies between components are designated with a solid line, whereas message flow is designated with a dashed line. Dependencies and message flows can be directed, which is indicated by a arrow on the related end of the edge that marks the dependency or message flow. Examples for dependencies and message flow are visualized in Figure 5.

(a) Dependency Edge (b) Message Flow Edge

Figure 5: Dependency and Message Flow Related Edges

Style Conventions

In the list of requirements for each contribution (Sections 4.1.1, 5.1.1 and 6.1.1), we use the terms *shall*, *should*, *must* and *may*, as well as their negations, such as *must not*, as defined in RFC2119 [80].

1.5 Previous Publications

During the course of research for this thesis, several research papers were published and submitted to conferences and journals. These publications are an integral part of this thesis. The most important journal and conference papers are enumerated below. Please note that these publications are listed here once, and are not referenced again in the remainder of this work.

1.5.1 Journal Articles

- **O. Moser**, F. Rosenberg, S. Dustdar (2011). *Domain-Specific Service Selection for Composite Services.* IEEE Transactions on Software Engineering, IEEE Computer Society. (forthcoming)

1.5.2 Conference Proceedings

- **O. Moser**, F. Rosenberg, S. Dustdar (2010). *Event Driven Monitoring for Service Composition Infrastructures*, 11th International Conference on Web Information Systems Engineering (WISE'10), Hong Kong, China; 12.12.2010 - 14.12.2010; Springer, LNCS 6488 (2010), ISBN: 978-3-642-17615-9; S. 38 - 51.

- **O. Moser**, F. Rosenberg, S. Dustdar (2008). *VieDAME - Flexible and Robust BPEL Processes through Monitoring and Adaptation*, Informal demo, Proceedings of the 30th International Conference on Software Engineering (ICSE'08), 10-18. May 2008, Leipzig, Germany

- **O. Moser**, F. Rosenberg, S. Dustdar (2008). *Non-Intrusive Monitoring and Adaptation for WS-BPEL.* Proceedings of the 17th International World Wide Web Conference (WWW'08), 21-25. April 2008, Beijing, China.

CHAPTER 2

Background

This chapter provides background information about concepts and technologies that form the basis of the contributions presented in the remainder of this work. Readers familiar with *service oriented computing*, *autonomic computing* and *event stream processing* might skip this chapter. However, we recommend continuing with this introductory section, as it provides a quick overview how the contributions of this thesis relate to the aforementioned technologies.

In the first section, we introduce the principles of *service oriented computing*, with a strong focus on the characteristics of *services*. We further discuss *Web services* as a broadly accepted implementation technology for services. The first section also emphasizes on the *composability* of services to build so called composite services, and discuss *WS-BPEL* as the de-facto standard for the implementation of composite services using Web services technology. The principles of service oriented computing are essential to the contributions of this thesis in several regards. First, service orientation provides the foundational abstraction for the architecture of systems that this thesis is focused at. Moreover, when considering Key Principle **[P4]** (cf. Section 1.2.2), service orientation also provides the means of how the proposed framework itself is designed as well as how end user interaction is delivered. Second, Web service technology and WS-BPEL represent the means for implementation technologies that were chosen for the proof of concept implementation of our approach. Please note that the decision to use Web service technology for the proof of concept implementation does not restrict our approach to Web service based service oriented systems. The principles of service orientated computing are leveraged throughout Contributions **[C.II]** to **[C.IV]** (cf. Section 1.3).

The second section examines the principles of *autonomic computing*. We introduce the *MAPE-K* architecture, which serves as a blueprint for building autonomic computing systems. In relation to the work presented hereinafter, the knowledge of autonomic computing principles is not strictly required. However, when discussing the theory of operation of our approach, we will reference these MAPE-K components. Thereby, we provide an alignment of our system architecture to the components found in the MAPE-K architecture. This allows us to position our approach in the autonomic computing space, which helps to compare our approach with related work in this area. The principles of autonomic computing are leveraged in Contributions **[C.II]** and **[C.III]**.

The final section in this chapter introduces the concept of *event stream processing*. The foundational abstractions used in event stream processing are discussed, and concrete examples are provided based upon the OpenSource event stream processing platform *Esper*. With regard to the contributions of this thesis, event stream processing is used to reason about the interaction between services participating in a service composition. The basic approach here is to reduce the service interaction to a message exchange that is resembled by a related event stream. This abstraction effectively allows us satisfy Key Principle **[P1]**. Moreover, event stream processing technology enables us to derive information of interest from the event stream, ranging from simple performance statistics to the detection of complex situations of interest that include temporal and causal dependencies. Event stream processing technology is applied in Contribution **[C.II]**.

2.1 Principles of Service Oriented Computing

The foundational idea behind *service orientation* is simple. A *service provider* delivers a *service* s to a *consumer* that is requiring a particular functionality provided by s. The functionality offered by s is formalized in the *interface definition* of s. Figure 6 displays this high-level view on service orientation.

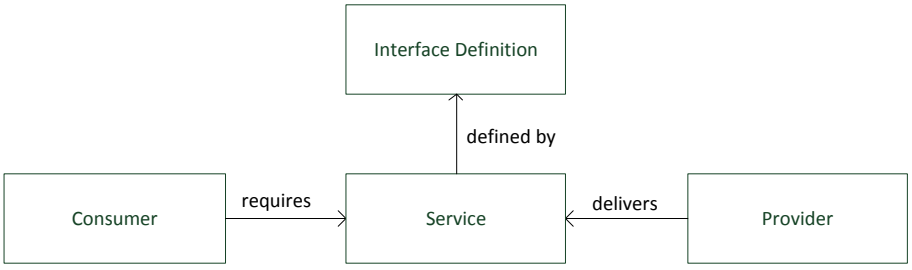


Figure 6: Basic Idea of Service Orientation

Service oriented computing (SOC) promotes the idea of assembling application components into a network of services that can be loosely coupled to create flexible, dynamic business processes and agile applications that span organizations and computing platforms [121]. Often, *service oriented architecture* (SOA) is falsely used as a synonym for SOC. However, it is important to note the difference, since SOA refers to an architectural style that leverages services as its building blocks.

Understanding the challenges and opportunities of SOC requires an understanding of the concept of *services* themselves at first. Literature on service orientation provides many definitions of what characterizes a service [16, 120, 53, 121, 75]. The definition given in [120] describes services as *self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications*. In [121], Papazoglou et al. extend this definition and additionally attest services that they can be *published, discovered and loosely coupled in novel ways*. In [53], Thomas Erl states that *services exist as physically independent*

software programs with distinct design characteristics that support the attainment of the strategic goals associated with service oriented computing. He further explains that *each service is assigned its own distinct functional context and is comprised of a set of capabilities related to this context*. In this regard, Papazoglou et. al provide a definition of services that is more specific about the characteristics of a service, such as that services can be described and published, while Erl's definition emphasizes on the fact that each service has its own context and provides a certain functionality in this context. On the other hand, Papazoglou's definition implies that services traverse through several lifecyle phases. This lifecycle starts with the description of a service, followed by the publishing of the service, until the service can be discovered and invoked by potential service consumers. In [104], Michelmayer et al. describe this basic model as the *publish-find-bind-execute* model where a service provider implements and publishes a service while the service consumer queries a *service registry* to find and bind the service, allowing the service consumer to invoke the service.

2.1.1 Service Characteristics

Aggregating the definitions listed above gives us a quite precise view on the characteristics of services. In the following, we provide a list of *service characteristics* that summarize the characteristics found in [120, 121, 126, 53, 147, 52]. Many of the characteristics found below are not restricted to service orientation but are derived from traditional software engineering practices. As an example, Perepletchikov et al. consider the traditional principle of *loose coupling*, which was originally defined by Stevens et al. in [143], from a service oriented perspective [130, 129].

Describable. A service s provides a formal description to its consumers that describes the interface exposed by s. This *interface description* d_s, which is provided by the service provider, defines the set of consumable functions offered by a service. Literature also refers to d_s as the *service contract*. Ideally, the interface description represents the only visible part of a service. Hence, services are often treated as "black-boxes" with regard to their implementation specifics.

Consumable. A service s is provided by a service provider and can be consumed by a service consumer. The technical aspects of the consumption of s, such as invocation patterns or message formats, are defined in the interface description d_s of s. Hence, the consideration of d_s is mandatory prior to the consumption of functionality provided by s.

Functionally Coherent. A service s provides a specific, coherent set of functionality. A particular functionality of s is represented in the interface description d_s of s by the declaration of an *operation* o_s of s. Such an operation can be compared to a method m in traditional object oriented design, where the interface of class c where m is declared corresponds to the interface description d_s where o_s is declared. Hence, consuming a functionality offered by s equals invoking a particular operation o_s.

Loosely Coupled. A service s provides loose coupling by physically and logically separating itself from its service consumer c_s. Moreover, the existence of service registries further loosens

coupling between service provider p_s and c_s by cutting the need for a preexisting relationship between p_s and c_s.

Publishable. A service s can be published by its service provider p_s using a service registry. Such a service registry enables p_s to organize information about s in a central repository and allows consumers of s to discover s.

Discoverable. A service s can be discovered by the service consumer using a service registry r. Discovering s using r is the inverse operation of publishing s into r, and hence, requires that s is published in r by its provider at first. Service discoverability also increases service reusability, since it allows to position services as IT assets with repeatable return of investment (ROI).

Technology Agnostic. A service provides its set of operations regardless of the platform where the service provider and the service consumer is deployed. This characteristic makes services ideal for application in heterogeneous infrastructures.

Composable. The services $s \in S$ of a set of services S can be composed to create a *service composition* that uses the operations provided by $s \in S$ to provide novel functionality to service consumers.

So far, we have discussed technology independent principles in the context of service orientation. Although there exist many implementation technologies [113, 165, 37, 67] that could be leveraged to build services, *Web service* technology emerged as the de-facto standard for the implementation of services [153]. Therefore, we will highlight this particular service implementation technology in the following section.

2.1.2 Web Service Technology

Web services represent a particular implementation technology for service oriented systems. Extending the general service definition that we provided earlier, a Web service is *a self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems or conducts transactions on behalf of a user or application* [122]. The foundation of Web service technology [150] is comprised of a markup language to structure documents (XML), a protocol definition for message transmission (SOAP) and a language to define service definitions (WSDL). Figure 7 gives an overview of the technologies that will be briefly described below.

XML The *eXtensible Markup Language* is a flexible and extensible markup language that provides the foundation for structuring and encoding documents related to Web service technology. Data is structured using *tags*, *attributes* and *elements*. XML is a W3C standard [151] and is broadly applied in diverse areas, ranging from word processors [50] to human-computer interaction in telephony [152]. XML documents must adhere to the rules of *well-formedness* and *validity*. Well-formed XML documents have to satisfy a list of syntax related rules [151], while

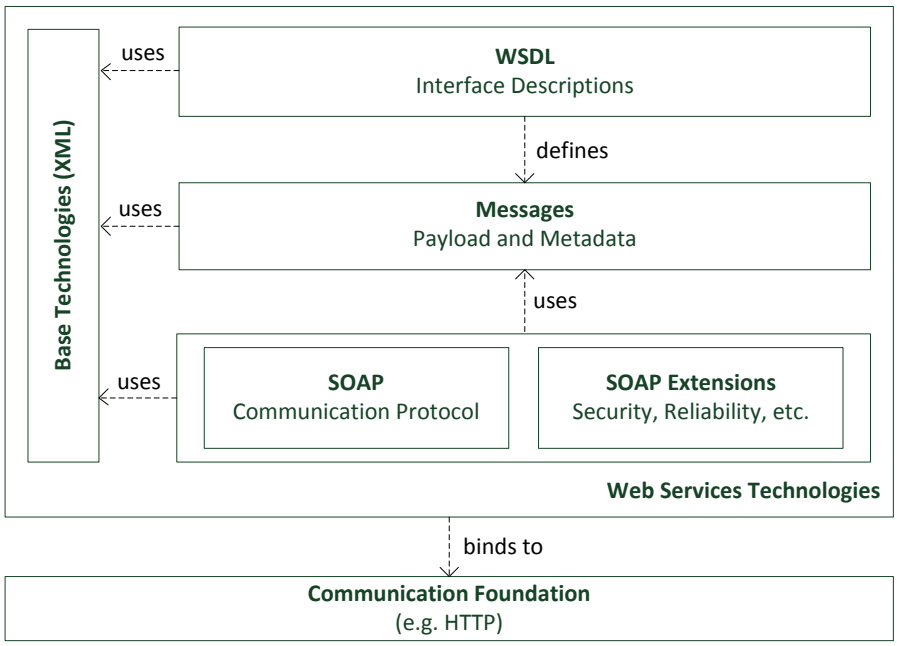


Figure 7: Web Service Technology

the validity of XML documents is defined by the means of a schema [160]. With regard to Web service technology, XML provides the basis for the message format as well as the structure of the interface definition.

SOAP SOAP [159], which was formerly an acronym for *Simple Object Access Protocol* [157], is a protocol for lightweight message exchange in distributed environments. SOAP is not defined in terms of application layer protocols. However, for compatibility reasons with network elements such as firewalls, it is often used in conjunction with HTTP(S). A SOAP message is comprised of an *envelope* that acts as a container for a *header* and a *body* element. The header element usually contains message metadata, such as hints regarding routing or encryption, while the body element contains the actual payload of the message. Listing 1 exemplifies a SOAP message.

WSDL The *Web Service Definition Language* [158] is used to define the interface description of Web services. A WSDL document is structured using XML technology and consists of an abstract part where data, message and interface types are declared, as well as a concrete part that binds the abstract definitions to a concrete transport protocol and endpoint address. Examples for WSDL documents can be obtained from the Appendix.

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:com="http://com.ebiz.services.external.validation">
  <soapenv:Header/>
  <soapenv:Body>
    <com:validate>
      <company>visa</company>
      <number>0000 0000 0000 0000</number>
    </com:validate>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 1: SOAP Message Structure

2.2 Composite Services

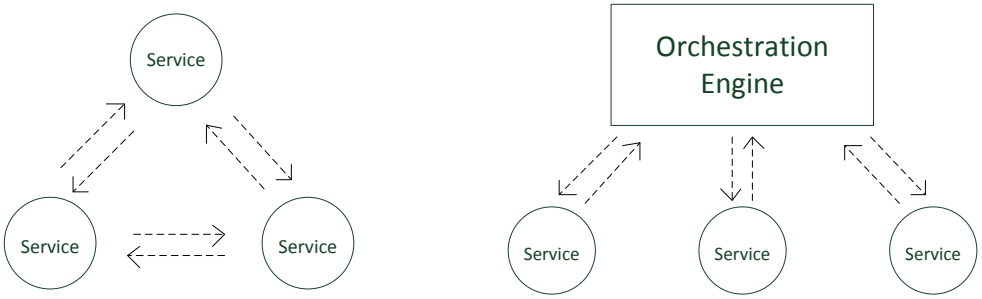
Papazoglou's definition of a service [120] emphasizes on the *rapid, low-cost composition* as a key feature of services. This characteristic can be leveraged to build *composite services*. Composite services, also referred to as service compositions, reuse functionality provided by existing services to automate a specific business task or process [53]. In many cases, composite services themselves are exposed by the means of a service to be consumable by service consumers. Hence, such a composite service shares the key characteristics of services listed above.

As a concrete example, consider the purchase order business process in an online book store. The checkout process is comprised of several steps such as checking the stock status of the items in your shopping basket or validating payment information. Each of these steps can be resembled by the means of a service that offers the required functionality. A service composition can then leverage these services to build a technical representation of the purchase order business process. Therefore, service compositions effectively enable the alignment of IT infrastructure to business processes [122].

In general, there are two distinct methods for composing services. In service *orchestration*, illustrated in Figure 8b, a central entity, which in turn can also be a service, is responsible for the invocation and structured coordination of the services participating in the composition. This central entity is aware of the service composition, while the orchestrated services do not have any knowledge of the service composition. This kind of service composition can be compared with an orchestra, where the members of the orchestra (the individual Web services) are lead by a conductor (the central orchestration process).

A service *choreography*, depicted in Figure 8a, does not rely on a central coordinating entity. With this kind of service composition, each of the services participating in the service composition knows exactly which operations should be invoked at a certain point in time, and what part it takes in the choreography. As an real world analogy, consider a dance choreography, where each dancer knows exactly what she is required to do.

Usually, the orchestration paradigm is applied to model high-level business processes that are intended for internal use, whereas modeling public business processes relies on choreography techniques. Apart from this difference, orchestration is more flexible and has several advantages



(a) Service Choreography (b) Service Orchestration

Figure 8: Service Composition Methods

over choreography. First, the participating services are not aware of the process. From this point of view, orchestration is less intrusive than choreography. Second, the coordination is controlled by a central and well-known entity, which improves manageability. Third, the orchestration paradigm provides better means for error compensation during a fault. Alternative scenarios can be used in such cases. Please note that for the remainder of this thesis, composite service are always assumed to be delivered by the means of orchestration.

2.2.1 WS-BPEL

So far, the principles of service composition have not referenced particular implementation technologies. However, and similar to the Web services technology, the *Business Process Execution Language for Web Services* (WS-BPEL, or simply BPEL) represents a composite services technology that is widely adopted. It uses Web services as the underlying implementation technology for composable services and shares many of the foundational technologies found in Web services, such as XML and WSDL. While we generally use the term *process* to refer to a business process, and use the term *service composition* to describe the structured composition of services, in the following paragraph we will adhere to BPEL terminology and use the term process to refer to WS-BPEL based composite service.

BPEL provides two different ways to define a process. First, abstract processes can be defined to describe the observable behavior and message exchange between involved services. On the other hand, executable processes allow to specify the exact details of a particular business process and are executable by an orchestration engine (BPEL engine). The reasoning behind separating the public behavioral aspects from the private aspects is that a company might not want to reveal their internal data and business management to the public. Furthermore, when using an abstract public representation of your business process, the internal/private aspects can still be changed without changing the public interface (the abstract process).

The language supports conditionals, variables, loops and other specific constructs such as fault handlers or compensation. The atomic steps in BPEL processes are represented by *activities*. Activities perform different tasks, such as invoking a Web service or assigning a variable. Web services, which are called *partner links* in BPEL terminology, can be invoked either sequentially or in parallel. Typically, a BPEL process is started by an external request, such as a Web service call, that triggers a start activity. Subsequent activities model the processing logic of the business process, before a response is finally sent back to the requester. The BPEL snippet from Listing 2 exemplifies the structure of BPEL process definitions.

```
<bpel:assign name="AssignOrderCC">
  <bpel:copy>
        <bpel:from part="purchaseOrder"
                              variable="order">
          <bpel:query>
                order/owner/creditCard
          </bpel:query>
        </bpel:from>
        <bpel:to part="isChargeable"
                    variable="creditCard">
          <bpel:query>card</bpel:query>
        </bpel:to>
  </bpel:copy>
</bpel:assign>
<bpel:invoke inputVariable="creditCard"
                           name="InvokeCheckCardService"
                           operation="isChargeable"
                           outputVariable="cardValid"
                           partnerLink="cardCheckService"
                           portType="ebiz:CreditCardCheck"/>
```

Listing 2: Variable Assignment and Invoke Activities in a BPEL Process Definition

2.3 Principles of Autonomic Computing

Conceived by IBM in early 2001 [72], the *autonomic computing* (AC) initiative was launched with the primary goal to deliver self-managing information systems. Facing a steady increase in the complexity of information systems, the need for an approach to tackle system evolution and management in a way that requires little to no human assistance became evident.

The foundational idea behind AC is to build information systems that are characterized by properties that resemble the properties of the human autonomic nervous system (ANS) [66]. In short, the ANS controls certain biological, automatically running processes that are inherent to the human body. As an example, the ANS controls the vital functions of heart rate and blood pressure. Therefore, it decouples and frees these processes from the consciousness of the brain.

Applying these thoughts to information systems yields the vision of systems that do not require human intervention for their operation. While Paul Horn originally identified eight characteristics of AC [72], we will constrain our discussion to the four so-called *self* properties* [66, 36, 89, 20] of autonomic computing systems (ACS), which are inspired by the properties of software agents [156]:

Self-Configuration enables ACS to automatically adjust to new or changed environmental situations according to human-defined high-level goals. The emphasis is on the fact that these goals only define what is desired or required, not how to actually reach these goals.

Self-Optimization enables ACS to continuously determine the performance of the system under consideration and apply optimizations to increase the performance or service quality. Hence, this feature leads to an optimal use of available resources.

Self-Healing enables ACS to capture its current operational state and thereby derive information about system health. This information is then used to eventually react accordingly by the means of repair and fault correction methods. Such methods include fail-over switching to redundant software or hardware components, as well as automatically acquiring and applying important software patches.

Self-Protection enables ACS to proactively sense and respond to malicious behaviour. Self-protection not only include security features to protect against malicious attacks but also tackles end-user caused problems, such as inadvertently deleted files.

Along with these self* properties, IBM proposed a reference model for designing autonomic computing systems. The following section introduces this reference model.

2.3.1 Architectural Blueprint for Autonomic Computing: MAPE-K

The *Monitor-Analyze-Plan-Execute – Knowledge* (MAPE-K) architecture [77], depicted in Figure 9, provides a blueprint for the architectural aspects of autonomic systems. Additionally, it provides a common ground for systems designed with autonomic features in mind to be classified and aligned to this MAPE-K architecture. Its building blocks are (1) a four-phases *control loop* (MAPE) that shares a common knowledge (K) base, (2) an *autonomic manager* that is in charge of the MAPE loop, and (3) a *managed element* that is subject to autonomic management.

The autonomic manager is a software component that can be configured using high-level goals. Such goals are mostly coarse grained descriptions of what should be achieved, and are configured by a human administrator in a declarative fashion, usually using event-condition-action (ECA) policies or goal policies [90]. ECA policies are declared using the following structure: "When *event e* occurs and *condition c* is true, then perform *action a*". An example for such an ECA policy would be "When *response-time of server X* $\geq$ *5000ms* and *additional available server resources* $\geq$ *1*, then *start additional server resource for X*." Goal policies, on the other hand, are more high-level. They only declare how a desired system state looks like, not which conditions have to be satisfied and which actions have to be taken to reach the desired system state. Hence, the analogon for the ECA policy from above would be: *"The response-time of server X should be* $\leq$ *5000ms"*. The autonomic manager then dissects these goals in the phases of the MAPE loop to derive the low-level steps required to accomplish the goals. Finally, a common *Knowledge* base serves as a repository for all kinds of information related to the managed element, as well as internal information related to the autonomic manager. The

Knowledge includes data such as metrics, metadata as well as previous execution traces. This information can be used in isolation, or it can be shared with other autonomic managers [77].

A managed element represents any software or hardware resource that should be augmented with autonomic behavior. Such resources can include physical devices, such as CPUs, hard-disk drives or a printer, as well as software resources, such as application servers, virtual machines or mobile applications. The interface for the autonomic manager to tap the autonomic element is realized by *Sensors* and *Effectors*. Sensors are required to collect information about the managed element, and provide this raw data via an interface to the autonomic manager. Hence, Sensors can be seen as a part of the extensive research stream on systems monitoring [149, 162, 137]. Effectors, on the contrary, are required to apply the low-level actions to the managed element. These low-level actions result from the computations performed in the MAPE loop. Similar to Sensors, an interface is provided for the autonomic manager to enact these changes.

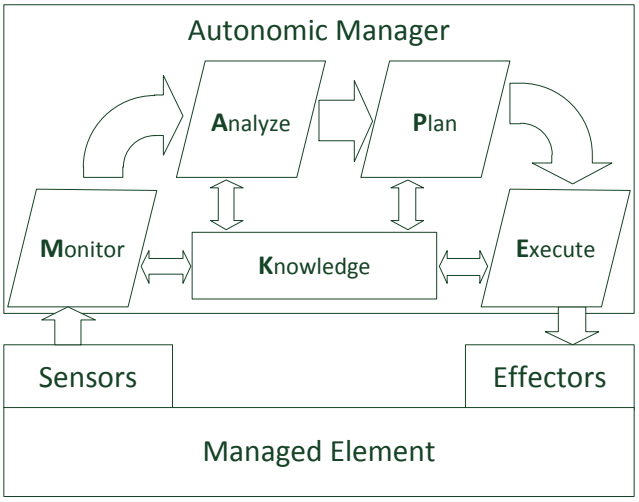


Figure 9: MAPE-K Autonomic Computing Blueprint

We will now briefly consider the individual phases of the MAPE-K architecture.

Monitor In the Monitor phase, the autonomic manager collects, filters and aggregates details on a managed element using a Sensor interface. These details consist of certain metrics that indicate the operational state of the managed element, such as performance and dependability related metrics, as well as business related metrics. The Knowledge base acts as a repository to store this data.

Analyze In the Analyze phase, the autonomic manager applies computations and algorithms on the data prepared by the Monitor phase to create a model of the current operational state of the managed element. The complexity of this model can vary dramatically, ranging from simple models that solely contain a series of data points, to highly complex models that include

temporal and causal dependencies. The resulting model is shared with the Plan phase using the Knowledge base.

Plan In the Plan phase, the autonomic manager uses the model created by the Analyze phase, as well as the policy that defines the desired optimization goals, to create a change plan. A change plan defines the required steps to improve and optimize the operation of the managed element. The complexity of such a change plan is use case specific, ranging from single commands to complex workflow definitions. Again, the Knowledge base is used to store, retrieve and share the required data, including the change plan definition and the goal policies, with the other phases.

Execute Finally, in the Execute phase, the autonomic manager applies the change plan derived from the human provided goal policies and computed by the Plan phase to the managed element. Hereby, the autonomic manager leverages the managed element's Effector interface to apply the changes defined in the change plan.

Please note that in the remainder of this thesis, we will use the term *MAPE-K* to refer to the architecture blueprint as a whole, while *MAPE-K phase* will be used to refer to a specific phase of the control loop of the MAPE-K architecture. Interested reader might refer to [72, 66, 89, 77, 73] for more information on the principles of autonomic computing.

The last section of this background chapter examines the concepts behind event stream processing, an emerging disciple that promises to tackle many problems in contemporary enterprises settings.

2.4 Event Stream Processing

Event stream processing (ESP) is an emerging discipline that promises to deliver a solution for processing vast amounts of data provided in the form of continuously flowing data streams. Its foundational abstraction is based upon the representation of data, emitted by producers, as a stream of events that eventually require additional processing to be meaningful for interested consumers. This high-level view on ESP systems is depicted in Figure 10. In the context of ESP systems, the term *event* refers to "anything that can happen", more specifically, to something that is meaningful for conducting commercial, industrial, governmental, or trade activities [38]. In the following, the technical representation of an event in an ESP system will be referred to as an *event object*.

Contemporary enterprises from diverse sectors offer many examples for such event streams. Considering the financial sector, stock tickers can be delivered in the form of event streams, with events representing changes in stock prices. In the health sector, ESP can be leveraged to support the surveillance of patient vital signs, using event streams to represent blood-pressure or heart-rate measurements. Examining the high-level ESP system architecture shown in Figure 10, ESP systems consist of (1) *event producers*, (2) *event stream processors* and (3) *event consumers*. Event producers, also referred to as event sinks, represent entities that emit events into an event

stream. Considering the example from the health sector, sensors that measure patient vital signs act as event producers, whereas a nurse information terminal, notifying the medical staff about alerting situations, represents the event consumer. The most demanding role in this architecture falls to the event stream processor. Its responsibility is to sense and respond to certain situations of interest, such as alerting changes in a patient's vital signs.

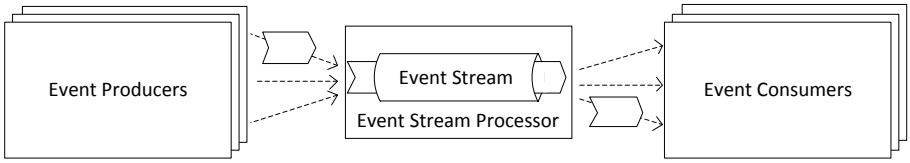


Figure 10: High-Level Architecture for Event Stream Processing Systems

Additional to the high volume of data that is generated by event producers, the vast majority of event consumers have very stringent requirements regarding the processing time of these high-volume data streams. Clearly, this becomes most evident considering the example from the health sector, as non-timely processing of the related data stream can lead to life-threatening situations.

Using the observations from above, we can derive two distinct characteristics for ESP systems. First, such a system is required to operate on a high-volume stream of events that arrive at unpredictable rates. Second, ESP systems have to provide low-latency processing capabilities to satisfy the high demands of event consumers. Although traditional database management systems (DBMS) provide similar features, they require data to be persistent before it can be processed. Moreover, traditional DBMS are built around the *human active - database passive* processing model [128], where the DBMS processes data only when it is explicitly triggered to do so. On the contrary, ESP systems, which have their origin in the area of active database systems [46], continuously provide feedback on the event stream it is operating on. In particular, user defined queries are stored in memory and event data is pushed through these queries. This approach provides better performance and scalability than disk based database systems for large amounts of data.

In the following, we will use the ESP platform Esper to exemplify the properties of such user defined, on-line queries.

2.4.1 Event Processing with Esper

Esper [57] is an OpenSource event stream processing engine for the Java platform. It provides a domain specific language for event processing (EPL). This EPL offers two methods to operate on the event stream. First, the EPL allows to query the event stream using *event stream queries*. Such event stream queries are similar to queries declared in the structured query language (SQL). However, the EPL provides distinct features that set it apart from traditional SQL, such as sliding

time windows. These sliding windows enable the limitation of the number of events that are considered for a certain query. An example for an EPL query is shown in Listing 3.

```
1 select
2   median(executionTime), name
3 from
4   ServiceInvocationEvent.win:time(5 min)
5 group by
6   name
```

Listing 3: EPL Sample using Sliding Time Window Semantics

Considering Listing 3, one can easily see that the general structure of the query resembles the structure of a SQL query. The result of the EPL query from Listing 3 is a list of the names of all ServiceInvocationEvents and the median of their executionTimes that were registered in the ESP system during the last five minutes. `Select`, `from` and `group by` have a meaning very similar to the corresponding key words used in SQL. For example, the `select` clause enables us to chose specific event properties, whereas the `from` clause is used to select a particular event stream. However, and contrary to SQL which operates on data structured using tables, the EPL operates on an event stream that contains events of a particular type. In the example above, we express our interest in events of type `ServiceInvocationEvent` by specifying it as the argument of the `from` clause in the EPL query, where we would normally specify a table name in the case of SQL. `Name` and `executionTime` are simple properties of this `ServiceInvocationEvent` type. Moreover, we define the length of the sliding window that should be considered to be 5 minutes (`win:time(5 min)`). Such a sliding time window extends to the specified time interval into the past using the system time. The statement from Listing 3 is a very simple example of the power of the EPL, as it provides much more sophisticated features that include joins over several event streams, aggregation or including historical data that resides in a traditional RDBMS. Due to space restrictions, we cannot go into more detail in this work, however, the interested reader can refer to [57] for reference.

Along with these event stream queries, the EPL provides another powerful mechanism to work with the event stream. *Event patterns* enable us to detect situations of interest. An event pattern matches when a single or multiple events occur that match the event pattern definition. As an example for such an event pattern, consider the pattern from Listing 4.

```
1 every
2   s1 = ServiceInvocationEvent(s1.success = true) ->
3   s2 = ServiceInvocationEvent(s2.success = true)
```

Listing 4: EPL Pattern Sample using the Followed-By Operator

In short, the pattern from Listing 4 matches every time a successful `ServiceInvocationEvent` is followed by another successful `ServiceInvocationEvent`. To this end, we leverage the `every` operator to specify that the pattern sub-expression, that is, the expression that follows the `every` operator, should restart when the pattern sub-expression evaluates to true or false. In other words, leaving out the `every` operator would instruct the event processing system to stop looking for further events once there was one successful match. The other important operator in the pattern from Listing 4 is the followed-by operator `->`. Using this

followed-by operator, we can specify that the pattern consists of two or more events that are followed by one another in terms of their time of occurrence. Hence, the followed-by operator enables us to effectively deal with the temporal and causal properties of events.

The last important aspect that we will consider here are the different *event representations* supported by Esper. Since we will leverage Esper on the Java platform, we will focus on the Java based representation of events in Esper. For a quick overview, we have summarized the supported event representations in Table 1.

Java Class	**Description**
java.lang.Object	any Java POJO (plain-old java object) with getter methods following JavaBean conventions; Legacy Java classes not following JavaBean conventions can also serve as events
java.util.Map	Map events are key-values pairs and can also contain objects, further Maps, and arrays thereof
org.w3c.dom.Node	XML document object model (DOM)
org.apache.axiom.om.OMDocument	XML - Streaming API for XML (StAX) - Apache Axiom (provided by EsperIO package)

Table 1: Summary of Event Representations in Esper (adapted from [57])

Which one of the different event types available should be used to model a specific event depends on the use case and should be carefully considered. From the various event representations listed in Table 1, we will leverage the events of types `java.lang.Object` and `org.apache.axiom.om.OMDocument` in the proof of concept implementation of our system. In particular, we will use events of type `org.apache.axiom.om.OMDocument` to model the message exchange within composite services. Thereby, we can define patterns and queries over the payload of messages exchanged between a composite service and its individual services, which provides us with a powerful tool to model complex, business-related monitoring requirements. Further details how we leveraged Esper for our monitoring system will be presented in Chapter 5 and Chapter 7.

2.5 Summary

This chapter provided a review of the foundational concepts and technologies that the remainder of this thesis relies on. The principles of SOC were discussed, with an emphasis on the characteristics of services, the atomic entities that service oriented systems are built upon. Along with the discussion of the foundational concepts found in SOC, the chapter discussed Web services technology and WS-BPEL. Web services are a commonly accepted implementation technology for services, while WS-BPEL provides the de-facto standard for implementing composite services that are built using Web services technology.

The second important topic that this background chapter discussed was autonomic computing, which is a method for building information systems with self-management capabilities. The

section characterized autonomic computing systems using the so called self-*properties, and introduced the MAPE-K architecture, which serves as a blueprint for the design of autonomic computing systems.

Finally, we introduced event stream processing as a technology for tackling the high-volume, low-latency processing of data streams. The foundational abstraction relies on representing data generated by producers and provided to consumers as a stream of events. The OpenSource ESP platform Esper was briefly introduced to give a concrete example for an ESP system as well as prepare the reader for the EPL examples provided in later chapters.

CHAPTER 3

A Motivating Example

This chapter introduces a sample business process from the telecommunications (telco) domain. The *SubscriberActivation* process will be referenced throughout the remainder of this dissertation. It will be used to motivate and highlight the challenges and opportunities of contemporary enterprises with regard to monitoring and adaptivity of composite services. In the first section of this chapter, we will focus on the functional requirements of this business process, whereas in the second section, we consider the non-functional requirements of the SubscriberActivation process. The chapter is concluded with a short wrap up of key points. Please note that in the remainder of this thesis, the term *composition* refers to a composite service, while the term *process*, if not noted otherwise, refers to a business process.

3.1 The SubscriberActivation Process

The *SubscriberActivation* process provides the functionality that is required to activate a new customer throughout the back-end systems of the telco enterprise *Phonyfone*. Prior to the discussion of the process semantics, the relevant terms are explained to give the reader a basic understanding of telco terminology. Although the following explanations explicitly reference Phonyfone, to a large extent, the terms apply to telco enterprises in general.

3.1.1 Terminology

The term *subscriber* refers to the technical representation of a customer in Phonyfone's information systems, while a *deal* represents an orderable service offered by Phonyfone, such as mobile email access or ManagedRoaming. The term *roaming* [69] describes a network operator's capability to extend their services, both voice and data related, to a network other than the home network. The term *MSISDN* (Mobile Subscriber Integrated Services Digital Network Number) [1], is, simply put, the cellphone number of a SIM card. The *SIM* (Subscriber Identity Module) [2] card is a removable chip that stores subscriber information in cellphones. Finally, the *IMSI* (International Mobile Subscriber Identity) and the *ICCID* (Integrated Circuit Card ID)

are identifiers stored on the SIM card and are globally unique [3]. The IMSI represents the key for the network operator to identify a subscriber, while the ICCID identifies the SIM card itself.

3.1.2 Semantics of the SubscriberActivation Process

In the following, we will elaborate on the semantics of the SubscriberActivation process. The process is depicted in Figure 11.

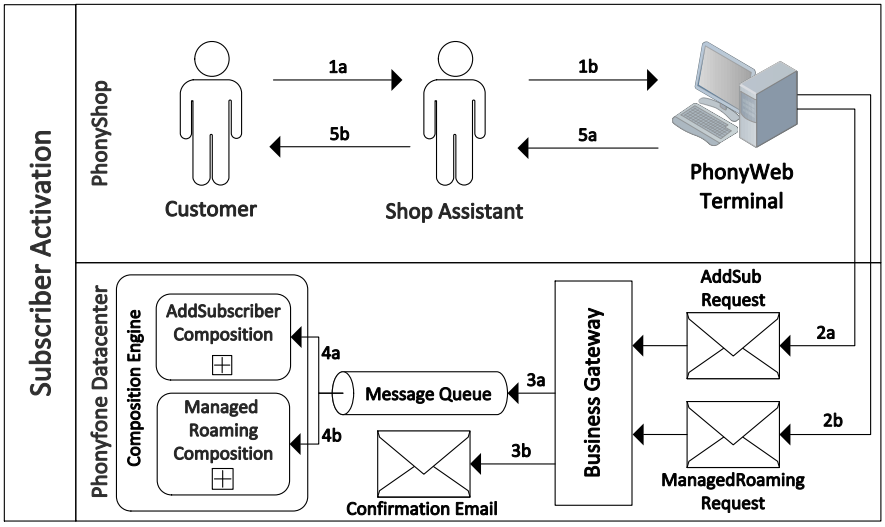


Figure 11: Phonyfone Subscriber Activation Process

The SubscriberActivation process formalizes the steps required for a customer to obtain a new cell phone contract with the telco provider Phonyfone. Hence, the *Customer* provides required information, such as personal and payment data, to a *Shop Assistant* (1a). The Shop Assistant creates a new contract by entering the related data into the *PhonyWeb Terminal* (1b). The PhonyWeb Terminal transmits the customer (2a) and service data (2b) to an on-premises *Business-to-Business Gateway* (B2BGW). The B2BGW dispatches the two requests into a *Message Queue* (3a), and submits a *Confirmation Email* (3b) to the email address that the customer provided. The Message Queue delivers the requests (4a and 4b) to Phonyfone's *Composition Engine*. The Composition Engine triggers the execution of both the *AddSubscriber* and *ManagedRoaming* compositions. These compositions enable Phonyfone to automate the steps required to create a *subscriber* object for a new customer in various Phonyfone back-end systems as well as setup a deal for the ManagedRoaming service. The ManagedRoaming service enables Phonyfone to offer the best rates to their customers when they use their cell phones in foreign networks. Upon normal completion of these compositions, the subscriber becomes active, and the PhonyWeb Terminal provides feedback to the Shop Assistant (5a). Finally, the Shop Assistant can handover a printout of the contract to the customer (5b).

```
1 <AddSubRequest>
2   <Msisdn>435550000001</Msisdn>
3   <Tariff>Phonyfone_Broadband</Tariff>
4   <IsPostpaid>true</IsPostpaid>
5   <Originator>p_2342</Originator>
6   <ReducedRecurringCharge>
7     <StartDate>2010/02/01</StartDate>
8     <EndDate>2010/03/01</EndDate>
9     <Amount>10.50</Amount>
10  </ReducedRecurringCharge>
11 </AddSubRequest>
```

(a) AddSubscriber Request

```
1 <RegisterRoamerRequest>
2   <Msisdn>435550000001</Msisdn>
3 </RegisterRoamerRequest>
```

(b) RegisterRoamer Request

Figure 12: SubscriberActivation Requests

The following sections highlight the two most important components of the Subscriber-Activation process, the *AddSubscriber* and *ManagedRoaming* compositions. Figures 13 and 14 illustrate the compositions using a BPMN (Business Process Modeling Notation) diagram [112]. Please note that fault and compensation handling are not shown for brevity.

3.1.3 The AddSubscriber Composition

The AddSubscriber composition is triggered by receiving an AddSubscriber request. Listing 12a shows the request structure along with sample values. The request contains the MSISDN (line 2), the tariff information (line 3), the username of the shop assistant that created the contract (line 5) and an optional reduced recurring charge (RRC) field with a given start and end date (lines 6-10). This RRC enables shop assistants to reduce the monthly fee for a certain duration.

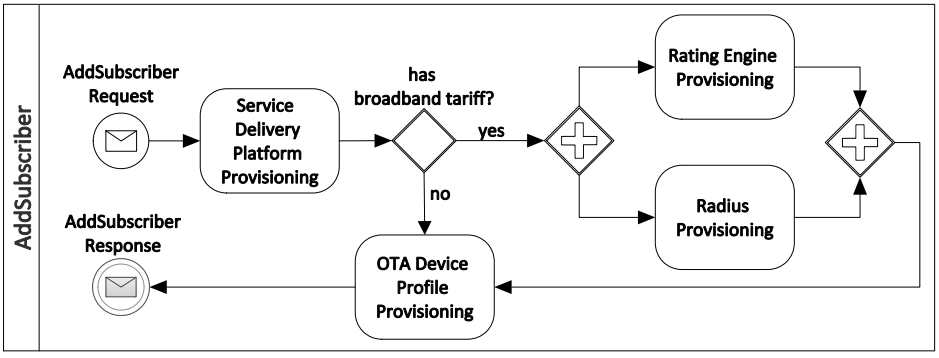


Figure 13: AddSubscriber Composite Service

Provisioning of core subsystems starts with persisting relevant data in Phonyfone's *Service Delivery Platform* (SDP). The SDP provides access to subscriber and service relevant data. Then, if the tariff information provided in the request indicates a broadband product (for mobile Internet access), subscriber data is provisioned to two additional systems (*RADIUS* and the

Rating Engine for dial-in services). Next, the *Over The Air* (OTA) device profile provisioning stores cellphone model and brand information in a device configuration server. Finally, an AddSubscriber response, indicating a success or failure message, is sent back to the requester.

3.1.4 The ManagedRoaming Composition

Initially, the ManagedRoaming composite service waits for an incoming *RegisterRoamer* request, which instantiates the composition. The request structure is shown in Listing 12b. It includes the MSISDN (line 2) of the subscriber that should be enabled for ManagedRoaming.

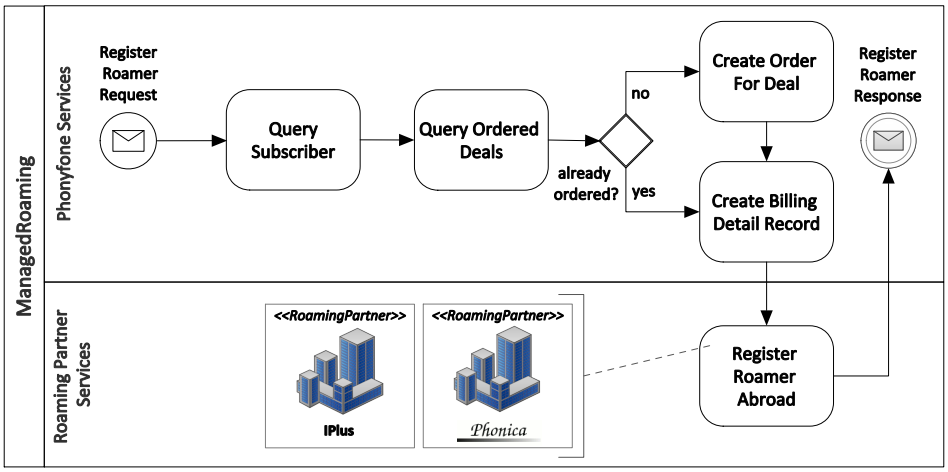


Figure 14: ManagedRoaming Composition

In the first activity, the composition maps the subscriber's MSISDN to the IMSI by invoking the *QuerySubscriber* service. This is necessary because some of the services in the ManagedRoaming composition require the IMSI of the related SIM card. Provided that the input MSISDN matches a subscriber object in Phonyfone's subscriber database, the service returns the details of the subscriber, including the IMSI and ICCID. The next activity checks whether the subscriber already has an order for the ManagedRoaming deal in Phonyfone's subscription platform. The *QueryDeal* service provides the required functionality, accepting the MSISDN as an input parameter. It returns relevant data of all deals the subscriber has ordered. If there is no order for the ManagedRoaming deal, the composition creates it by invoking the *CreateOrder* service. It takes the MSISDN and the desired deal name, i.e. ManagedRoaming, as input parameters. Then, the composition creates a *Billing Detail Record* (BDR for short). This BDR is required internally for Phonyfone's billing and accounting system.

Up to this point in the process execution, only on-premises services are involved in service interaction. This means that Phonyfone has full control over the participating services. The final and most important activity in the ManagedRoaming composition covers the registration of the subscriber in the foreign network. For this purpose, Phonyfone contracts several roaming partners. Figure 14 exemplifies two of these contractors, *IPlus* and *Phonica*. These partner

companies have to offer an interface that abstracts the details of subscriber registration in the foreign network. However, there is no guarantee that the roaming interfaces offered by different roaming partners are compatible with each other. The *RegisterRoamerAbroad* activity invokes this roaming interface and provides the MSISDN, the home network provider, e.g., Phonyfone, and the IMSI and ICCID of the SIM card to the external partner service. A *composition designer* links the corresponding partner service with the activity in the ManagedRoaming composition. Hence, this person statically defines which contracted roaming partner is used for the RegisterRoamerAbroad activity. Using a different roaming partner requires a design-time modification of the composition, as well as a redeployment.Therefore, switching roaming partners means (1) an expert resource, that is, a composition designer, is required to perform the modifications and (2) an unavailability of the composition during the redeployment is inevitable.

Along with the functional requirements that have been described so far, the SubscriberActivation process has certain non-functional requirements that are crucial for unrestricted operation. These non-functional properties are examined next.

3.2 Non-Functional Requirements

The non-functional requirements of the SubscriberActivation process can be distinguished using (1) the *category* of the requirement, such as performance or dependability, and (2) the level of *granularity* The category describes *what* needs to be considered, whereas the granularity describes *where* in the composition stack the requirement needs to be considered. Table 2 summarizes the requirements for a quick overview. Regarding the listed granularity, the *service level* and the *composition level* refer to a requirement that needs to be considered on the service or the composition level, whereas *multi-composition level* refers to a requirement that spans across several compositions.

3.2.1 Performance and Dependability Requirements

A paramount upper bound for both the AddSubscriber and the ManagedRoaming compositions is their execution time. If it exceeds a certain threshold, the related customer is directly affected. The customer either cannot use the better rates offered by ManagedRoaming, or, in case the AddSubscriber composition fails to execute in time, cannot use her cellphone at all. Furthermore, the availability and the response times of the partner services, such as the RegisterRoamerAbroad service, is highly important for a successful execution of the compositions, since an unavailable or unresponsive partner service directly affects the execution of the related composition.

Besides the worst case scenario from above, there are certain average values for the response time and the availability of the AddSubscriber and ManagedRoaming compositions, as well as an average combined execution time of both compositions. Deviations in these numbers can indicate potential problems, either in the back end systems or the network backbone itself.

3.2.2 Business Related Requirements

Previously, we described that Phonyfone contracts various roaming partners for their ManagedRoaming offering. These roaming partners offer their service at different rates and fees. The *CallSetupFee* are costs that incur during the initial setup of the roaming call and have to be paid by Phonyfone. The *VoiceTrafficRate* is charged for the actual duration of the call and has to be paid by the customer. Minimizing these costs is highly desirable for Phonyfone. Additionally, Phonyfone requires that roaming partners offer their service via an encrypted channel.

Moreover, there are business performance indicators that reflect marketing related aspects, such as customer acceptance of a new product offering. In this regards, Phonyfone faces a certain average number of subscriber activations per day, with the vast majority of activations taking place during the working hours. For various reasons, these numbers can rise or fall on short notice. Moreover, it is highly desirable to determine best-selling products and tariffs. Certain trends in customer behaviour can be derived, as well as in-depth analysis can be performed, such as correlating the number of sales with the location of the shop where the customer contract was signed. Another important issue is the abuse of the reduced recurring charge (RRC) fields. We described above that a RRC can be set to reduce the monthly fee for a customer for a certain duration. However, a shop assistant is only allowed to reduce the recurring charge for a limited amount of customers per day. Hence, it is important to find out if there are any violations to this constraint.

Category	Requirement	Granularity
Performance	responseTime(`RegisterRoamerAbroad`) $\leq 500ms$	S
	responseTime(`ManagedRoaming`) $\leq 2500ms$	C
	responseTime(`AddSubscriber`) + responseTime(`ManagedRoaming`) $\leq$ 10000ms	C^2
Dependability	accuracy(`RegisterRoamerAbroad`) ≥ 0.99	S
	accuracy(`ManagedRoaming`) ≥ 0.99	C
	availability(`ManagedRoaming`) ≥ 0.999	C
Business	`RegisterRoamerAbroad`: minimize *CallSetupFee* and *VoiceTrafficRate*	S
	`RegisterRoamerAbroad`: require secure channel	S
	`AddSubscriber`: number of new subscribers with tariff `PhonyfoneBroadband` per week ≥ 2000	C
	`SubscriberActivation`: number of activations between 09:00 and 17:00 in [500..600]	C^2
	`AddSubscriber`: number of activations with RRC ≤ 5 per day	C

S service level – C composition level – C^2 multi-composition level

Table 2: Non-Functional Requirements of the SubscriberActivation Process

3.2.3 Operational Aspects

During Phonyfone's company history, several acquisitions of smaller telco firms have been made. The result is a vastly heterogeneous information system landscape, including different composition engines from various vendors. Hence, the composition engines that execute the AddSubscriber and ManagedRoaming composite services are provided by different vendors.

Another important aspect is the relatively high frequency in interface changes of the individual services of both the AddSubscriber and ManagedRoaming compositions. Interface changes are often driven by short term market requirements, such as the support for a new product offering. In rare cases, such changes are not backward compatible, which means that the service interface description that is used in the composite service is incompatible with the service interface truly supported by the individual service. Evidently, the situation is worse with service that are not directly operated by Phonyfone, such as the RegisterRoamerAbroad services. Ultimately, such situations can lead to an unavailability of the related composite service.

3.3 Summary

This chapter presented a genuine case study from the telco domain. A description of the SubscriberActivation process, as well as its two most important components, the AddSubscriber composition and the ManagedRoaming composition, were provided. Along with the description of functional requirements, the chapter also emphasized on the non-functional requirements of the process and its components. These non-functional requirements will be used to derive the requirements for a corresponding QoS model in the following chapter.

CHAPTER 4

An Adaptive Quality of Service Model

This chapter discusses the *adaptive Quality of Service model* that forms the foundation of the framework presented hereinafter. This model provides an extensible method to formalize the non-functional requirements found in service oriented computing systems. The proof of concept implementation of this model features *runtime adaptivity* to support additionally needed non-functional requirements that were not originally supported by the model.

In the first section of this chapter, we discuss the the concept of *Quality of Service* (QoS) in general. We examine how non-functional service requirements are mapped to *QoS attributes* and discuss how such attributes can be classified. Our case study from Chapter 3 is used to exemplify this QoS model. This motivating introduction is further used to derive the key requirements for such a QoS model, which are listed at the end of the first section.

In the second section, we consider the theory of operation of the proposed QoS model. We list *domain agnostic* QoS attributes that our model supports, that is, QoS attributes that are applicable regardless of the application domain. What follows is a discussion of what we call *domain specific QoS*, which is used to motivate the need for *domain specific* QoS attributes. Such attributes are only applicable to certain application domains, such as the telco domain from our running example. The case study is used to examine which non-functional requirements map to domain agnostic attributes, and which requirements are only relevant in the specific case of Phonyfone, hence, map to domain specific attributes. In the last part of this section, we present an *implementation blueprint* for the QoS model. This blueprint was also used for the proof of concept implementation that will be discussed in Chapter 7. Additionally, we briefly present the fundamental *service model* that is assumed for all contributions of this thesis.

The third and final section concludes this chapter by summarizing the key points that were presented.

4.1 Motivation

The term Quality of Service can refer to different aspects. We encounter Quality of Service in our daily routine, be it customer service in your preferred coffee shop, voice quality in mobile networks or the page load time of an online trading platform. From a technical perspective, QoS is a concept that originally stems from the telco domain. The International Telecommunication Union (ITU), defines the term Quality of Service as the *Totality of characteristics of a telecommunications service that bear on its ability to satisfy stated and implied needs of the user of the service.* [83].

With regard to service oriented computing, QoS is a broad concept that covers various non-functional properties [41, 108], including response times, availability or service provider reputation [164, 119, 93]. These properties can be used to benchmark services, e.g., to determine whether a service performs poorly or is unreliable [100, 97]. In scenarios where several services are available for a specific task, such a quantification can be leveraged to select the most suitable service out of a list of available alternative services [106]. This also holds true if a service is benchmarked as a part of a service composition. In this case, the QoS of the service taking part in the service composition has a direct impact on the QoS of the service composition [85, 99]. This becomes evident if we consider the non-functional requirements of the SubscriberActivation process, where the execution time of the ManagedRoaming service composition must not exceed 2500ms. Since the ManagedRoaming composition invokes services sequentially, the accumulated response times of all invoked services, plus the time required for internal processing, forms the execution time of the service composition. Thus, a single unresponsive or faulty service can violate this execution constraint. On the other hand, if there are alternative manifestations of a service available, which is for example the case with the RegisterRoamerAbroad service, then the service with the lowest response time can be selected instead of the service originally referenced in the service composition definition. Hence, QoS can be used to effectively optimize the execution of a composite service by quantifying its non-functional properties.

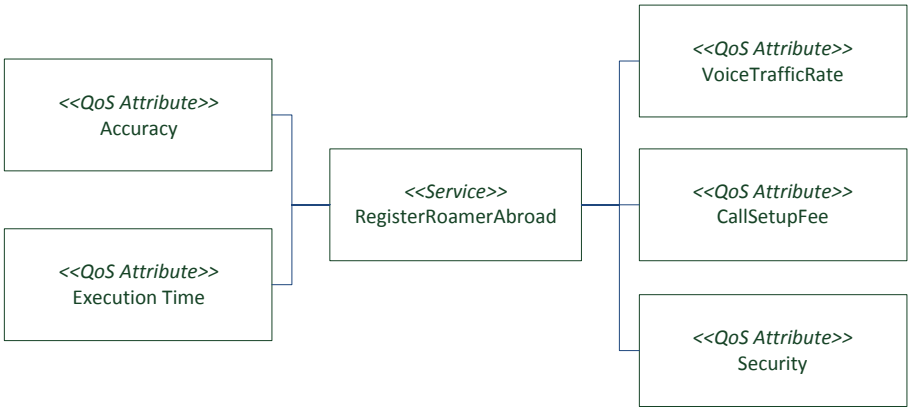


Figure 15: QoS Model for Service RegisterRoamerAbroad

So far, we have examined the non-functional properties of SOC systems in a rather informal fashion. Reasoning about these non-functional properties requires a more formal model. This formalization is provided by the means of a *QoS model*. It represents non-functional properties as *QoS attributes* that are associated with the related service or composition. Hence, it organizes the different QoS attributes for a particular use case, e.g., a service such as the RegisterRoamerAbroad service. Figure 15 exemplifies the QoS model of the RegisterRoamerAbroad service.

When considering the properties of the QoS attributes shown in Figure 15, one can see that QoS attributes differ in several regards. One such regard is the class of a particular QoS attribute. Rosenberg et al. [139] group QoS attributes into the four basic classes *Performance*, *Dependability*, *Security and Trust* as well as *Cost and Payment*, as shown in Figure 16. As an example, *Accuracy* of service RegisterRoamerAbroad is an attribute of class Dependability, whereas *CallSetupFee* belongs to class Cost and Payment. Although we will also use this classification in the following, it lacks information about important properties of a QoS attribute, such as measurability and objectiveness. Likewise, diverse domains feature different requirements with respect to service quality, which are not reflected in such a taxonomy. Details about the organization of the QoS attributes in our QoS model are provided below.

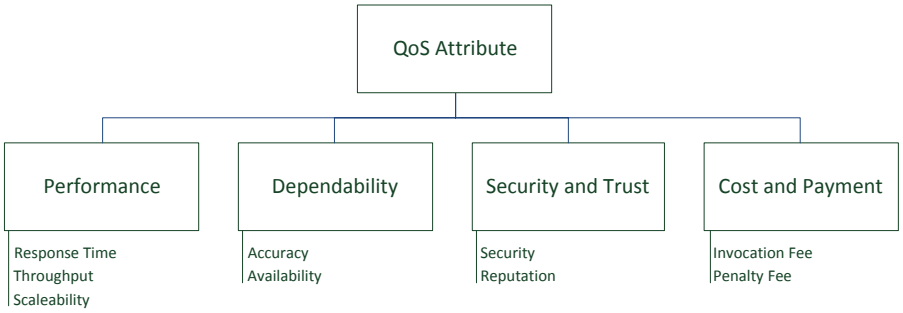


Figure 16: Classification of QoS Attributes, adopted from [139]

Considering Phonyfone, it is important to note that such enterprises are subject to certain market situations. Both regulators as well as marketing requirements can impose constraints on enterprises. A regulator could define the maximum price for text messages originating from a foreign network to protect consumers from high roaming fees. On the other hand, marketing departments are often obliged to react on short-term offers by the competition. In many cases, being the first operator to launch a new product or a promotional offer is the key factor that decides about success or failure. As an example, consider Phonyfone launching a product package that includes the free usage of a value added service, such as mobile TV. Such mobile TV solutions are provided by third party media gateways. Keeping the inter-enterprise fees between Phonyfone and the various media gateway partners low is required to make the product package profitable. However, a media partner could decide to charge different rates for different content, such as a free news channel in contrast to expensive adult content, or simply increase the rates

on existing content. All too often, such market situations cannot be foreseen, and additional QoS attributes may need to be added to the QoS model. Ideally, the implementation of such a QoS model supports runtime adaptivity. This allows to dynamically tailor the QoS model to specific needs and domains, such as the telco domain. Static QoS model implementations or implementations that allow only design time modifications do not support the adaptivity required by contemporary enterprises such as Phonyfone.

4.1.1 Key Requirements

We propose an *Adaptive QoS Model* implementation to address the concerns that were discussed above. There are several key requirements for such an QoS model, which will be listed in the following.

- **Extensible Model.** The fundamental model that is used to classify QoS attributes must be open for extension. This requirement stems from the fact that non-functional requirements might change over time, and additional QoS attributes may be needed.
- **Support for Domain Specific Attributes.** Besides support for QoS attributes that are applicable regardless of the application domain, the model should provide support for domain specific attributes. This support enables a more fine grained mapping of non-functional requirements to QoS attributes.
- **Runtime Adaptivity of the Model Implementation.** Besides the extensibility of the foundational QoS model, the corresponding implementation should provide support for runtime adaptivity with regard to the supported QoS attributes.

In short, such an adaptive QoS model implementation eliminates the need for a system downtime when adding, changing or removing QoS attributes, as well as provides the capabilities to cover domain specific needs. In what follows, we will examine such an adaptive QoS model in detail.

4.2 Theory of Operation

Considering the supported QoS attributes, our model classifies QoS attributes using the dimensions of *determinism* and *applicability*. This classification has several distinct properties which are examined in this section.

At first, QoS attributes differ with respect to *determinism*. For certain domains, it might be well suited to observe only QoS attributes whose values are known in advance. These values are not intended to constantly change during the service offering, such as the VoiceTrafficRate attribute of the RegisterRoamerAbroad service. It is contracted by the service offering party, e.g., IPlus, and the service consuming party, i.e. Phonyfone, prior to the service invocation and will rarely change. On the contrary, values of other QoS attributes might change over time. The response time of the roaming partner services in our ManagedRoaming composition is a QoS attribute whose values are subject to constant change. Therefore, we follow related work [164] and distinguish between *deterministic* and *non-deterministic* QoS attributes. Deterministic

QoS attributes are attributes whose values are known prior to service invocation. Values of non-deterministic QoS attributes are not known in advance and require constant observation. In the following, the supported deterministic and non-deterministic QoS attributes are discussed.

4.2.1 Deterministic QoS Attributes

The supported deterministic QoS attributes of our QoS model include *cost*, *penalty* and *security*. These attributes were adopted from [139] and are discussed below.

Invocation Fee

The invocation fee for an operation o provided by service s refers to the monetary amount that is due for a successful invocation of o. This amount has to be paid by the client c that triggered the invocation of o. It can be charged on a per invocation basis, or in bundles, e.g., 10 monetary units for 100 invocations.

Penalty Fee

The penalty fee for an operation o provided by service s refers to the monetary amount that is due for an invocation of o that failed on the provider side. This amount has to be cleared by the provider of s to the client c that triggered the invocation of o on s.

Security

The security of a service s refers to the degree of security capabilities that the provider of s supports, including authentication, authorization and accounting [78]. Such security capabilities can range from transport level security (TLS) [79] to service specific security features [109]. The security of s can be expressed either on a scale using a numerical value, e.g., 3 of 10, or using boolean expressions, such as $hasSecureChannel(s)\&\&hasBasicAuthentication(s)$.

4.2.2 Non Deterministic QoS Attributes

As described above, the values of non-deterministic QoS attributes change over time and need constant measuring. In the following, the non-deterministic QoS attributes that are supported by our QoS model are introduced. If not mentioned otherwise, we follow the definitions provided in [139]. For a quick overview, the reader might also refer to Table 3.

Response Time

The response time of an operation o provided by service s is defined as the time needed for sending a message from a client c to s in order to invoke o until the response message returns back to c. The average response time of an operation o provided by a service s, which will be used later as an important performance indicator, is calculated using the formula

$$\frac{1}{\#requests} \sum_{i=1}^{n} rt_i \tag{4.1}$$

where the value rt_i, $0 < i < n$, denotes one observation of the response time of o on s in a series of n response times of operation o on service s.

Throughput

The number of requests r for an operation o that can be processed by a service s within a given period of time is referred to as throughput. Using seconds as the time period under consideration, it can be calculated by the following formula

$$\frac{r}{second} \tag{4.2}$$

Availability

The availability of service s defines the probability that s is reachable and produces expected results. The availability can be calculated using the following formula

$$1 - \frac{downtime(s)}{uptime(s)} \tag{4.3}$$

where *downtime* refers to a duration in seconds where s was not reachable while *uptime* refers to a duration in seconds where s was reachable and produced expected results.

Accuracy

The accuracy of an operation o provided by service s is defined as the success rate produced by o. It is calculated by recording all invocations of o within a given time interval and relate it to all failed requests during that interval. The following formula expresses this relationship

$$1 - \frac{\#failed\ requests(o)}{\#total\ requests(o)} \tag{4.4}$$

4.2.3 Domain Specific Quality of Service

So far, we have conducted our discussion of supported QoS attributes with regard to the determinism dimension. However, QoS attributes can further be distinguished regarding their *applicability*. In this regard, some QoS attributes, such as security or availability, are common to many service implementations and application domains. Such attributes are *domain agnostic* in the sense that their applicability is not restricted to a certain business domain. *Domain specific* QoS attributes, on the other hand, are viable in their specific business domain only. As an example, consider the CallSetupFee from the RegisterRoamerAbroad service. Although essential for our ManagedRoaming composition, this attribute is of no relevance for other services or compositions, such as purchase order or loan approval scenarios. Figure 17 exemplifies the QoS attribute classification using non-functional requirements of the SubscriberActivation process.

A benefit of such an additional dimension is that it allows to model QoS concerns more appropriate to a specific use case or problem domain. Hence, stakeholders are enabled to express

QoS Attribute	Formula	Unit	Class	Deterministic
Average Response Time	$\frac{1}{\#requests} \sum_{i=1}^{n} rt_i$	milliseconds	Performance	no
Throughput	$\frac{\#requests}{second}$	requests per second	Performance	no
Availability	$1 - \frac{downtime}{uptime}$	percentage	Dependability	no
Accuracy	1 - $\frac{\#failedrequests}{\#totalrequests}$	percentage	Dependability	no
Security	n/a	n/a	Security	yes
Cost	n/a	monetary value	Security	yes
Penalty	n/a	monetary value	Security	yes

Table 3: Supported Domain Agnostic QoS Attributes

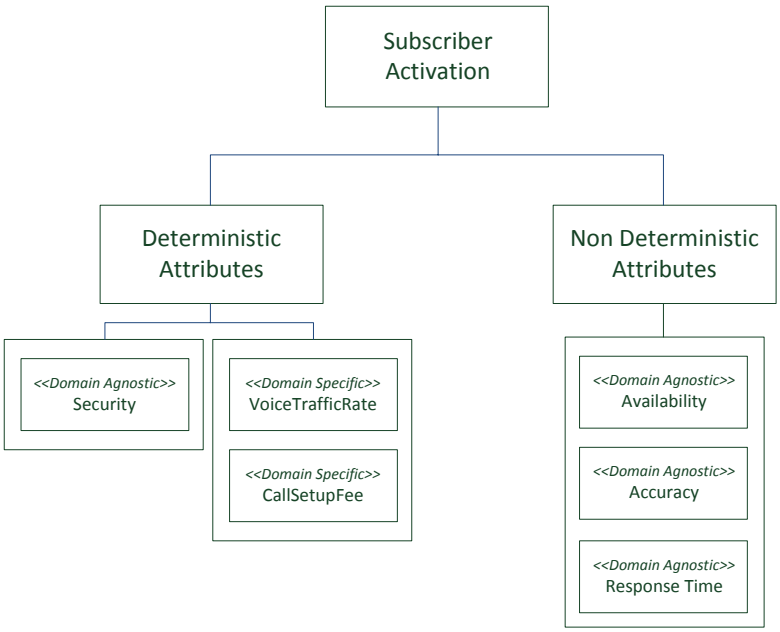


Figure 17: Overall SubscriberActivation QoS Model

their specific QoS requirements in a more natural manner, as it would be possible if there would be generic attributes available only. Another important aspect is the matter of granularity. When we again consider the CallSetupFee from the RegisterRoamerAbroad service, one could argue that it could be modeled simply using the domain agnostic attribute *Invocation Fee*, which we

explained previously. However, the *VoiceTrafficRate* is yet another cost related attribute that needs consideration with regard to RegisterRoamerAbroad invocations. The value of this attribute is not directly related to the value of the CallSetupFee, and might differ. With only the Invocation Fee attribute available, there would be no distinction between those two important attributes. This would severely constrain the mapping of non-functional requirements to QoS attributes, and in the case of Phonyfone, would prohibit to minimize each attribute separately, as described in Chapter 3.

Taken together, the fundamental assumption of our QoS model is that QoS attributes can be classified as either deterministic or non deterministic attributes, with the additional distinction of attributes as being domain agnostic or domain specific. Figure 17 depicts our overall QoS attribute classification.

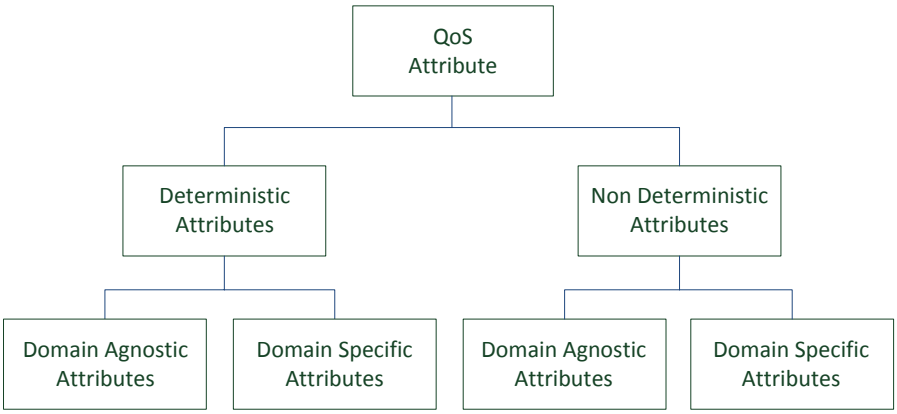


Figure 18: Overall QoS Classification

4.2.4 Design Considerations

In the following, we will present a blueprint for the implementation of the QoS model that we examined above. At first, however, we will consider the fundamental *service model* that is assumed for all contributions that are discussed hereinafter.

Service Model

The service model that we assume makes use of the terms and concept discussed in Chapter 2. Hence, the fundamental entity in our model is a *service* that provides *operations* and is described by an *interface description*. The service itself is an abstract concept that is realized by either the means of a *service endpoint* or a *service composition*. Both realizations inherit certain base properties of service, such as a service name or address. The relationship of the aforementioned

entities is depicted using an UML class diagram in Figure 19. Please note that class members are excluded for brevity.

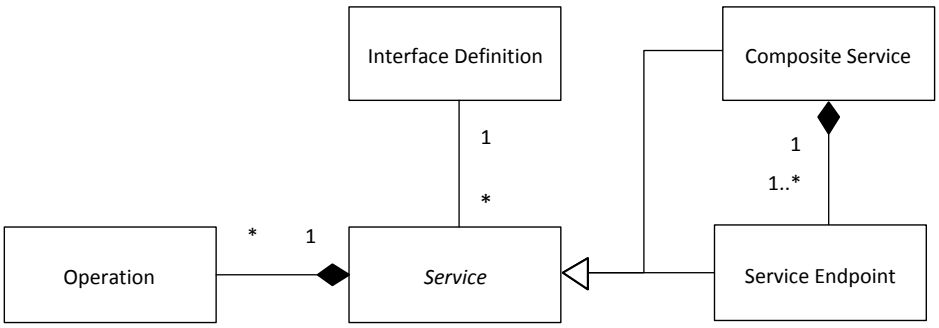


Figure 19: Service Model Class Diagram

QoS Model

In Figure 20, we have depicted a blueprint for the implementation of the QoS model that was the main matter of this chapter. Please note that we do not distinguish between ServiceEndpoints and CompositeServices in this model, since for our purposes, a CompositeService will always be exposed by the means of a ServiceEndpoint. This assumption is fundamental for the contributions that will be presented in Chapter 5 and Chapter 6.

Following our fundamental service model from Figure 19, our basic entities are `ServiceEndpoints` and `ServiceOperations`, whereas a `ServiceEndpoint` might have an arbitrary number of `ServiceOperations`. A `ServiceEndpoint` is uniquely identified by its `name` and its `endpointAddress`, while a `ServiceOperation` is uniquely identified by its `name` within the context of its enclosing `ServiceEndpoint`. This means that there might be several `ServiceOperations` with the same name, but no two `ServiceOperations` must share their name within the same `ServiceEndpoint`.

Each `ServiceOperation` is linked with a `QualityOfServiceData` entity that represents a container for QoS related information. This `QualityOfServiceData` container manages a list of `QualityOfServiceAttributes`. These `QualityOfServiceAttributes` represent QoS attributes in our model, such as accuracy, response time or penalty fee. `QualityOfServiceAttributes` feature a `name`, `description`, `valueType` and `type` attribute. The `name` attribute identifies the `QualityOfServiceAttributes`, while the `description` attribute provides more information on the propose of the attribute. This description attribute is predominantly helpful in the context of domain specific attributes, where the purpose might not always be clear for users that are not familiar with the related business domain. The `type` attribute classifies `QualityOfServiceAttribute` as either determin-

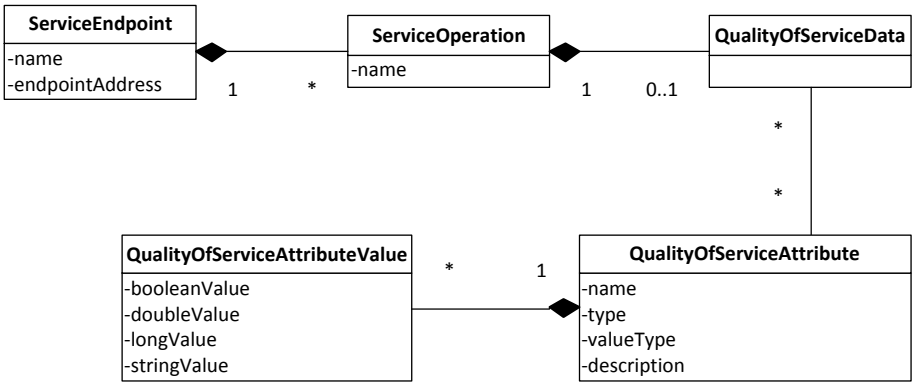


Figure 20: QoS Model Class Diagram

istic or non-deterministic (D/N) and domain agnostic or domain specific (A/S). As an example, the CallSetupFee from Figure 17 has set the value `DS` for attribute `type` to indicate that it is a deterministic and domain specific attribute. The assignment of concrete values for a `QualityOfServiceAttribute` is represented by the `QualityOfServiceAttributeValue` class. This class features members that represent the `booleanValue`, `doubleValue`, `longValue` or `stringValue` of the corresponding `QualityOfServiceAttribute`, depending on the `valueType` attribute of the linked `QualityOfServiceAttribute`.

One might argue that the design of the QoS model would be more concise if we would represent QoS attributes as members of the `QualityOfServiceData` class directly and with an appropriate type information. However, and contrary to QoS models that predefine QoS attributes of a specific type, our model uses a more generic approach to remain open for future extensions. This extensibility is necessary to add additionally required QoS attributes on demand, as previously described. Moreover, this "key-value" approach for representing a particular QoS attribute enables a rather simple implementation of the QoS model.

4.3 Summary

This chapter presented our QoS model for the representation of non-functional requirements in service oriented systems. We argue that there are certain important aspects for such a QoS model, in particular with regard to its extensibility. Besides the extensibility of the formal model, an original aspect of our approach is the *runtime adaptivity* of the model implementation. Such an adaptive QoS model implementation allows not only design time changes but enables the modification of the model at runtime. This enables us to add and modify QoS attributes that might be additionally needed on short term, without disrupting the execution of systems that rely on the QoS model. An example for such a system that relies on this QoS model is the monitoring system that we will examine in the following chapter.

CHAPTER 5

Holistic Monitoring for Composite Services

This chapter presents our approach for holistic monitoring of composite services. The first section of this chapter reasons about the need for *monitoring of composite services* and why this subject requires special considerations in large scale heterogeneous enterprise settings. Our running example will be used to motivate this need and to carve out the monitoring requirements in such enterprise environments.

In the second section, we present the theory of operation of our monitoring approach. This section will emphasize on the use of *event stream processing* technology to analyze and inspect situations of interest that can be derived from raw monitoring data. The section reviews the event model that we adhere to and briefly explains the event processing language that is used. Finally, we examine a method for determining *service interface compatibility* issues in composite services.

The third and final section of this chapter assembles the insights of the previous sections into a *system architecture*. There, we will discuss the architectural aspects of such a monitoring system. This system architecture also serves as a blueprint for the proof of concept implementation that will be presented later in this thesis. A short summary at the end of this chapter concludes the discussion of our monitoring approach for composite services and gives some final remarks.

5.1 Motivation

Contemporary enterprises rely on a multitude of heterogeneous technologies. This heterogeneity in IT landscapes has various reasons. First, there are historical reasons that force enterprises to support legacy systems. Although a considerable amount of research is dedicated to the subject of legacy systems modernization [44, 55, 18, 134] and guidelines from industry on related migration strategies exist [105], migrating legacy systems to unified platforms and architectures remains problematic. Such migration strategies put high demands on a company resources, both

from a staff as well as monetary point of view. On the other hand, legacy systems cannot be switched off easily, as they perform crucial task within an organization [35]. Therefore, in many enterprises, legacy technology and platforms continue to exist and their migration is progressing only at a slow pace. Second, different products from different vendors provide different features. Hence, a certain product might be better suited for a particular job or application than a product from a different vendor. A similar reason for enterprises to adopt new products in IT is market competition. New product offerings from market competitors might require the adoption of new IT systems, oftentimes leading to competing technologies and platforms within an enterprise IT landscape. Finally, there is a political dimension to the dilemma of heterogeneity in IT. In certain cases, enterprises are bound by contractual obligations to adopt the product solution from a particular vendor. Hence, such enterprises are prohibited to streamline their IT landscapes, since they are bound to a specific vendor.

With regard to monitoring, such a heterogeneity in IT leads to several important issues that affect service composition platforms. On the one hand, we claim that it promotes a *fragmentation of monitoring data*. This stems from the fact that service composition platforms from different vendors are not required to adhere to a certain monitoring standard. As an example, the log file formats or notification mechanisms used by different vendors of composition engines are rarely compatible. One reason for this drawback is the lack of a specification of monitoring support within a certain composition technology itself. For instance, the WS-BPEL standard [110] lacks any specification of monitoring support.

Moreover, large enterprise business processes, such as the SubscriberActivation scenario from our running example, *span across several composite services*, in our case the AddSubscriber and ManagedRoaming composite services. As these compositions represent the backbone of Phonyfone, the need for monitoring is evident. In Chapter 3, we described that there is a non-functional requirement of Phonyfone that spans across the AddSubscriber and ManagedRoaming compositions. Hence, it is necessary to provide the means to correlate the monitoring data for both service compositions. However, we further noted that these composite services might be executed on composition platforms from different vendors. The *lack of a common representation of monitoring data* prohibits such a correlation of monitoring data from different service compositions.

Another important issue related to heterogeneity arises when considering the need for an *integration of external data* into the monitoring of composite services. In our SubscriberActivation scenario, the final step in the process is the sending of a confirmation email to the customer that includes important account data. The SubscriberActivation process is not complete until this email is sent to the customer. Ideally, the monitoring system that observes the AddSubscriber and ManagedRoaming compositions is capable of accepting a notification for the transmission of this email. Hence, such a monitoring system can observe the lifecycle of the SubscriberActivation process in an end-to-end manner.

As outlined in Chapter 4, QoS is an important aspect of composite services that can be used to quantify the operational state of the composition and its individual services. The importance of this subject is also emphasized by the large body of research in this area [161, 144, 70, 22, 33, 32, 86] that considers the issues related to monitoring of composite services. Additional to the performance and dependability related QoS attributes, there are certain business related

monitoring criteria that need consideration. Such business related criteria are crucial as they enable the derivation of key performance indicators (KPI). One example for such a KPI is the number of subscriber activations during the working hours of Phonyfone (cf. Section 2). If this number is below the expected amount of 600 to 700 activations, it might indicate a potential problem with the SubscriberActivation process. If, on the other hand, it exceeds the expected numbers, it can be used to derive customer satisfaction with new or enhanced products. Thus, such KPIs can be used to *detect anomalies* in the operational aspects of composite services.

The last important aspect that we will consider is related to changes of service interface descriptions. In general, the stability of service interface descriptions is essential for the normal operation of composite services. This is due to the fact that certain service interface changes inevitably introduce an incompatibility between the composite service and the individual service that was affected by the interface modification. As stated in Section 3.2.3, this situation is evidently worse in cases where the individual services of a composite service are not hosted on premises. Hence, it is highly desirable to *detect changes in service interface descriptions* that can disrupt the normal operation of a composite service.

5.1.1 Key Requirements

We will address the issues that were outlined above with a *holistic monitoring approach for composite service platforms*. To this end, we use the term holistic to summarize the following distinct key features of such a system:

- **[M1] Platform agnostic and unobtrusive**. The monitoring system must be platform agnostic. This means that it must be independent of any concrete composition technology and runtime. Additionally, it should be unobtrusive to the systems being monitored. No modifications within the composition engine should be necessary to interact with the monitoring runtime. This feature is in line with Key Principles **[P0]** and **[P1]**.
- **[M2] Integration with other systems**. The monitoring system must be capable of integrating monitoring data from other subsystems outside the composition engine. Such subsystems include databases, message queues or other internet applications. This feature is crucial for providing end-to-end monitoring capabilities in large scale enterprise business processes that rely on functionality provided by systems outside the composition platform.
- **[M3] Multi-composition monitoring**. The monitoring system must support monitoring across multiple composite services, and their instances. This stems from the fact that composite services often cannot be monitored in isolation because they influence each other. This feature is crucial for providing end-to-end monitoring capabilities in large scale enterprise business processes that span across several service compositions.
- **[M4] Detecting anomalies**. The monitoring system must be capable of unveiling potential anomalies in the expected behavior of composite services and its individual services. Such anomalies can reveal problems in associated subsystems as well as indicate changes in customer behavior. This enables maintaining the normal operation as well as provides the foundation for certain optimizations of service compositions.

- **[M5] Interface compatibility checking**. The monitoring system must provide the means to determine incompatibilities in the interface descriptions of evolving individual services in a service composition. That is, in case the interface description of a service changes, a mechanism must be provided to check if the related changes break the compatibility between the composite service and its constituent services.

In the following section, we will describe the theory of operation of such a holistic monitoring approach.

5.2 Theory of Operation

When considering the monitoring requirements of composite services that we outlined above, it is evident that a simple log file based monitoring is not sufficient. Rather then yielding the monitoring responsibility to the provider of the composition engine, which would lead to the problem of fragmentation as discussed above, a common representation of monitoring data is required. Such a common representation abstracts from the details of the underlying technology and enables a platform agnostic handling of all monitoring concerns. The fundamental abstraction that we will leverage in our monitoring approach will be based on the following two assumptions:

Assumption 1. *The interaction of services within a composite service is solely realized by the means of a message exchange.*

Assumption 2. *A composite service is exposed by the means of a service. This means that a composite service is triggered by a request message and returns a response message upon completion.*

Under these assumptions, we can represent the message exchange as a *stream of events*. Each message that is sent or received as a part of a service composition is mapped by a corresponding event. This also means that there is a certain temporal ordering of events that is determined by the particular timestamp when the message was sent or received. Figure 21 provides a high-level overview of the event based abstraction of message exchange in composite services that we use for our monitoring approach. Please note that in Figure 21, `Service A` and `Service B` are invoked using different message exchange patterns (MEP) [71]. `Service A` is invoked by the composition engine using the request-reply MEP, while `Service B` is invoked via the one-way MEP.

In summary, our approach for holistic monitoring of composite services is based on an *event driven* model of the message exchange in composite services. The following section discusses the details of such an event driven monitoring method.

5.2.1 Event Driven Monitoring of Composite Services

As outlined above, our monitoring approach operates on the message level. This makes the system we propose agnostic to implementation details of the monitored platform, as we do not rely on any specific features of a particular composition engine implementation. Hence, it allows

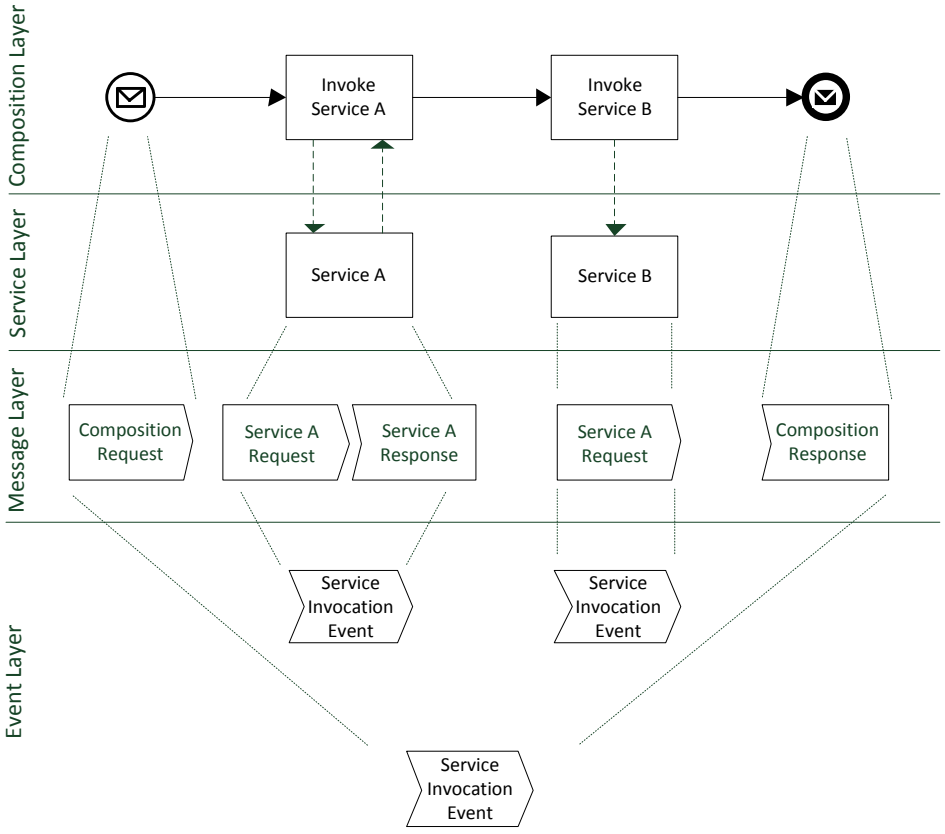


Figure 21: Event Based Abstraction for Message Exchange in Composite Services

us to address Key Principle **[P2]**. In general, any composition platform that uses message based interaction of participating services can be monitored by our approach. Therefore, our method supports a large variety of current and future composition platforms.

Clearly, some messages are dependent on other messages, both from a causal as well as temporal point of view. In our running example, the successful execution of the AddSubscriber composite service is a prerequisite for the execution of the ManagedRoaming composition. Hence, a request message for the ManagedRoaming composition without a related and prior request message for the AddSubscriber composition is an strong indicator for a problem. Therefore, a supporting technology should enable us to model such dependencies in a way that the monitoring system can use this information to detect similar situations of interest.

Event stream processing technology (ESP) allows us to reason about this stream of events and respect temporal and causal dependencies between events, such as the coupling of AddSubscriber and ManagedRoaming events described above. Furthermore, we can combine the required presence of both the AddSubscriber and ManagedRoaming events into more coarse grained events. In this regard, the atomic events that represent a single message in the message exchange are referred to as *base events*, whereas a structured combination of base events is referred to as *composite events*.

Event Model

Evidently, the monitoring approach we propose is built around the concept of events. From various available definitions [39, 94, 58, 63], we follow a rather technical view of an event as *a detectable condition that can trigger a notification*. In this context, a notification is an event-triggered signal sent to a runtime-defined recipient [63]. In the Phonyfone scenario, each monitoring requirement breaks down to a detectable condition that can be represented by events of various types. In this regard, the term *event type* refers to the formal description of a particular kind of event, whereas the term *event object* represents an instance of a particular event type that was triggered in response to a situation of interest.

An event type describes the data and structure associated with an event. Event types have several *event attributes*. The most essential attribute is a timestamp that enables us to derive a total order of the elements of the event stream. Clearly, the various event attributes are strongly related to the use case of the event type. For example, our approach supports an *service invocation event* type. This event type describes events that are triggered if a composition or an individual service that takes part in a composition is invoked.

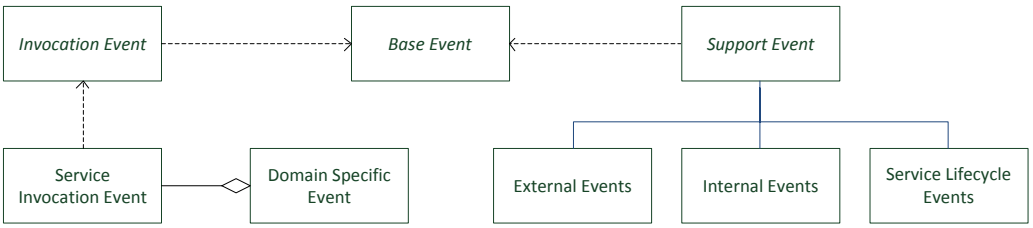


Figure 22: Event Model

To reason about and to provide a generic way to deal with the various event types, an *event model* is needed. Similar to the QoS model that we introduced in Chapter 4, such an event model organizes the different event types and generalizes them by introducing an event hierarchy. Thus, it is possible to leverage simple base events to define more specific and complex events. Figure 22 shows the event model that we use.

The type *base event* is the most general event type that we use in our event model. It provides the fundamental attributes that are required to build a total ordering of events. One level below the base event type, the model distinguishes between *invocation events*, *domain specific events*

and *support events*. Invocation events and support events are abstract event types that inherit from the base event type, whereas domain specific events aggregate service invocation events, which, in turn, are a specialization of invocation event. This aggregation of service invocation events provides use-case specific event types. All of these event types feature a variety of event attributes. These attributes are summarized in Table 4 for a quick overview. In the following, we will explain these event types in more detail.

Base Events and Invocation Events Both base events as well as invocation events are abstract event types whose event objects are not directly instantiated. These event types are solely provided as a means for abstraction, and provide the basic event type attributes that are required by other event types in our model. Most importantly, base events provide a timestamp that records the time at which an event occurred, as well as a unique identifier that is associated with each event object. Additionally, a generic map is provided to store other metadata that is not reflected by any event type attribute in the hierarchy. Please note that the invocation event type abstraction is provided to keep the model extensible, e.g., to support invocation events that are not service based.

Service Invocation Events An event object of type service invocation event is created in two situations. First, when a service participating in a service composition is invoked by the composition engine. Second, a service invocation event is also triggered if the service composition itself is invoked by the consumer of the composite service. As listed in Table 4, a service invocation event features a multitude of event type attributes. These attributes include the payload of the request and response messages that were sent from the composition engine to the individual services and vice versa. Common base attributes are inherited from type invocation event (cf. Figure 22). This type of event is used in our monitoring system to cover, for instance, non-deterministic QoS related metrics, such as service response times, availability or accuracy. The values for these QoS attributes are calculated by applying the `duration` or the `success` indicator of the service invocation event type to the related formula (cf. Section 4.2.2).

Domain Specific Events Domain specific events represent a special case in our event model. This event type does not introduce additional event type attributes, but composes service invocation events into new event types that have a special meaning for certain use cases. In this regard, domain specific events are similar to domain specific QoS attributes that were described in Section 4.2.3, as their applicability is restricted to a certain application domain. As an example, we can build the domain specific event type `SubscriberActivation` event by composing two service invocation events `AddSubscriber` event and `ManagedRoaming` event.

Clearly, there are certain matters that need consideration when composing events. In our example, only related `AddSubscriber` and `ManagedRoaming` events can be composed into a `SubscriberActivation` event. Hence, domain specific events have to provide a way to correlate the service invocation events that they are comprised of. To this end, our approach uses the message payload that is available via the service invocation event type to extract information from the related request and response messages. This information can be effectively used to correlate service invocation events and compose them into a domain specific event such

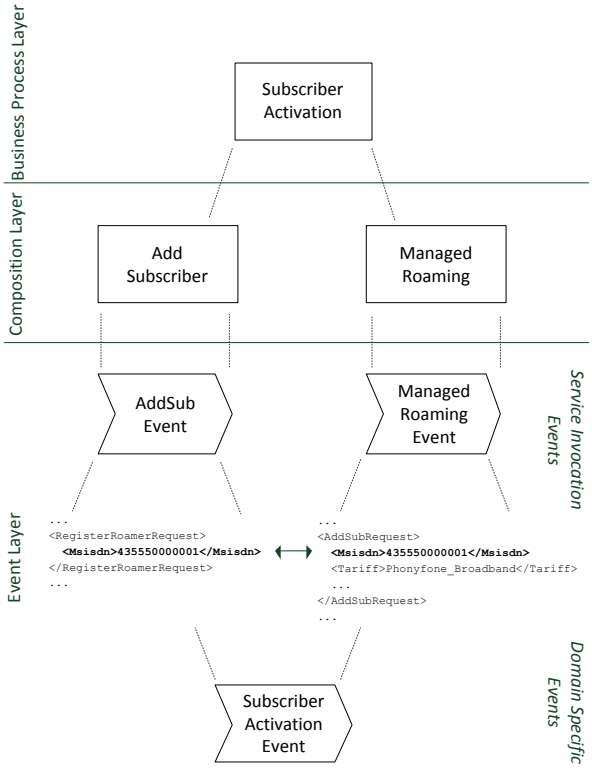


Figure 23: Composition of Service Invocation Events into Domain Specific Events

as the `SubscriberActivation` event. Considering the request messages that trigger the AddSubscriber and ManagedRoaming compositions, the `MSISDN` of the subscriber is used for those requests to identify the related customer. Hence, it can be used to match previously unrelated request messages for the AddSubscriber and ManagedRoaming compositions. Figure 23 visualizes the composition of the two service invocation events into a domain specific event.

Support Events The last class of event types that our model provides are support events. Additional to the event types described above, which are directly related to the message exchange within a service composition, this class of event types contains event types to represent *internal* and *external* events, as well as *service lifecycle* events. Internal events are triggered by the monitoring system to notify the monitoring runtime system about certain situations related to the inner workings of the monitoring system. On the other hand, service lifecycle events are emitted when a change in a service is detected, such as changes in the service interface description. Such events play a role for the service interface surveillance method that we will describe

below. Finally, external events refer to situations of interest that occur outside of the monitored composite service. The need for representing external events within the monitoring system has already been stressed above.

Event Type	Attribute	Data Type	Description
Base Event	timestamp	Timestamp	records the time at which the event occurred
	identifier	String	uniquely identifies each event object
	metadata	Map	a map that can be used to store generic event metadata
Invocation Event	requestSent	Timestamp	records the time when the request message was sent
	responseReceived	Timestamp	records the time when the response message was received
	duration	Long	duration between `requestSent` and `responseReceived`
	success	Boolean	indicates whether the invocation was successful
Service Invocation Event	serviceName	String	name of the service that the request was sent to
	endpointAddress	String	the address of the endpoint where the service is located
	operationName	String	name of the operation that was invoked
	requestPayload	String	the request message payload
	responsePayload	String	the response message payload

Table 4: Summary of Event Type Attributes

Event Processing Language

So far, we have described that our monitoring approach relies on an event stream that is comprised of event objects that have a specific event type. Furthermore, we argued that these events can represent certain situations of interest, ranging from simple QoS related metrics to complex business related metrics. The next building block that we require for our event driven monitoring system is an entity that supports certain operations on the event stream. *Event processing agent*s (EPA) provide such functionalities, including the filtering of events, detecting patterns of events and transforming events [58]. However, the EPA cannot operate on the event stream without a precise description which operations should be applied to the event stream. This description is provided by the means of an *event processing language* (EPL).

As described in Chapter 2, such an EPL is similar to SQL of relational databases, but provides distinct features that make it more appropriate than SQL for our needs. Given the list of requirements that we have enumerated in Section 5.1, we can derive two requirements that are crucial for an EPL. First, the EPL must provide capabilities to model complex *patterns* of events. Considering the coupling between invocations of the AddSubscriber and ManagedRoaming composite services in our SubscriberActivation scenario, this sequence can be modeled as the following event pattern:

```
1 AddSubscriber → ManagedRoaming(msisdn = AddSubscriber.msisdn)
```

The arrow (→) denotes that a ManagedRoaming event follows an AddSubscriber event. Moreover, we need to specify that the MSISDN included in both messages need to be the same. A *filter criteria* (`msisdn = AddSubscriber.msisdn`) can be set to connect several events through a common attribute, such as the MSISDN. Considering our requirements list, this property effectively tackles requirement **[M3]** as it enables us to model monitoring requirements that span across several composite services. Moreover, besides detecting the presence of

a particular event, the EPL should also support detecting the absence of an event, e.g., where the PhonyWeb Terminal issues an AddSubscriber request but no related ManagedRoaming request.

Second, the EPL has to support event stream *queries* to cover online queries against the event stream, ideally using a syntax similar to existing query languages such as SQL. This feature can be used to analyze specific information about the business domain, e.g., the number of failed subscriber activations during a particular time window or even the related MSISDN:

```
1 SELECT msisdn
2 FROM AddSubscriber.win:time(15 minutes)
3 WHERE success = false
```

This query returns all MSISDNs of AddSubscriber requests that failed within the last 15 minutes. Furthermore, event stream queries can be leveraged to deal with marketing specific requirements such as finding the most wanted postpaid tariff of the last business day:

```
1 SELECT count(tariff)
2 FROM AddSubscriber.win:time(24 hours)
3 WHERE success = true
4 GROUP BY tariff
```

Reasoning about the potential abuse of the RRC feature that we discussed in Chapter 3 leads us to yet another EPL requirement. Detecting an unusual high number of requests with a reduced recurring charge can be handled by defining the following event pattern:

```
1 [5] (a = AddSub → b = AddSub(
2         originator = a.originator
3         AND reducedRecurringCharge.amount > 0
4         AND a.reducedRecurringCharge.amount > 0).win:time(1 day))
```

The repeat operator ($[n]$, $n > 0$) triggers when the statement following the operator evaluates to true n times. However, we need to know which shop assistant triggered the activation. Thus, the final requirement for our EPL is the combination of both event pattern matching and event stream queries:

```
1 SELECT a.originator
2 FROM PATTERN (
3   [5] (a = AddSub → b = AddSub(
4         originator = a.originator
5         AND reducedRecurringCharge.amount > 0
6         AND a.reducedRecurringCharge.amount > 0).win:time(1 day))
7   )
```

This query will show which shop assistant abuses the RRC fields. Taken together, these EPL features enable us to effectively detect business and performance related anomalies, and hence, satisfy requirement **[M4]**.

Considering the list of requirements, there is one requirement left that we need to address. In the following, we present an integrated method to sense service interface mismatches that may arise due to independently evolving services.

5.2.2 Service Interface Surveillance

Besides the event processing capabilities that we leverage in our monitoring method, there is another important aspect inherent to our holistic monitoring concept. *Service interface surveillance* enables our approach to proactively detect changes in service interface descriptions and identify which changes introduce potential incompatibilities that might disrupt the normal operation of the service composition. To this end, we follow the categorization of changes that was introduced by Papazoglou in [123] and used in related work [124, 8]. This classification is based on the various effects and potential side effects that changes may cause. Therefore, it differentiates between *shallow* and *deep* changes. Shallow changes, on the one hand, are localized to a single service or the clients of this service. Deep changes, on the other hand, are a cascading type of changes whose effect may not be localized to a particular service.

Our mechanism for service interface surveillance only considers shallow changes. In this regard, and more precisely, our approach determines whether changes in a new version of a service interface description are *backward compatible* to the old version of the same service interface description. Contrary to *forward compatible* changes, where the focus lies on the support of unknown future consumers, backward compatibility effectively means that changes on the service provider side do not break the compatibility between a service provider and its service consumers. This, in turn, means that service consumers do not require adaptation whenever changes on the service provider side are introduced.

Determining backward compatibility is usually realized by listing all compatible changes to a service interface description [124]. If a change is not tagged explicitly as being backward compatible, it is consider as a change that breaks backward compatibility. The list of backward compatible and incompatible changes that we use for our method is based upon various related work [60, 51, 54]. Table 5 summarizes this list.

Change	**Description**	**Compatible**
Adding a new operation	existing consumers are unaffected as they do not use the new operation	Yes
Adding new optional data structures to the input message	existing consumers are unaware of new optional data structures	Yes
Changing existing input data structures from mandatory to optional	existing consumers are unaffected as they use the data structure as it were mandatory	Yes
Change to the service provider implementation which have no impact to the interface	Service implementation can be modified as long as there is no material impact to the interface	Yes
Removal of an operation	Existing consumers using the removed operation will be affected	No
Changing cardinality of existing output data structures	Changes to the cardinality of fields in the output message, such as changing mandatory fields to optional fields, will break existing consumers	No
Change to the definition of data types	Most changes to the data types in the input or output message are not backward compatible	No

Table 5: Compatibility Matrix of Service Interface Changes, adopted from [51]

Scheduled Service Interface Compatibility Checking

Effectively determining whether an updated version of the service interface description of an individual service is backward compatible to the old version requires that a supporting method has access to both the original and the updated version. That is, we need access to the version of the service interface description that was used when the composite service was originally defined, as well as the updated version of this interface description. To this end, we assume that this interface description is publicly available and can be accessed by the monitoring system via a *interface description resolver* (IDR). Such a IDR serves as an abstraction from the concrete details of how a service provider exposes the service interface description for its services. When we delve into the details of the system architecture in the next section, we introduce the *Monitor* component. For the time being, we only need to know that the Monitor component provides the means to extract certain service metadata, including the service endpoint address. Using the service endpoint address, the IDR tries to determine the corresponding service interface description. Our monitoring runtime leverages this information to perform the service interface compatibility check in two phases. First, it retrieves the service interface description that is assumed to be the original version of the interface description. That is, the interface description that was used during the design of the service composition. If needed, this version can be manually overridden. Second, it periodically retrieves the most current version of the service interface description. Thereby, the service interface surveillance component has access to the original and a potentially updated version of a service interface description. Finally, the interface surveillance component can compare the original and the current version and check the backward compatibilty using the rules from Table 5.

5.3 System Architecture

The previous section presented the theoretical approach and main components of our holistic monitoring method for composite services. In the following, we will use these building blocks to deliver an architectural blueprint for such a monitoring system. Parts of this blueprint are aligned to the MAPE-K architecture, as discussed in Section 2.3. Where applicable, the figures provided below use the symbol conventions that we introduced in Chapter 1 to align components related to the MAPE-K architecture to their corresponding MAPE-K phase.

We decided to divide this section into three parts to prevent the figures from being overloaded and provide a clear understanding of the architectural aspects. First, we will develop an architecture for the event driven monitoring method described in Section 5.2.1. Second, the service interface surveillance method from Section 5.2.2 will be provided with a corresponding architecture proposal. Finally, the last part assembles all components into a big picture that provides a top-level view on the overall system architecture.

5.3.1 Architectural Blueprint for an Event Driven Monitoring System

In Figure 24 we have depicted our proposed system architecture for the event driven monitoring approach from Section 5.2.1. This architecture provides special considerations with regard to the requirements that we stated in the beginning of this chapter. In particular, two components

that are also leveraged in the architecture blueprint for the service interface surveillance method, provide the means for platform and technology agnostic architecture in this regard. These two components are the *Composition Engine Adapter* (CEA) as well as the *Interception and Adaptation Layer* (IAL). Both components adhere to our foundational assumptions **Assumption 1** and **Assumption 2**.

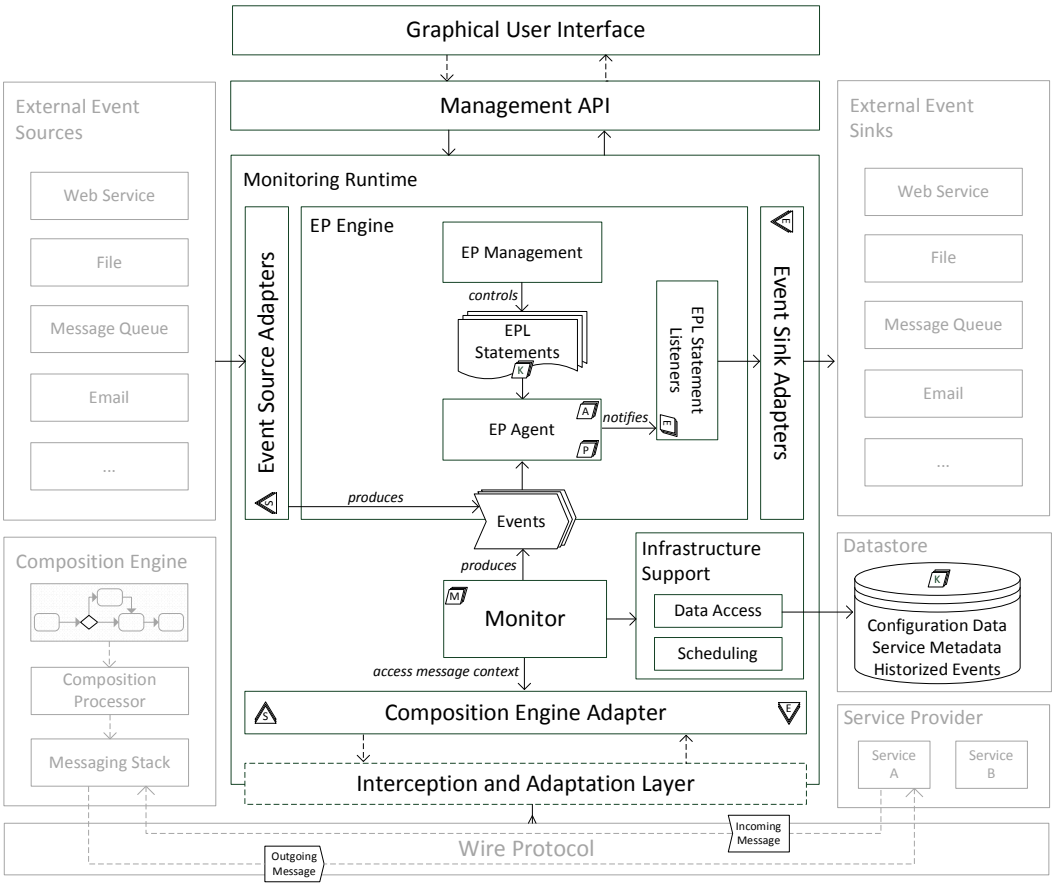


Figure 24: Architecture of an Event Driven Monitoring System

The IAL represents an intermediate layer between the *Monitoring Runtime* and the *Wire Protocol* layer, which provides the foundational technology for the message exchange. It is noteworthy that our approach leverages the Wire Protocol for message interception, and not the underlying transport protocol. The important distinction here is that the Wire Protocol specifies the form or shape of the data to be exchanged between disparate applications, whereas the term transport protocol signifies the method by which that data is transferred from system to system [122]. This approach enables us to work with an higher level of abstraction, which simplifies the realization of such a message interception facility. The IAL, in turn, is responsible for providing the means to transparently tap the message exchange between the *Composition*

Engine and the *Service Provider* of the individual services. Several alternative technologies are available for the realization of such an IAL, however, our requirements towards platform independence and unobtrusiveness constrain the list of implementation technologies to a few candidates. For several reasons that will be emphasized in Chapter 7, our proof of concept implementation leverages *aspect orient programming* (AOP) to implement this component. There are two main features of the IAL. First, it provides access to the message context of incoming and outgoing messages. This message context can be used to analyze the meta data associated with a message, such as the service name or the endpoint address that the message is routed to. Additionally, it provides access to the payload of the message. Hence, the IAL is a fundamental component with regard to the extraction of information that is required to derive corresponding service invocation events.

The CEA, on the other hand, uses the benefits of the IAL to provide an abstraction for the implementation of the composition engine that executes a service composition. As such, it can be seen as the "driver" for a certain service composition engine. The CEA leverages the fact that most available *Composition Processors* rely on some sort of *Messaging Stack* to trigger the service invocations as defined in the composite service definition. Thus, the support for a particular composition engine requires the implementation of a CEA that supports the Messaging Stack used by the Composition Engine. This approach has the advantage that several distinct composition engines that use the same Messaging Stack are automatically supported as long as there exists an CEA for their Messaging Stack. In short, the CEA allows to access and modify the message exchange within composite services. Hence, when considering the responsibilities of the CEA in the autonomic computing context, it resembles the functionality of Sensors and Effectors. Although the message exchange will be used in a read-only manner until this thesis introduces an adaptation environment for service compositions in Chapter 6, the CEA is labeled with both the Sensor and Effector symbol in Figure 24, for reasons of completeness. In effect, the IAL and the CEA together cover requirement **[M1]**.

The *Monitor* is another key component of our system architecture. It serves two main purposes. First, it extracts service related metadata, such as service name and service endpoint URL, and persists this information in a *Datastore* using the *Data Access* component. This metadata is not used by the Monitor component exclusively, but is also crucial to other components, such as the service interface surveillance component. We will refer to a service whose metadata was registered by the Monitor component as a *captured service*. Second, the Monitor translates the message exchange into an event stream. It emits events, such as ServiceInvocationEvents, into the *Event Processing Engine*. The Monitor uses the aforementioned CEA to access the message context of incoming as well as outgoing messages. In the autonomic computing context, the Monitor component covers the MAPE-K phase of the same name.

The events emitted by the Monitor are handed over to the *Event Processing Agent* (EP Agent), which is part of the *Event Processing Engine* (EP Engine) incorporated in our system. The EP Engine provides the capabilities discussed in Section 5.2.1. The EP Agent is the main work horse of the EP related components. Its main responsibility is to provide several important operations on the event stream, such as event filtering, event pattern detection and event transformation. Details about the capabilities of such an EP Agent can be obtained from Chapter 2. The EP Agent deals with the MAPE-K phases Analyze and Plan, as it filters and aggregates event

objects, and prepares queries and event patterns.

The event filtering, pattern detection and transformation specifications that instruct the EP Agent are provided by the means of *Event Processing Language Statement*s (EPL Statement). These EPL Statements are registered with the EP Agent and represent the choice of an event pattern, an event stream query or a combination of both, as described earlier in this chapter. EPL Statements can be seen as a part of the Knowledge inherent to autonomic computing systems, as they resemble the functionality of monitoring policies.

For each EP Statement, one or more *EP Language Statement Listeners* (EPL Statement Listener) can be configured. In case the EP Runtime produces a match for an event pattern or query, each EPL Statement Listener that is associated with the defining EPL Statement is notified about this match. The EPL Statement Listener is provided with a reference to the related event. It can either further analyze the event by examining certain event type attributes, or it can trigger an action in response to an event. Therefore, we classify EPL Statement Listeners as a part of the Execute MAPE-K phase.

The last EP Engine core component is the *EP Management Interface* (EP Management). It enables certain administrative tasks related to the EP Engine, such as the registration of EPL Statements within the EP Agent, or the instantiation of new EPL Statement Listeners.

The last two components that build the core of our Monitoring Runtime are *Event Sinks* and *Event Sources*. An Event Sink subscribes to the information represented by the event objects, and will leverage this information to perform further analysis, aggregation or simply alert an operator. *Event Sink Adapters* encapsulate these details to abstract from the implementation and protocol specifics of the supported Event Sinks. Similarly, *Event Source Adapters* provide integration for *External Event Sources*, which can push external event data into our monitoring system. This allows to represent data in EP statements that does not originate from the monitored service (composition). Hence, it enables a holistic view on monitoring data from various heterogeneous systems and effectively covers requirement **[M2]**.

On top of the Monitoring Runtime, our architecture blueprint proposes a *Management API* that provides access to information exposed by the Monitoring Runtime. This Monitoring API is delivered as a service, hence, enables platform and technology independent integration of our system into third party software. Such third party software includes business activity monitoring (BAM) systems or performance dashboards. Moreover, this approach effectively realizes Key Principle **[P4]**. The capabilities of this Management API range from simple queries for monitoring statistics, such as service response times, to administrative tasks related to the inner workings of the Monitoring Runtime, such as EPL Statement creation. Finally, a *Graphical User Interface* (GUI) visualizes the information provided by the Management API and facilitates the aforementioned administrative tasks.

5.3.2 Service Interface Surveillance Architecture

As illustrated in Figure 25, the proposed system architecture for the service interface surveillance method that we introduced previously shares many commonalities with the system architecture from the previous section. Therefore, we will only highlight components that have not been discussed above or play an additional role in this regard.

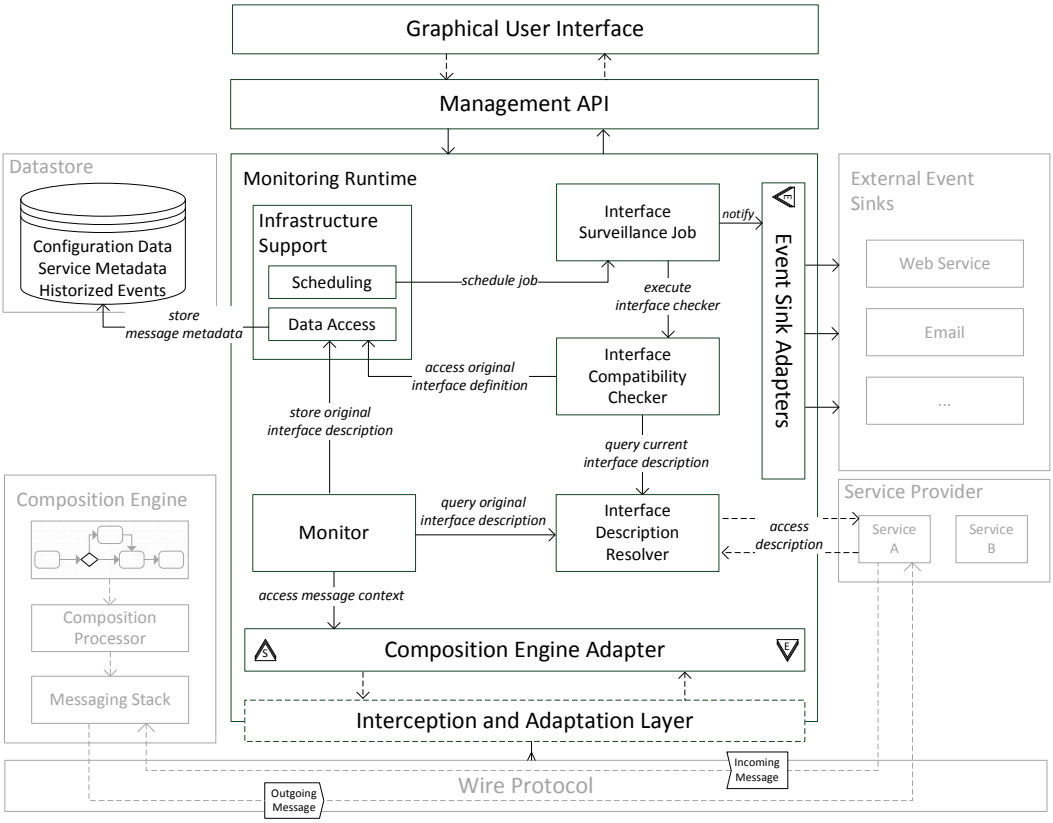


Figure 25: Architecture for a Service Interface Compatibility Checking System

Additional to emitting events when a service invocation is registered, the Monitor plays another important role within the interface surveillance architecture. As we stated earlier, the Monitor is also responsible for storing service metadata such as service endpoint addresses. When describing our interface checking method, we mentioned that we assume that service interface descriptions are publicly accessible via the interface description resolver (IDR) component. Given the endpoint address of a service, the IDR is capable of deriving the location where the corresponding interface description is available. Moreover, it abstracts the means required to access the interface description. The IDR effectively acts as a broker that provides transparent access to the interface description of a captured service. However, in cases where this is not possible, this location can be set manually. Ultimately, the Monitor component stores the original service interface description along with other service related metadata in the Datastore.

Along with the determination of the original interface description of a captured service by the Monitor component, our architecture provides an *Interface Surveillance Job* that *periodically* determines whether the original and the currently available interface description of a captured service differ and whether potential changes are backward compatible. To this end, our archi-

tecture features an *interface compatibility checker* (ICC). Simply put, the ICC implements the backward compatibility rules that we presented in the previous section. The ICC leverages the Data Access component to retrieve the original interface description, as well as the IDR to retrieve the most current version of the related interface description. The result of the compatibility check is reported back to the Interface Surveillance Job. Finally, if an incompatibility is detected, the Interface Surveillance Job uses the Event Sink Adapters to eventually send out notifications.

5.4 Summary

This chapter discussed the theory of operation and provided a corresponding system architecture for a novel monitoring system for composite services. Our method leverages a what we call holistic approach to the matter of composite service monitoring. The foundation of our method is an event based abstraction of the message exchange in composite services. Each message that is sent or received within a composite service is represented by a corresponding event. This enables us to model monitoring concerns in a generic fashion.

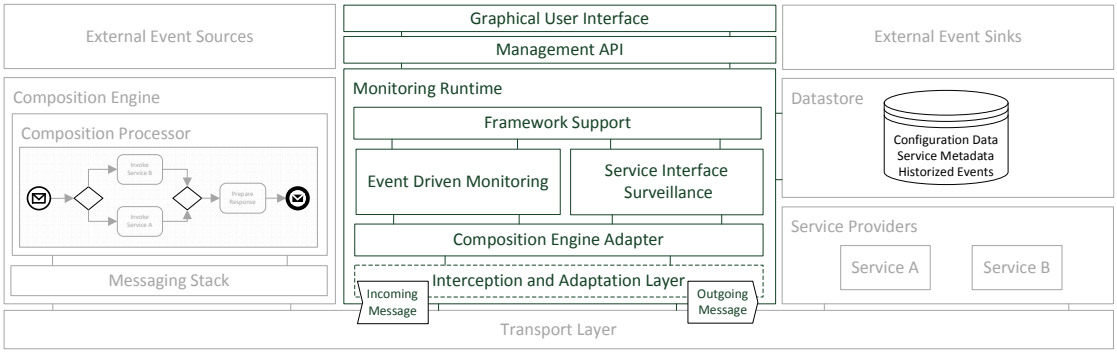


Figure 26: Overall Monitoring Architecture

The outstanding features of such an holistic monitoring system are (1) the capability to integrate monitoring data that is not produced by the composite service but originates from external systems, (2) covering monitoring requirements that span across several composite services, (3) a generic method to detect certain QoS or business related situations of interest, and (4) an integrated method to detect service interface compatibilities. Additionally, our approach enables the unobtrusive integration into existing composite service engines by the means of an interception and adaptation layer. This layer taps the message exchange within a composite service and provides access to the context of incoming and outgoing messages. The specifics of the composition engine are abstracted by so called composition runtime adapters, which enables an easy integration of additional composition engines.

The proposed architecture for such an holistic monitoring system integrates event stream processing technology to operate on the stream of events that represents the message exchange within a composite service. This technology enables us to model complex monitoring concerns that require the consideration of temporal and causal dependencies between messages. Event

source adapters abstract the details of external systems that provide information that is relevant for certain monitoring requirements. Event sink adapters, on the other hand, are used to abstract external systems that subscribe to certain situations of interest that were detected by our monitoring system. Finally, the service interface surveillance component complements the overall system architecture. A final top level view on the overall system architecture is provided in Figure 26.

CHAPTER 6

Domain Specific Adaptation in Composite Services

This chapter introduces an approach for *domain specific adaptation* in composite services. In the first section, we will illustrate the need for certain runtime adaptations of composite services. In particular, we will focus on the need for a *runtime service selection mechanism* that enables the dynamic exchange of the individual services within a composite service. Again, we will leverage our motivating example to frame the requirements for such a runtime service selection method.

The second section presents the theory of operation of a system that satisfies the requirements that were determined in the first section. We will discuss the concept of *selectors*, which are the components mainly responsible for choosing an appropriate alternative service for an individual service of a composition. To this end, several approaches for such selectors are compared to show their specific strengths and limitations. Thereafter, we will focus on so called *domain specific selectors* and dive into the details of a *domain specific service selection language*. Finally, we present the *transformer* concept. A transformer provides the means for *runtime service interface mismatch compensation* that may be required in case a service originally defined in a composite service differs from its alternative service in terms of their corresponding interface descriptions.

The third and final part of this chapter illustrates a system architecture that combines the findings of the previous sections. In this regard, this section serves as an *architectural blueprint* for the proof of concept implementation that is the subject of the next chapter. Finally, the chapter is conclude with a short wrap up of key points and final remarks.

6.1 Motivation

As we illustrated in Chapter 4, Quality of Service (QoS) is an important concept that allows us to quantify the state of a service with regard to its non-functional properties. To this end,

we presented an adaptive QoS model that enables us to formalize non-functional properties into specific QoS attributes. Moreover, we argued that the QoS of a composite service is directly influenced by the QoS of its individual services [85, 99]. As an example, the unavailability or unresponsiveness of a single individual service that is part of a service composition affects the availability and responsiveness of the composite service. Such performance and dependability problems can be identified by considering the QoS of each individual service of a composite service. A possible way to tackle such situations is the *replacement of poorly performing individual services* within a composite services. Of course, a replacement of individual services requires that there are appropriate alternative services available that do not suffer performance problems.

However, not only performance or dependability related QoS attributes are important in enterprise scenarios. When considering the SubscriberActivation scenario from our running example, we presented several business related QoS attributes, such as the CallSetupFee and the VoiceTrafficRate. In the case of Phonyfone, an important requirement in this regard is the minimization of the rates and fees that are due for the cooperation with their roaming partners. Hence, a method for replacing individual services within a service composition should also consider such *domain specific QoS attributes*.

Up to this point, we did not consider *when a service replacement takes place*. The most basic way to replace an individual service within a composite service is to alter the composite service definition. However, there are several major drawbacks to such an approach. First, a modification of the composite service definition implies a downtime of the service composition. This downtime is due to the fact that the composite service definition needs to be adapted, recompiled and redeployed. Second, such modifications require the skills of a system engineer to perform the changes. Another possible alternative is to define the service replacement logic in the composite service definition. Although composite service technologies may provide the means to dynamically assign the individual services during the execution of the composite service, there are some shortcomings to such an approach. On the one hand, the service replacement logic would be mixed into the business logic of the composite service, which clearly violates the principle of separation of concerns. On the other hand, the service selection criteria, that is, which service out of a set of alternative services, should replace the service originally defined in the composite service definition, is hard-wired into the composite service definition.

Additional to the aspect of when such service selections take place, another important issue is *how the corresponding service selection strategies are defined*. Such service selection strategies specify which QoS attributes should be considered by the selection mechanism to determine an appropriate alternative service. In this regard, we argued that the composite service definition does not represent an appropriate means to define such service selection strategies. Ideally, service selection strategies are managed separately from the composite service definition and can be adjusted during the execution of the composite service. Moreover, another noteworthy aspect in this regard is the link between the QoS model and the service selection strategies. That is, changes in the QoS model must be reflected accordingly in the service selection strategies to stay consistent in all participating components.

Finally, another important issue that needs consideration with regard to service selection in composite services is *service interface compatibility*. In our case study, we exemplified two roaming partners of Phonyfone, IPlus and Phonica. These roaming partners offer different rates

in terms of the CallSetupFee and the VoiceTrafficRate. Thus, these two attributes can be considered within a related service selection strategy that determines the most price effective roaming partner. However, we also stated that there is no guarantee that these roaming partners provide an interface that is syntactically equivalent to the interface that is referenced in the composite service definition. That means, although IPlus and Phonica are *semantically* equivalent, as they provide the same functionality to Phonyfone, they might be *syntactically* different in their service interface descriptions. Ideally, a comprehensive service selection approach requires that alternative services only have to be semantically equivalent, not necessarily syntactically equivalent.

6.1.1 Key Requirements

Hereinafter, we will present a *domain specific service selection* mechanism that addresses the issues that were discussed above. The key requirements for such a domain specific service selection mechanism can be summarized as follows:

- **[A1] Platform agnostic and unobtrusive**. Similar to the corresponding requirement with regard to the holistic monitoring system that we presented in the previous chapter, the system implementing the service selection mechanism must be technology agnostic and should be unobtrusive. That is, only a minimum of changes to the composition platform should be required to use the service selection mechanism.
- **[A2] Runtime adaptivity**. This requirement stems from the fact that design time modifications are not sufficient to cover the scenarios that were discussed above. Hence, the service selection mechanism shall determine an appropriate alternative service dynamically at runtime. Additionally, the tight integration with the adaptive QoS model that was presented in Chapter 4 requires that any changes to the QoS model should be immediately reflected and made available for the service selection mechanism.
- **[A3] Domain specific**. The specification of the service selection strategies must not rely on a general purpose language. Instead, a tailored domain specific language (DSL) shall be used for this purpose. Such a DSL should be intuitive and easy comprehensible to be usable by stakeholders that do not have programming skills. Yet, the feature set of such a domain specific service selection language shall allow to model service selection strategies that require advanced features, such as conditional selection.
- **[A4] Interface Mismatch Compensation**. The service selection mechanism shall require that alternative services are semantically equivalent to the individual services that should be replaced, but not necessarily syntactically equivalent. Hence, a related service selection mechanism shall provide the means to compensate the service interface mismatch that may exist between the the services originally defined in the composite service definition and their alternatives.

The following section describes the theory of operation of such a domain specific service selection mechanism.

6.2 Theory of Operation

Before we address the details of our domain specific service selection mechanism, we make the following explicit assumptions that our approach is based upon:

Assumption 3. *The interaction of services within a composite service is solely realized by the means of a message exchange.*

The first assumption is consistent with **Assumption 1** that we made for our monitoring approach from Chapter 5. This allows us to reuse certain building blocks that we originally introduced for our monitoring approach and enables the easy integration of the two orthogonal features.

Assumption 4. *Only a small number of alternative services (i.e., max. 1-5) is available for each replaceable service in a service composition.*

Additional to the relatively small number of alternative services that the second assumption states, autonomic discovery of alternative service is out of scope of this work. We assume that the import of alternative services is a manual process. This stems from the fact that in enterprise settings such as Phonyfone, there are usually certain obligations when contracting new service partners, which cannot be resolved automatically and need human intervention.

Assumption 5. *The unit of measurement for each QoS attribute is fixed and cannot be dynamically adapted.*

The last assumption is important for the interpretation of certain QoS attributes in the context of service selection strategies. As an example, consider the case where Phonyfone's roaming partner might have listed its CallSetupFee in EUR while another roaming partner might list the same attribute in USD. Such different manifestations of the same QoS attribute are currently not supported and out of scope of this thesis.

6.2.1 Runtime Service Selection with Selectors

This section presents the concept of *selector*s to dynamically replace the constituent services of a service composition by implementing a specific service selection strategy. Its primary task is to determine which service, from a set of available alternative services, matches the service selection strategy best. If none of the alternative services outperforms the originally defined service in terms of the service selection strategy, a selector returns the originally defined service.

The selection strategies reflected by a selector implementation can range from trivial randomized or single QoS attribute optimizing strategies, to complex strategies where several QoS attributes need to be included in the selection decision. When handling multiple QoS attributes in a selection strategy, it might further be desired to express conditional selection and weight the different QoS attributes according to their relative importance for the selection strategy. Besides the actual selection semantics, another important feature of such a selector component is *how* and *when* the selection strategy needs to be defined. In the following, we discuss several approaches for selector implementations. Table 6 summarizes the features of the selector implementation approaches.

Feature	Static Selector	Parameterizable Selector	Scripted Selector	Domain Specific Selector
Design-time Deployable	✓	✓	✓	✓
Runtime Deployable	✗	✗	✓	✓
Design-time QoS Weighting Adaptation	✓	✓	✓	✓
Runtime QoS Weighting Adaptation	✗	✓	✓	✓
Runtime Selection Strategy Adaptation	✗	✗	✓	✓
Implementation Agnostic Authoring	✗	✗	✗	✓
			✓ supported	✗ not supported

Table 6: Summary for Selector Implementation Approaches

Static Selectors

Static Selectors implement a hard-coded selection strategy and certainly represent the simplest solution. They have the inherent drawback that any change to the selection strategy implies a system downtime. A modify-recompile-redeploy cycle and a restart of the service composition environment are required.

Parameterizable Selectors

Parameterizable Selectors include parameters that can be adapted during runtime. As an example, consider a selection strategy where the average response time of a service should be minimized and the availability should be maximized. Assuming that the response time is more important than the availability, the QoS attributes need to be weighted. This weighting can be adjusted using weighting *parameters* in the selection strategy implementation. If these parameters are exposed, that is, through a user interface, the weighting can be adapted at runtime. This might be necessary when the requirements for the selection strategy change. Though, there are two major drawbacks of Parameterizable Selectors. First, implementation specific know-how is needed to adapt the underlying selection strategy. Second, if modifications of the selection strategy cannot be covered via parameter changes, a system downtime for the redeployment of the adapted selector implementation is inevitable.

Scripted Selectors

Scripted Selectors can be "hot-plugged" into the execution environment at runtime. The source code of the selection strategy implementation is interpreted during runtime rather than compiled at design-time. Thus, Scripted Selectors can be leveraged to provide true runtime adaptation of the service selection strategy. However, the implementation of the selection strategy does not abstract from programming language details. This makes Scripted Selectors practically unusable for domain experts that do not possess the required programming language skills.

Domain Specific Selectors

*Domain Specific Selector*s inherit the runtime deployability features of Scripted Selectors. However, instead of relying on a general purpose programming language, Domain Specific Selectors leverage a Domain Specific Language (DSL) to describe the underlying service selection semantics at the abstraction level of the problem domain. Additionally, it overcomes the drawbacks of the previously listed selector approaches (cf. Table 6) with regard to ease of strategy adaptability.

As we mentioned at the beginning of this chapter, we will focus on such domain specific selectors in this thesis. In the following section, we propose a DSL to implement the selector's underlying selection algorithm. The language is tailored to specifying service selection strategies and allows domain experts with no or little programming skills to use it.

6.2.2 A Domain Specific Service Selection Language

The *Vienna Domain Specific Service Selection Language* (VieDASSL) is a language that implements a domain specific selector approach as described in the previous section. In particular, VieDASSL addresses requirement **[A3]** to provide a flexible, yet reasonably manageable specification language for service selection strategies. It allows domain experts to create and adapt these selection strategies without programming language know-how. Additionally, it leverages the QoS model described in Chapter 4 to maximize flexibility with respect to QoS attributes.

Supported QoS Attributes

For each QoS attribute that we presented in Section 4.2 as part of our adaptive QoS model, VieDASSL provides a *QoSSymbol* that represents the QoS attribute in the language. Table 7 exemplifies the mapping of QoS attributes to QoSSymbols supported by VieDASSL and their related unit of measurement.

QoS Attribute	QoSSymbol	Unit
Response Time	`responseTimeMs`	milliseconds
Throughput	`throughputReqPerSec`	requests per second
Availability	`availabilityPercent`	percentage
Accuracy	`accuracyPercent`	percentage
CallSetupFee	`callSetupFee`	EUR
VoiceTrafficRate	`voiceTrafficRate`	EUR

Table 7: QoS Attribute to VieDASSL QoSSymbol Mapping

For brevity, only QoSSymbols relevant to the case study and for the examples hereinafter are shown in Table 7. However, it is important to note that the set of supported deterministic QoSSymbols can be arbitrarily extended during runtime, in compliance with requirement **[A2]**.

Furthermore, the QoSSymbols from Table 7 include a hint about their unit of measurement in their name. When adding new QoSSymbols to the set of deterministic QoSSymbols, their name should also include the related unit of measurement, both to improve traceability and make QoSSymbols self-explanatory.

The values of the QoS attributes in Table 7 are based on *all* available monitoring data for a particular service. In addition, our approach also provides *time-windowed* versions of these QoS attributes. By default, the 5, 10 and 15 minute averages are available. The QoSSymbols of these time-windowed attributes are equal to the QoSSymbols from Table 7, with the suffix *5Min*, *10min* or *15min* added at the end of the QoSSymbol. As an example, the response time over the last 10 minutes can be referred to by the QoSSymbol *responseTimeMs10Min*. Other examples for commonly used time-window based QoSSymbols are listed in Table 8.

QoSSymbol	Description
`responseTimeMs`	overall average response time
`responseTimeMs5Min`	average response time during the last five minutes
`responseTimeMs10Min`	average response time during the last 10 minutes
`responseTimeMs15Min`	average response time during the last 15 minutes

Table 8: Sample Time-Window based QoSSymbols

6.2.3 Language Description

VieDASSL supports three types of rules that represent the building blocks for defining service selection strategies. Please note that the service defined in a composite service definition, which is subject to replacement by alternative services, is hereinafter referred to as the *original service*.

Every selection strategy modeled with VieDASSL has at least one *FactorRule*. FactorRules are used to calculate a *score* for each of the services by considering (1) a QoS attribute, reflected in a rule by its corresponding QoSSymbol, and (2) a weight $W_j \in [0, 1]$ and $\sum_{j=1}^{k} W_j = 1$ for k QoS attributes used in the rules. The weighting factor expresses user preferences with regard to the considered QoS attributes. As with all other rules that we will discuss shortly, FactorRules have to be enclosed in a `selection{}` block. Listing 6 shows a VieDASSL selection specification having a single FactorRule.

```
1 selection {
2   max(availabilityPercent, 1)
3 }
```

Listing 6: Simple Factor Rule

A selector configured with the specification from above would try to maximize the availability QoS attribute with a weighting factor of `1`. Thus, it would pick the service that has the highest

```
1 <SelectionRule> ::= selection '{' <RuleBody> '}'
2 <RuleBody> ::= <FactorRule> | <SumSelectionRule> | <ConditionalSelectionRule>
3 <SumSelectionRule> ::= <FactorRule> { <FactorRule> }
4 <ConditionalSelectionRule> ::= 'inCase(' <QoSSymbol> ',' <Comparator> ','<QoSAttributeValue>')
5                                {' <RuleBody> '}
6                                {' <RuleBody> '}'
7 <FactorRule> ::= <MinimizingRule> | <MaximizingRule>
8 <MinimizingRule> ::= 'min(' <QoSSymbol> ',' <QoSAttributeWeighting> ')'
9 <MaximizingRule> ::= 'max(' <QoSSymbol> ',' <QoSAttributeWeighting> ')'
10 <Comparator> ::= 'less' | 'greater' | 'equals'
11 <QoSAttributeWeighting> ::= '0' | '1' | '0' '.' <Digit>
12 <QoSAttributeValue> ::= <Float>
13 <Float> ::= <Digit> '.' <Digit> {<Digit>}
14 <Digit> ::=  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
15 <Letter> ::= 'a' | ... | 'z' | 'A' | ... | 'Z'
16 <Character> ::= <Letter> | '_'
17 <QoSSymbol> ::= <Letter> {<Character> | <Digit>}
```

Listing 5: EBNF of VieDASSL

availability of all available alternative services. Of course, with only a single FactorRule, the weighting is of no relevance to the total score.

To enable the use of multiple QoS attributes in a VieDASSL definition, a *SumSelectionRule* can contain several FactorRules. Similar to [164], we use Simple Additive Weighting (SAW) [76] to determine the score of each alternative service. The algorithm is applied in two phases. First, a *scaling phase* transforms each considered QoS attribute value into a scaled attribute value $v \in [0, 1]$. This process allows for (1) *positive* and *negative* QoS attributes and (2) *uniform measurement*. For positive QoS attributes, higher values mean higher quality, e.g., service availability. Contrary, for negative QoS attributes, lower values mean higher quality, e.g., service response times. Uniform measurement enables the range and unit independent comparison of different QoS attributes, that is, to compare availability, measured as a probability, and response times, measured in milliseconds. Second, the *weighting phase* computes the overall quality score using the values provided by the scaling phase. Thus, the overall quality score for a service is determined by the formula $\sum_{j=1}^{k} sv_j * W_j$ where sv_j denotes the scaled value for a QoS attribute for k different QoS attributes. A detailed discussion of both the scaling and the weighting phase of SAW can be obtained from [164].

As an example, the specification shown in Listing 7 defines a VieDASSL selection strategy that focuses on the service availability QoS attribute, but additionally takes the average response time into consideration. The higher importance of the service availability over the average response time is expressed by the higher weighting value for the availability attribute.

```
1 selection {
2   max(availabilityPercent, 0.75)
3   min(responseTimeMs, 0.25)
4 }
```

Listing 7: SumSelectionRule

In simple scenarios, the SumSelectionRule might be well suited to model selection strategies that incorporate several QoS attributes. In scenarios where a threshold for a particular QoS attribute must be enforced, a *ConditionalSelectionRule*, as shown in Listing 8, is applicable. It consists of (1) a condition, (2) a positive and (3) a negative rule. The condition (line 2) includes a QoSSymbol, a comparator and a threshold value. If it evaluates to true, the score of the positive rule (lines 3-5) is returned, otherwise the score of the negative rule (lines 7-8) is returned. Moreover, conditional rules can be nested, which allows for modeling complex selection strategies. Listing 8 quotes a ConditionalSelectionRule that picks the most cost effective alternative service, provided that the average response time does not exceed 500 milliseconds. If the threshold is exceeded, we fall back to the SumSelectionRule described in Listing 7.

```
1 selection {
2   inCase (responseTimeMs, less, 500.0) {
3     min(callSetupFeeEur, 0.3)
4     min(voiceTrafficRateEur, 0.7)
5   } {
6     max(availabilityPercent, 0.75)
7     min(responseTimeMs, 0.25)
8   }
9 }
```

Listing 8: ConditionalSelectionRule

The complete language specification of VieDASSL is shown in Listing 5.

6.2.4 Selection Postprocessors

Under certain conditions, a Selector that naively applies the score based ranking mechanism described above can face one considerable problem. Choosing the top scored alternative service for each process invocation can overload the selected alternative service in very high load scenarios. This renders the runtime service selection useless or even makes the situation worse. To address this and similar issues, our Selector approach supports the concept of *Selection Postprocessors*. A Selection Postprocessor is provided with the map of alternative services and their related scores. It performs an additional selection among these top scored services. One approach for such a Selection Postprocessor would be a round-robin selection among the top ranked services. Another approach, the *weighted randomized selection*, is described next.

A prerequisite for a reasonable weighted randomized selection is that the score of the alternative services that are subject to the selection does not fall below a certain *relevance threshold*. This prevents bad performing services from being considered for the selection. To explain our approach for this problem, we assume an additional roaming service provider, *Roamwave*. The corresponding scores are listed in Table 9.

The table also features the related selection probabilities for each alternative service provider. For the example, the relevance threshold has been set to 0.20, meaning that only services that

Service Provider	Score	Selection Probability
Phonica	187.5	0.55
IPlus	151.3	0.45
Roamwave	98.7	0.0

Table 9: Service Scores for Weighted Randomized Selection

have a score above or equal to $minimumScore = score_{max} * (1 - rt)$ where $score_{max}$ is the highest score of all alternative services and rt is the relevance threshold are taken into account for the weighted random selection. As the score of the Roamwave partner service is below $minimumScore = 150$, it is not further considered and its selection probability is 0. The probability for a remaining service S' is calculated by $\frac{s_j}{\sum_{i=0}^{n} s_i}$ for n remaining alternative services and s_j being the score of S'.

Algorithm 1 Weighted Random Selection

```
1:  function WEIGHTEDRANDSELECTION(SCRS, RT)
2:      probabilities ← ∅                          ▷probabilites is a map structures
3:      totalScore ← 0
4:      random ← RANDOM(1)                         ▷random value between 0 and 1
5:      defaultService ← FINDMAXSCORESERVICE(SCRS)
6:      minimumScore ← max(SCRS.values) * (1 − RT)
7:      for all service_i ∈ SCRS do
8:          totalScore ← SCRS(service_i) + totalScore
9:      end for
10:     for all service_i ∈ SCRS do
11:         probability ← SCRS(service_i)/totalScore
12:         probabilites(service_i) ← probability
13:     end for
14:     for all service_i ∈ probabilities do
15:         probability ← probabilities(service_i)
16:         score_i ← SCRS(service_i)
17:         if score_i < minimumScore then
18:             continue
19:         end if
20:         random ← random − probability
21:         if random < 0 then
22:             return service_i
23:         end if
24:     end for
25:     return defaultService
26: end function
```

Algorithm 1 shows the simplified algorithm in pseudo-code. The algorithm is provided with a map structure that holds service references and their related scores ($SCRS$) as well as the

relevance threshold parameter (RT). Lines 2-5 initialize the required data structures. Line 6 calculates the minimum score for a service to be eligible for consideration based on the relevance threshold parameter RT. The first loop (lines 7-9) calculates the cumulative score of the available services. The second loop (lines 10-13) calculates the selection probability for each service and stores this information in the `probabilities` map. Finally, the third loop (lines 14-24) determines a candidate service. First, Lines 16-19 check if a service is eligible for selection using the relevance threshold as described above. Second, by evaluating if the difference between a random number $random$ (where $0 \leq random \leq 1$) and the service probability $probability$ (where $0 \leq probability \leq 1$) is positive (line 21), a service is finally chosen. By default, the algorithm returns the service with the highest score (Line 25).

So far, we implicitly assumed that an original service and its alternative services adhere to the same service interface. That is, their interface descriptions are syntactically equivalent. In turn, if a service offers the same functionality as an original service, but differs in its interface description from the original service, it could not be leveraged as an alternative service. In the following section, we will present an approach that relaxes this constraint.

6.2.5 Service Interface Mismatch Compensation with Transformers

In the context of our approach, a *Transformer* is a mediation component that compensates the interface mismatch between the original service and an alternative service. To this end, it applies transformation rules to incoming and outgoing messages. Thus, only *semantic equivalence* is necessary for a service to qualify itself as a replacement for another service. The concept of Transformers is visualized in Figure 27. For our service selection method, we propose two different types of Transformers:

Internal Transformer. As its name implies, an *Internal Transformer* is an integral part of the service selection system. Internal Transformers are provided with (1) an incoming or outgoing message and (2) transformation rules that should be applied to this message. Moreover, Internal Transformers have direct access to the building blocks of the system architecture that we will discuss shortly.

External Transformer. On the contrary, an *External Transformer* is not part of the service selection system, but is provided as a service by a third party. This means that the service selection system forwards incoming and outgoing messages to this third party service, which returns an adapted representation of the original message. In this case, our service selection runtime hands over the responsibility for the interface mismatch compensation to an external entity.

Both internal and external transformers have their strengths and limitations. On the one hand, in most cases, internal transformers might have a performance advantage over external transformers. This is due to the fact that with internal transformers, the required service interface mismatch compensation can be performed on-premises with the service selection system. In contrast, using external transformers entails interaction with a remote system, and thus, network latency that increases the overall processing time. Another important advantage of internal transformers over their external counterparts is the fact that the configuration of the transformer, that is, the interface mismatch compensation rules, reside within the service selection system. If adaptations to these rules are required, these adaptations can be done promptly. In the case of external transformers, it depends on the responsiveness of the third party how and when such adaptations are carried out.

The predominant case where external transformers may be applied is when the capabilities of internal transformers cannot cover the mediation requirements. Another area where external transformer can be applied beneficially is in scenarios where the transformation requirements are very demanding, especially in terms of performance. In such scenarios, using external transformers in favor over internal transformers can reduce the performance impact on the service selection system.

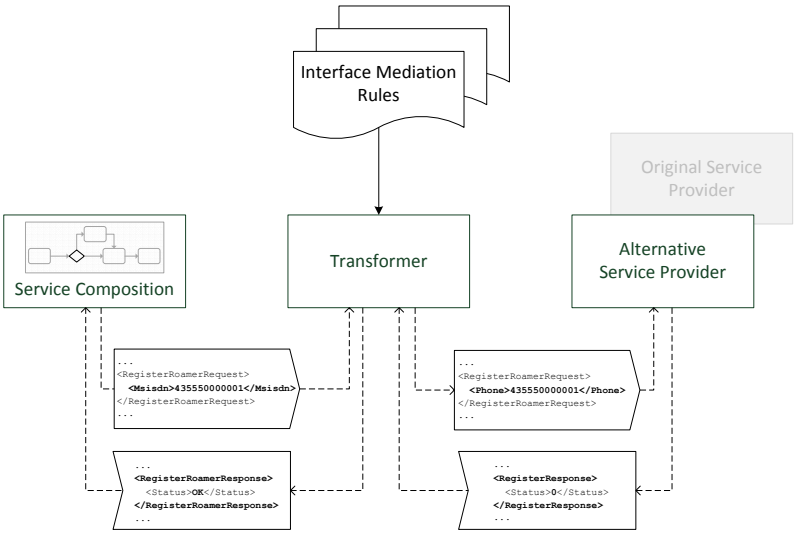


Figure 27: Conceptual View on the Transformer Component

We now have the necessary components to assemble a system that supports the requirements listed in Section 6.1.1. In the next section, we propose an architectural blueprint for such a system and go into detail about the interaction of the various components that we presented so far.

6.3 System Architecture

The architecture blueprint that we will now consider reuses several building blocks that we introduced when we discussed the architecture for our event driven monitoring system. Therefore, we will only briefly describe these components. Additional information can be obtained from Chapter 5. Moreover, and analog to Chapter 5, Figure 28 references certain MAPE-K phases. Such a reference is denominated by the corresponding symbols that we introduced in Chapter 1.

The two foundational components that are shared with the monitoring system architecture from Chapter 5 are the Interception and Adaptation Layer (IAL) as well as the Composition Engine Adapter (CEA). As aforementioned, these two components are used to tap the communication between a composite service and its individual services on the message level in the case of the IAL, and abstract from the specifics of arbitrary composition engine implementations in the case of the CEA.

The first key component of the service selection runtime architecture is the *Selector*. As previously described, the Selector component is responsible for determining an appropriate alternative service for an original service. The decision making is based on certain QoS attributes that serve as input for a selection strategy defined in VieDASSL. The values for these QoS attributes are either collected by the Monitor component, as we illustrated in Chapter 5, or predefined by the domain expert in case of deterministic attributes. This implies that alternatives for an original service need to be stored, together with the related QoS data. The storage for this data is provided by the means of a Datastore, which is described below. In combination with the Transformer component, the Selector covers the Plan and Execute parts of the MAPE loop. To this end, during the Planning phase the Selector evaluates a VieDASSL Rule, which is built by the Rule Builder in its Analyze phase. Furthermore, it executes the message adaptation by the means of the CEA. In this regard, the CEA serves as both a Sensor and Effector interface, as it enables the Selector to access (Sensor) as well as modify (Effector) the context of messages exchanged within a service composition. Please note that the Monitor component is not shown in Figure 28 to keep the figure concise.

In case the original service and the related alternative services do not adhere to the same interface, we argued that this interface mismatch needs to be considered and compensated accordingly. As aforementioned, the *Transformer* component is required to apply interface mediation to enforce compliance with the service interface as defined in the process. A *Transformation Adapter* brokers the transformation requests towards the configured *Transformation Engine*. In this regard, the Transformation Adapter abstracts from the internals of the internal *Transformation Engine* and the *External Transformation Engine*, which may be applied in certain conditions that we stressed above. The related *Transformation Rules*, which specify the details of such an interface mismatch compensation, are stored in the Datastore.

The Selector and Transformer components leverage the *Infrastructure Support* that we already introduced in Chapter 5, which provides *Data Access* to a *Datastore*. This Datastore holds important information and metadata such as the QoS information associated with a partner service, the available alternative services for a particular service, the service selection strategy definitions as well as the transformation rules for interface mediation. Again, the QoS data is either autonomously collected by the Monitor component, or defined by a domain expert. Other data,

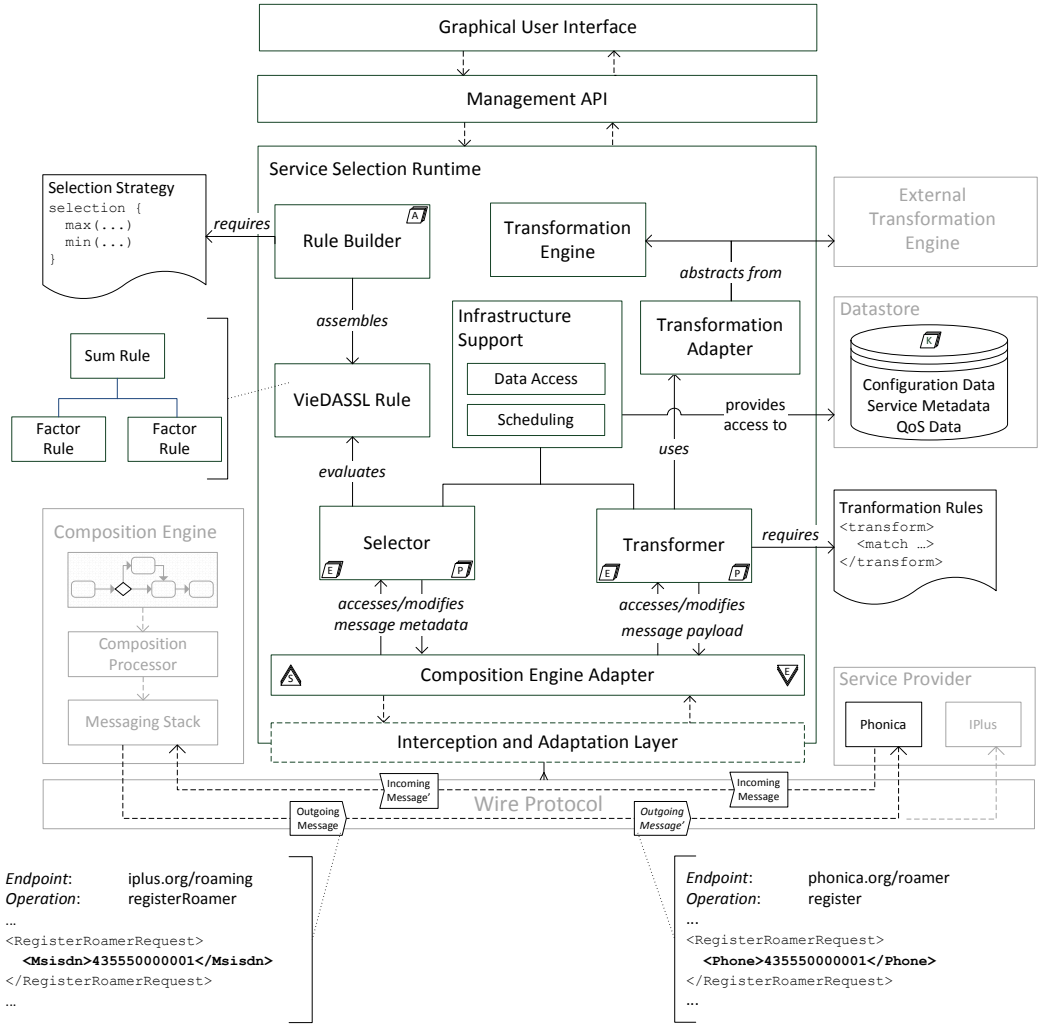


Figure 28: Architectural Blueprint for a Domain-Specific Service Selection Runtime

including VieDASSL and transformation rules, need to be defined manually. In terms of the autonomic computing context that we established for our components, the Datastore provides access to the Knowledge associated with the presented Service Selection Runtime.

When considering the service selection rules defined in VieDASSL, the resulting *VieDASSL Rule* is truly a tree of rules, with its root rule assigned to the Selector component. The Selector invokes this top level rule for each alternative service, resulting in the root rule recursively invoking all its child rules to evaluate the score of each relevant service. The (re)creation of such a VieDASSL Rule lies in the responsibility of the *Rule Builder*, which parses and verifies selection strategy definitions. Dynamic language support offered by current programming languages enables the runtime adaptivity of both the service selection language as well as the QoS model.

The selection strategy definitions as well as the transformation rules are passed in by the *Management API*. This Management API provides access to important information and certain operations that control the inner workings of the Service Selection Runtime. Moreover, it decouples the Service Selection Runtime from the *Graphical User Interface* (GUI). The GUI enables the user to perform service management tasks, such as defining the service selection strategies, preparing services for runtime replacement by marking them replaceable, importing alternative services, displaying and comparing QoS data. The GUI provides an authoring system that supports the domain expert with syntax highlighting and statement auto-completion. Finally, a QoS model browser that is part of the GUI displays available QoSSymbols and supports the domain expert in creating additional QoSSymbols as needed.

6.3.1 Conceptual Approach

In what follows we provide a description of the conceptual approach that corresponds to the system architecture that was discussed above. In this regard, this section should illustrate the end user view of a service selection system that was implemented based upon the presented architecture blueprint. We will further emphasize on this aspect in Chapter 8.

Applying our domain specific service selection runtime to composite services such as the ManagedRoaming composition from our SubscriberActivation scenario enables users (1) to monitor the QoS of the composite service and its individual services and (2) to define a selection strategy for the roaming partner service. The domain expert does not have to deal with a complex setup to access QoS data of the services involved in process execution. Instead, she can focus on the analysis of this information. To leverage the service selection features for the Roaming partner service, the domain expert selects the related service from the list of services captured by the Monitor and marks it replaceable. Only services that are marked replaceable are considered to be replaced by alternative services. What follows is the import and assignment of additional services, such as the *Phonica* service. Then, a selection strategy needs to be defined. The strategy shown in Listing 8 provides a good starting point, as it tries to maximize performance and dependability as well as takes cost efficiency into account. Clearly, the considered domain specific attributes need to be added to the QoS model by defining them in the User Interface. This is done by creating a QoSSymbol for the attribute (e.g., CallSetupFee, cf. QoSSymbol in Listing 5) and defining a value range. The new QoS attribute is immediately available in the service selection runtime. Its values for relevant service operations, e.g., `registerRoamer`, can be set via the Management API, and the QoSSymbol can be used in service selection strategies. Finally, transformation rules need to be specified since the interface of the originally used roaming partner service (IPlus) does not match the Phonica service interface.

6.4 Summary

In this chapter, we presented an approach to dynamically adapt composite services at runtime. Our method relies on runtime service selection and exchange of individual services within a service composition. To this end, we introduced the concept of selectors, which represent the core component for our mechanism. These selectors are provided with a pool of alternative

services for a service that was originally defined in a composite service definition, as well as selection strategies that define how such a service selection should be conducted. In this regard, our selection strategy definitions are specified using a domain specific service selection language that is tailored for this task. Using this DSL, domain experts can define selection strategies that encompass both traditional QoS as well as business related QoS aspects. This DSL is tightly integrated with the adaptive QoS model that we presented in Chapter 4, and thus, provides a tool to model service selection strategies that are up to date and consistent with the underlying QoS model. Moreover, we discussed the transformer component, which provides the means to compensate a possible service interface mismatch between an original service and its alternative services. Thus, the transformer component enables us to relax the constraints for alternative services in the sense that only semantical equivalence between original and alternative services is required, not necessarily syntactical equivalence. Finally, we presented an architectural blueprint for an implementation of such a service selection runtime. This blueprint will be leverage in the next chapter, when we present details on the proof of concept implementation of our monitoring and service selection runtime.

CHAPTER 7

Runtime Support for Domain Specific Monitoring and Adaptation

The previous chapters mainly focused on the theoretical foundations of this thesis. That is, we covered the theory of operation of the building blocks of our monitoring and adaptation approaches for composite services, as well as presented an architectural blueprint for such a system. On the contrary, this chapter will use this architectural blueprint to illustrate our *proof of concept implementation* of this architecture for the Java platform, called *VieDAME* (Vienna Dynamic Adaptation and Monitoring Environment). To this end, we will briefly discuss the key technologies that were used for the implementation. We will present *aspect oriented programming* (AOP) as the key technology that enables us to extend an existing composition engine while staying in line with Key Principle **[P1]** (unobtrusiveness). Additionally, we discuss the key features of *Spring Framework*, which is used in our implementation to promote loose component coupling by leveraging dependency injection.

The second section provides an *implementation specific big picture* of the overall system architecture and shows where and how the key technologies from the first section were leveraged. Due to length restrictions, we cannot cover all implementation specific details in depth. However, we will exemplify the implementation of monitoring requirements using a custom configuration language based on XML. As usual, the chapter is concluded with a wrap up of key points as well as some final remarks.

7.1 Technology Review

In what follows, we briefly present two cornerstones of the proof of concept implementation that is subject of this chapter. First, we will explain the key ideas behind aspect oriented programming (AOP). Additionally, we give some examples using the AOP toolkit that was leveraged for VieDAME, which was AspectJ [49] (in combination with Spring AOP).

Second, we give a high level overview of Spring Framework [148], which is one of the most popular application development framework for the Java platform. Some short code examples are given to illustrate how Spring Framework was leveraged for component configuration and wiring within VieDAME.

7.1.1 Aspect Oriented Programming

In general, AOP is seen as a technology that is used to manage so called *cross cutting concerns*. While object oriented programming (OOP) is often listed as the most commonly applied technology to manage core concerns [91], that is, concerns that cover the functional requirements of a software component, AOP is aimed to cover concerns that are common to many of the components that cover core concerns. Typical examples for such cross cutting concerns are logging, transaction handling and security, as they *cut across* the core modules. AOP uses a particular terminology, which we will cover next.

An *aspect* in AOP can be seen as a concern that is related to one or more places in existing code. In AOP, those places are called *joinpoints*. Instructing the AOP framework where exactly it should add the additional functionality, which is called an *advice* in AOP jargon, requires the definition of *pointcuts*. Pointcuts are expressions that identify arbitrary events in the runtime system such as method invocations or field access. Advices can be execute before, after or around a pointcut, depending on the particular requirements. An example for an around advice is provided in Listing 9 using AspectJ's pointcut syntax.

```
1 @Around("execution(* org.apache.axis2.description.OutInAxisOperationClient.send(..))")
2 public Object interceptAxisSend(ProceedingJoinPoint joinPoint) throws Throwable {
3   long start = System.currentTimeMillis();
4   MessageContext responseMessageContext = (MessageContext) joinPoint.proceed();
5   long end = System.currentTimeMillis();
6   serviceInvocationMonitor.stop();
7   long executionTimeMillis = end - start;
8
9   ServiceInvocationEvent event = new ServiceInvocationEvent(
10        executionTimeMillis, true, start);
11
12  event.setRequestPayload(getRequestPayload(messageContext));
13  ...
14  registerServiceInvocation(event);
15  ...
16  return messageContext;
17 }
18
```

Listing 9: Intercepting Message Sending with AOP

The pointcut definition on Line 1 is interpreted as follows. The term `execution` indicates that the pointcut is related to a method execution. The values inside the parentheses refer to the method's return type (which is of no concern in this example, as indicated by the `*`), the full qualified method name (`org.apache.axis2.description.OutInAxisOperationClient.send`) as well as the method arguments (which are also of no concern, indicated by `(..)`). Hence, the method `interceptAxisSend()` from Listing 9 is invoked before the method declared in the pointcut expression is invoked. Truly, the `send()` method would not be invoked at all if `interceptAxisSend()` would not invoke it in Line 5 by executing the `proceed()` method of the `joinPoint`, which is passed in as the only parameter of method `interceptAxisSend()`. The `proceed()` method effectively invokes the `send()` method unaltered, that is, no modifications to the method parameters are performed. The expected return value of the method call is then returned by the advice code on Line 17.

The final responsibility of the AOP framework is the combination of the base system and the additional aspect code. This step is called *code weaving* and affects the performance and deployment of the altered code. Whereas compile-time weaving requires an intermediate step to generate the modified classes, *load-time weaving* adds the advice code during class loading time. As an additional benefit, aspects can be deployed and undeployed during runtime. What is noteworthy here is that classes are unaware of aspects that are applied to them. For example, this means that the particular module is not aware that its execution is being logged by a monitoring aspect that is woven into it, resulting in independently emerging implementations.

7.1.2 Component Configuration and Wiring with Spring Framework

The project website states that *Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform* [148]. As such, Spring includes support for AOP, declarative transaction management, unified database access as well as a framework to build Web applications. However, the foremost reason why we opted to introduce Spring Framework (Spring for short) into VieDAME is its support for *dependency injection* [98].

Traditionally, if an application consists of several components that interact with each other, the interacting component have to resolve their dependency by themselves. Truly, this means that component dependencies have to be resolved programmatically by the application developer. Using the principle of dependency injection, resolving the aforementioned dependencies is performed by an external entity – a dependency injection framework like Spring – at object creation time. This external entity coordinates creation and management of each object in the system. Thus, application developers only have to define which components collaborate. Additionally, dependency injection promotes loose coupling as it removes any internal wiring between components. Declaring such component dependencies is usually realized via XML files or by the means of Java annotations. Listing 10 exemplifies the use of XML to declaratively configure email support for sending VieDAME email notifications.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4        xsi:schemaLocation="http://www.springframework.org/schema/beans
5         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
6
7   <bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
8       <property name="host" value="mgate.chello.at"/>
9   </bean>
10
11  <bean id="templateMessage" class="org.springframework.mail.SimpleMailMessage">
12      <property name="from" value="viedame@infosys.tuwien.ac.at"/>
13      <property name="subject" value="-- VieDAME Monitoring Alert --"/>
14  </bean>
15
16  <bean id="emailNotificationSender"
17        class="at.ac.tuwien.infosys.viedame.core.services.notification.EmailAdapter">
18      <property name="receivers">
19          <list>
20              <value>moser@infosys.tuwien.ac.at</value>
21          </list>
22      </property>
23      <property name="mailSender" ref="mailSender"/>
24      <property name="templateMessage" ref="templateMessage"/>
25      <property name="mailBodyTemplate"
26                value="this is a generated VieDAME error notification."/>
27  </bean>
28 </beans>
```

Listing 10: Spring Configuration for Email Sending in VieDAME

In Spring, plain old Java objects (POJO) that are configured via XML or the use of annotations, are usually referred to as Spring *beans*. Considering Listing 10, Lines 7-9 declare such a bean. Its type is declared via the `class` attribute, while an unique identifier is assigned using the `id` attribute. A bean property is set using the `<property>` tag within a bean definition as on Line 8. Such property definitions can have simple values, such as the email host on Line 8, but can also be backed by an implementation of the Java Collection interface. An example for such a collection can be seen on Lines 19-21, where we use `<list>` and `<value>` tags to set the elements of the underlying list. Finally, we can reference other Spring beans by using the `ref` attribute of the `<property>` tag, as shown on Line 23.

The example from above shows how Spring promotes loose coupling by enabling a declarative definition of component dependencies. To understand the benefits of this declarative component wiring approach, again consider the `emailNotificationSender` bean. This bean references two other beans, the `mailSender` and the `templateMessage` beans. Resolving these dependencies programatically, that is, using Java code, would mean that the `emailNotificationSender` itself is responsible of resolving its dependencies. Dependency injection frameworks, such as Spring, on the other hand, use *inversion of control* [98] to reverse the responsibilities with regard to resolving dependencies. Hence, in our example, switching to another `mailSender` implementation requires only to modify the reference in the `emailNotificationSender` bean.

As Listing 10 shows, bean definitions are rather generic. For certain configurations such as the email sender from above, this is not of an issue. However, in other cases, such an approach might be too verbose. For such situations, Spring offers a mechanism for schema-based extensions to the generic Spring XML format. Such a specialized configuration format has the advantage that it is fully compatible with all Spring features and allows to build domain specific configuration containers. In VieDAME, we leverage these capabilities for the configuration of monitoring related components. We will illustrate the use of such custom schema-based extension in Section 7.2.

7.2 Implementation Details

In the following, we will briefly explain which implementation technologies have been leveraged to build the proof of concept implementation of the monitoring and adaptation architecture that was presented in the previous chapters. In general, we only used free and open-source software (FOSS) [115] for the implementation of the prototype. Figure 29 provides a visual overview of these technologies and tools.

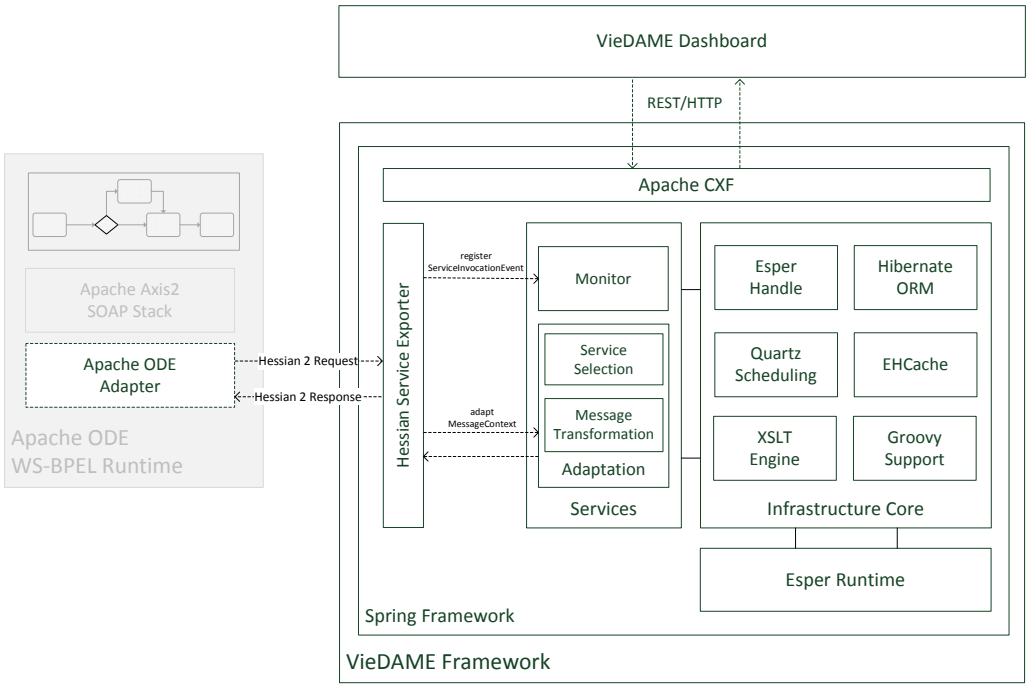


Figure 29: Implementation Technologies

We decided to support WS-BPEL based service compositions for our proof of concept implementation, since WS-BPEL represents the de-facto standard for compositions that leverage Web service technology as the underlying service implementation technology. In this regard, VieDAME currently supports two WS-BPEL based composition engines, Apache ODE [12] and

ActiveBPEL [4]. Both composition engines rely on *Apache Axis* as their SOAP stack provider. However, Apache ODE uses Axis 2, whereas the version of ActiveBPEL that is currently supported by VieDAME still uses Axis 1.4. As we explained previously, our approach taps the message exchange of composite services using composition engine adapters. In Figure 29, we exemplified the *Apache ODE Adapter*.

Exposing the monitoring and adaptation capabilities of VieDAME is realized by the means of a *Hessian Service Exporter*. This enables us to transparently expose the monitor, selector and transformer interfaces. The decision to use a binary protocol such as Hessian 2 is mainly based on the superior performance [68] over text based protocols such as SOAP. There are a many implementations of the Hessian 2 protocol available [37], which enables a platform agnostic integration of VieDAME into third party software. The REST based management API was implemented using Apache CXF [9]

The infrastructure core of VieDAME is comprised of several tool kits and components that are required for common functionality such as logging and scheduling. In this regard, we use *Quartz* [116] for scheduling tasks inside the VieDAME framework, such as the service interface surveillance job that we described in Chapter 5. The Object/Relational mapping framework *Hibernate* [135] was used to map the service model to a relational database. Furthermore, we used *EHCache* to equip various aspects of the framework with caching capabilities to improve performance. The event processing platform Esper [57] was used to integrate event stream processing technology into our framework. Finally, the dynamic language features of *Groovy* [43] were used to implement VieDASSL. In this regard, we leverage the ExpandoMetaClass features [42] as well as the Builder Pattern [65] to achieve the required runtime adaptivity. Finally, *Spring Framework* is used to manage the dependencies between the various components.

7.2.1 Configuration of Monitoring Requirements

Spring provides a mechanism to extend the basic Spring XML format to define beans in a custom manner. This requires to (1) author the XML schema definition that describes the custom configuration format, (2) code a `NamespaceHandler` as well as a `BeanDefinitionParser` and (3) register these artifacts within the Spring runtime. Due to space restrictions, we cannot go into details about these steps. The reference documentation provides an in-depth discussion of this matter [142]. However, we show a sample configuration file below. This configuration snippet is parsed and handled according to the steps listed above.

As mentioned in Chapter 2, we are using the `org.apache.axiom.om.OMDocument` event type provided by Esper to represent the payload of SOAP messages that are exchanged within a service composition. This enables us to define event queries and patterns that access the message payload, such as the examples that we provided in Chapter 5. Using this XML event type to access particular tags within a SOAP message requires that the Esper runtime is aware of the schema of the message. This allows to automatically derive the event properties, such as the `msisdn` property that reflects the related MSISDN of a subscriber in the AddSubscriber request. If no schema is provided, we can still access such event properties. However, in this case, the explicit configuration of event properties using a corresponding XPath expression is required. The sample configuration listed below uses such an explicit event property declaration to exemplify the modus operandi.

```
1 <viedame:monitor id="subscriberActivationExecutionTimeViolation"
2                  eventTypeAlias="AddSubscriber">
3
4   <viedame:soapPayloadConfig>
5
6       <viedame:namespaces>
7           <viedame:namespace prefix="com" namespace="http://com.phonyfone.services" />
8       </viedame:namespaces>
9       <viedame:xPathProperties>
10          <viedame:xPathProperty
11             expression="$SOAP_BODY$/com:addSubscriber/com:subscriber/com:msisdn"
12             name="msisdn"
13             type="STRING"/>
14
15      </viedame:xPathProperties>
16  </viedame:soapPayloadConfig>
17  <viedame:statement name="addSubViolationStmnt">
18    every addSubViolationAlert =
19      AddSubscriber -> (timer:interval(10 sec)
20        and not
21      ManagedRoamingEvent = ManagedRoaming(
22        msisdn = AddSubscriber.msisdn
23      )
24  </viedame:statement>
25  <viedame:listeners>
26      <viedame:listener ref="loggingXmlDomEventTypeListener"/>
27  </viedame:listeners>
28 </viedame:monitor>
29
```

Listing 11: Custom Spring Bean for Declaring Monitoring Requirements in VieDAME

When considering Listing 11, the custom configuration directives are configured within the `viedame` namespace (as required by [142]). A new VieDAME monitoring rule is declared on Lines 1-2 using the `viedame:monitor` tag. The `id` attribute is a unique identifier for this monitoring rule, whereas the `eventTypeAlias` can be used to reference the corresponding event within the Esper runtime. The `soapPayloadConfig` element is used to specify namespaces and XPath properties. By default, the `soapenv` prefix is linked with the `http://schemas.xmlsoap.org/soap/envelope/` namespace, which is common to most SOAP messages. Additional namespaces require specification inside the `namespaces` tag (Line 6), which contains `namespace` definitions (Line 7). Such a namespace definition requires a namespace prefix as well as the actual namespace, which are provided by the `prefix` and `namespace` attributes. As we explained above, in case no XSD is available, XML based event properties have to be declared explicitly using a XPath expression. On Line 10, Listing 11 exemplifies such a event property definition. Each property requires a related `xPathProperty`, where the `expression` attribute defines the XPath expression. A `name` is used to alias the event property in EPL queries and patterns, as seen on Line 22. The `type` attribute tells the Esper runtime the type of the property. This is relevant when applying arithmetical expressions on an event property, which is only valid for numbers. The actual EPL query or pattern is specified using the `statement` element (Line 17). Finally, a set of `listeners` (Line 25) should be specified to make use of the matching event.

7.3 Summary

This chapter briefly discussed some implementation specific details of VieDAME, which represents the proof of concept implementation of the monitoring and adaptation approach that we presented in the previous chapters. We provided a big picture that referenced the implementation technologies and tools that we used, as well as illustrated a specific implementation detail of VieDAME by exemplifying a monitoring rule definition. In the following chapter, VieDAME will undergo a performance and qualitative validation that should highlight the strengths and limitations of our system.

CHAPTER 8

Evaluation

This chapter provides an evaluation of the proof of concept implementation that we discussed in Chapter 7. A WS-BPEL based implementation of the SubscriberActivation case study from Chapter 3 provides the basis for the performance validation.

In the first section of this chapter, we introduce the *experiment setup*. That is, we list software and hardware components that were required to conduct the experiments that the evaluation is based upon. Furthermore, the setup of the evaluation environment is discussed, including details on the implementation of the case study.

The second section presents the results of several *system benchmarks*. These benchmarks are primarily targeted at system performance. The experiments were conducted using a *multi-stage scenario*, where each scenario shows the performance penalty introduced by a specific feature of our VieDAME runtime. These benchmarks also provide information on the performance impact that our approach has on several important system parameters, such as CPU usage and memory consumption. We also show an *end-to-end benchmark* that highlights the performance gains which can be achieved in composite services using our VieDAME system.

In the third and final section of this chapter, we provide an in-depth *qualitative discussion* of our contributions. A stakeholder analysis is presented, as well as the strengths and known limitations of our system. A short summary concludes the chapter.

8.1 Experiment Setup

The following section presents the hardware and software components that were required to conduct the experiments for the performance evaluation.

8.1.1 Hardware Setup

The hardware setup that was chosen for the performance evaluation experiments consisted of three different hosts: machine (1) acted as the host for the WS-BPEL engine, machine (2) hosted the individual services that took part in the AddSubscriber and ManagedRoaming compositions,

and finally, machine (3) hosted our VieDAME system. The reasoning behind the decision to use three different host for the individual components was that we wanted to achieve a high level of isolation for the different components as each component has a distinctive impact on system performance. The hosts were interconnected with gigabit ethernet to ensure that network performance would not constrain the overall system performance and thus affect the benchmark results. Details on the used equipment are summarized in Table 10.

Machine	OS	CPU	Memory	Disk
(1) WS-BPEL Engine Host	Linux 3.0.0	4 x 2.6GHz	8GB	SSD
(2) Web Service Host	Linux 2.6.36	2 x 2.6GHz	4GB	HD
(3) VieDAME Host	Linux 3.0.0	8 x 3.4GHz	16GB	SSD

HD conventional hard disk drive – *SSD* solid state disk

Table 10: Summary of Hardware Equipment for Performance Evaluation

8.1.2 Software Setup

As shown in Table 10, all machines that participated in the performance evaluation were running a version of Linux. In general, all software that was part of the performance evaluation is provided either as OpenSource [115] or is at least freely available. Similar to our statement from Chapter 7, we believe that is important to use an evaluation environment that is reproducible for interested readers.

Composition Engine. Previously, we stated that our proof of concept implementation currently supports ActiveBPEL and Apache ODE as WS-BPEL target platforms. We decided to use Apache ODE for the performance evaluation. This is due to the fact that Apache ODE is widely used in the community as well as provides the foundation for many research, OpenSource and commercial business process management systems [11, 10, 136, 59, 82]. Furthermore, we used a visual WS-BPEL designer provided by Intalio [81] to create a WS-BPEL based implementation of the service compositions that are part of our SubscriberActivation scenario from Chapter 3. The related implementation diagrams are provided in the Appendix for reference. For the deployment of the Apache ODE engine we used Apache Tomcat [14].

Individual Services. Since we used a WS-BPEL based implementation of the case study, Web services were the implementation technology of choice for the individual services. The Web service framework Apache CXF [9] was used to create JAX-WS compliant implementations for the individual services that are orchestrated by the AddSubscriber and ManagedRoaming composite services respectively. Each Web service supports an additional `setQos()` operation. This operation allows to dynamically influence certain QoS attributes, such as the service response time or its accuracy. The `setQoS()` operation controls these attributes by adding a processing delay or error probability. Together with a random factor, we were able to simulate service quality in a basic way. This `setQoS()` method plays an important role in the end-to-end evaluation scenario, since it is leveraged to simulate a service outage.

Load Testing Related Components. The load testing tool LoadUI [141] was used to control the load test as well as generate the required load. That is, it generates SOAP requests and concurrently issues these requests. LoadUI is an OpenSource tool that provides a free and platform agnostic solution to manage and perform load tests. It is tightly integrated with SoapUI [61], a generic testing tool for Web based applications. The capabilities provided by LoadUI, together with the metrics provided by JConsole [117], were used to capture several system performance indicators, such as CPU usage or memory consumption.

VieDAME Setup. Our VieDAME setup was deployed on a dedicated host to illustrate how the service based approach that we have taken influences system performance. The communication between the VieDAME engine-adapter component and the VieDAME core component is realized by the means of the Hessian 2 protocol, as described in Chapter 7. Hence, and important to note when considering the results below, each monitoring or adaptation request makes use of a full blown TCP/IP connection.

Various Components. PostgreSQL [133] was the RDBMS of choice for the datastore that is required by our VieDAME system to store configuration, service meta-data as well as historized events.

Component	Version Information	Description
Oracle Java	1.7.0	Java JDK/JRE
Apache Tomcat	7.0.26	Servlet 3.0 and JSP 2.2 container
Apache ODE	1.3.5	WS-BPEL 2.0 complaint composition engine
Apache CXF	2.4.6	JAX-WS (TCK compliant) Web service stack
LoadUI	2.0 Beta	JavaFX based load testing UI
SoapUI	4.0.1	Java based functional testing tool
PostgreSQL	9.1	PostgreSQL RDBMS

Table 11: Summary of Software Component Versions

For a quick reference, Table 11 provides version details of the software components that were used in the system performance evaluation.

8.2 System Performance

In this section, we consider the system performance of our VieDAME system. To this end, we use, at first, a *multi-stage* evaluation scenario, where a VieDAME enhanced Apache ODE WS-BPEL engine executes the ManagedRoaming composite service. This multi-stage evaluation shows the performance penalty introduced by each VieDAME feature, that is, the monitoring and adaptation capabilities, in comparison with a default Apache ODE engine. After this multi-stage comparison of system performance, we present an end-to-end evaluation scenario where we will highlight the performance gains that can be achieved from our VieDAME system. A

service outage will be simulated to show how the VieDAME system responds to such a situation and how performance is brought back to where it was before the outage.

8.2.1 Multi-Stage Performance Evaluation

As noted above, we stressed the system performance using several stages, with each stage adding a specific VieDAME feature to the previous stage. In stage (1), a default Apache ODE instance was stressed, with the VieDAME extension completely disabled. In stage (2), VieDAME was stressed in a monitoring-only configuration to highlight the performance penalty introduced by the monitoring capabilities. The monitoring-only configuration was created by disabling the selection and transformation aspects. Finally, in stage (3), both the monitoring and service selection features were stressed. In terms of service selection, the VieDASSL Selector shown in Listing 12 was setup. This Selector is configured to pick the service that performed best during the last hour in terms of its response time, as well as considers the overall availability of the service alternatives. The `QuerySubscriber` service was cloned to create five alternatives. The original `QuerySubscriber` service and three of its alternatives were preloaded with worse QoS attribute values than the remaining alternative `QuerySubscriber` instance. For this particular `QuerySubscriber` alternative, a Transformer was configured since its interface was slightly different from the interface description of the original `QuerySubscriber` service. Taken together, this means that the full set of VieDAME features, including monitoring, runtime service selection and interface transformation, was stressed in stage (3).

```
1 selection {
2         min(responseTimeMs60Min, 0.49)
3         max(availabilityPercent, 0.51)
4 }
```

Listing 12: VieDASSL Selector for Multi-Stage Performance Evaluation

With regard to the number of concurrent users, which we will refer to as *VUsers* hereafter, we chose 50 and 100 VUsers as we believe this represents a real world workload for a single server. We used a ramp-up time of 2 seconds per VUser, that is, not all VUsers were started in parallel, but sequentially, with 2 seconds in between the start of two consecutive VUsers. Additionally, the LoadUI parameters test-delay and random were set to 1000 milliseconds for the delay, and 0.5 for the randomness. This means that between two consecutive requests, at most 1000 milliseconds delay was used. The QoS of the individual services remained unchanged for this part of the performance evaluation, since random processing delays or faults of the individual services would prevent us from clearly distill the performance impact that is cause by the VieDAME runtime.

Moreover, we collected various system performance metrics during the loadtest. For brevity, we only show the *average response time* and *transactions per second* as indicators for system performance, as well as *CPU usage* and *memory usage* as indicators for system health, in the figures below. The detailed results of these performance experiments are presented in the following.

Results and Discussion

The results for the multi-stage performance evaluation are shown in Figure 30 and Figure 31 These figures show results for the 50 VUsers and 100 VUsers load tests, respectively.

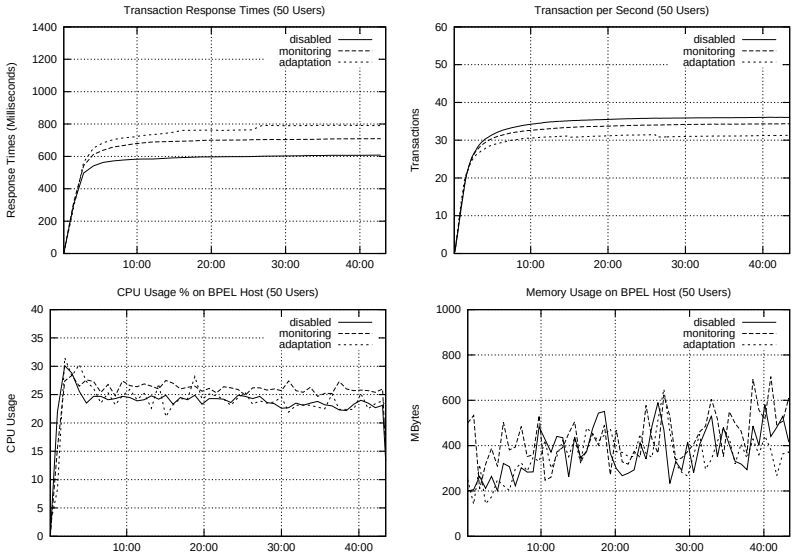


Figure 30: Results of 50 VUsers Loadtest

Regarding the CPU and memory usage, we could not observe a particular increase in memory usage or CPU load. This is mainly due to the fact that the processing is truly performed on the machine that hosts the VieDAME runtime, and not on the BPEL host itself. Therefore, we can conclude that system load is only affected to a negligible extent.

On the other hand, the overall performance of the ManagedRoaming composition is affected to a certain extent. This performance penalty primarily results in a higher average response time as well as a decreased number of transactions per seconds. However, it is important to keep in mind that the VieDAME runtime is decoupled from the BPEL host, and the transmission of each monitoring event or adaptation request requires a network connection. In this regard, performance can be improved by various means. One way to improve the performance is to change the deployment model, that is, to deploy the VieDAME runtime and the composition engine on the same host. This will improve performance as it minimizes latency introduced by the network stack. However, each monitoring request, as well as adaptation request, still requires a socket connection, which is not necessary in case the VieDAME runtime and the composition engine are deployed on the same host. For such scenarios, an in-memory transport of the monitoring and adaptation information would be more efficient. This will be subject of future work.

Another important lesson learned is related to the memory consumption on the VieDAME hosting machine (not shown here). We observed an ever increasing usage of main memory by the JVM that executed the VieDAME runtime. The problem in this regard were the sliding windows that were configured for monitoring. Consider the EPL statement from Listing 13:

```
1 select
2   avg(executionTimeMillis), operationName
3 from
4   AddSubscriber.win:time(60 min)
5 group by
6   operationName
```

Listing 13: Sliding Window with 24 Hours Event Retention Time

The statement from above uses a 60 minutes sliding window. This means that all events that enter this window will be retained for 60 minutes *in memory*. In our case, the ServiceInvocationEvents can be rather larger objects, since they store the payload of service request and response messages. Hence, when using sliding window based metrics in monitoring statements, it is important to minimize either the number of objects that will be retained by the ESP engine (e.g., by using filtering clauses) or consider alternative windows, such as the `length(n)` window which retains the last `n` events.

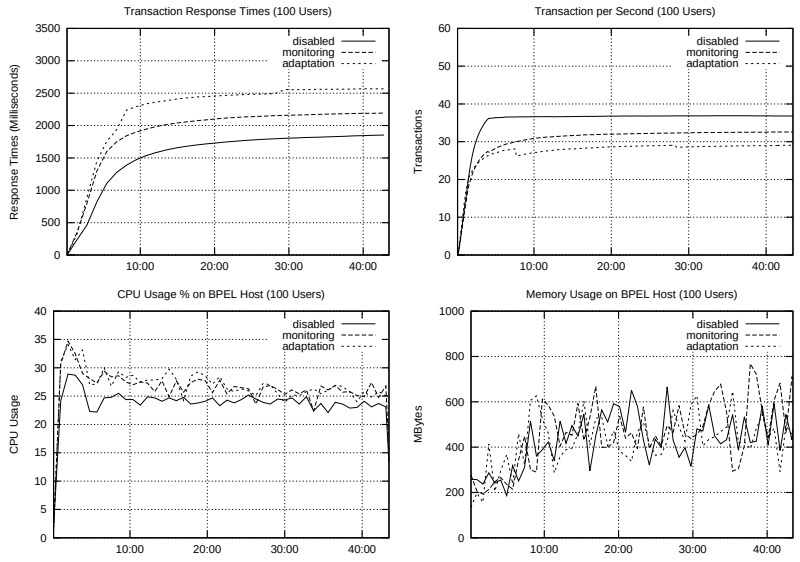


Figure 31: Results of 100 VUsers Loadtest

8.2.2 End-To-End Benchmarks

In this section, we focus on a real-world scenario where the VieDAME system can develop its capabilities to full extent. That is, the adaptation features are leveraged to improve the performance of a composite service where one or more of its individual services show performance issues. Again, we adopt a multi-stage evaluation to compare the behavior of the ManagedRoaming composition that is executed in a default Apache ODE environment (stage (1)) with a VieDAME enhanced ManagedRoaming composition that leverages a Static Selector (stage (2)) and a VieDASSL Selector (stage (3)). In the stage (3) scenario, we can therefore also emphasize on the benefits of the VieDASSL Selectors over the Static Selectors from a performance point of view. Both stage (2) and stage (3) scenarios where equipped with a transformer configuration that compensated the interface mismatch between the IPlus and the Phonica service.

Outage Simulation

During this end-to-end load test, we simulated a *service outage* to highlight the performance gain in process execution time when replacing a slow service with a reasonable performing alternative. To this end, it is assumed that the service invoked by the RegisterRoamerAbroad activity, e.g., IPlus, provides two service instances which are hidden behind a load balancer. This is a common approach to increase service performance and dependability in real world applications. For our experiment, we assume that one of the two service instances behind the load balancer fails. Therefore, the load balancer excludes the faulty service instance from its service queue. This means that the remaining instance must service all incoming requests, resulting in higher response times for the client, that is, the BPEL process.

Additional to this outage simulation, and contrary to the multi-stage evaluation setup we presented earlier, we assigned realistic processing times to all individual services that were part of the ManagedRoaming composition. Thus, the processing times are inherently higher in the end-to-end scenario, due to this synthetic processing delay of each individual service. For the exact parameter values assigned to the Phonyfone hosted Web service instances, please refer to Table 12. The given delay is an upper bound, the real delay is calculated using the formula $delay * \frac{random.nextInt(factor)}{100}$.

Web Service	Delay	Factor	Failure Rate
QuerySubscriber	250ms	90	0%
QueryDeal	180ms	90	0%
CreateOrder	300ms	90	0%
CreateBillingRecord	500ms	90	0%

Table 12: QoS Attributes of ManagedRoaming Individual Services

SoapUI was used to schedule the outage by creating a test case that includes only two test steps: (1) A delay test step that introduces a 25 minute wait state and (2) the actual invocation of the `setQos()` operation to increase the processing delay of the RegisterRoamerAbroad partner service. The default values, that is, the values before the service outage, for the `setQos()`

QoSAttribute	IPlus	Phonica
Delay (ms)	150/1500/2500	250/1500/2500
FailureRate (%)	0/0/0	0/0/0
CallSetupFee (€)	0.05	0.03
VoiceTrafficRate (€)	0.32	0.32

Table 13: Roaming Partner QoS Settings

operation as well as the values during the outage simulation for 50 and 100 user scenarios are listed in Table 13. The delay factor was set to `90` in all cases. Furthermore, the table also lists the values of the rates and fees that are considered for service selection by the stage (3) VieDASSL Selector.

Based on the values from Table 13, the Static Selector used in the stage (2) setup favors the default IPlus service due to slightly better response times. The VieDASSL Selector of stage (3) chooses the Phonica service since it does not violate the configured response time threshold and offers better domain-specific QoS.

Of course, the ManagedRoaming composition executed in the stage (1) scenario can only invoke the default (IPlus) service. Due to these preferences, we increased the processing time of the IPlus service in the stage (1) run during the service outage simulation to 1500ms (50 users scenario) and 2500ms (100 users scenario), while for the stage (3) run, we increased the processing time of the Phonica service to 1500ms and 2500ms for the 50 and 100 users scenarios, respectively.

Figure 32 illustrates the results of the load tests. The charts on the left show the results for the 50 users scenarios, while the charts on the right presents the numbers for the 100 users scenarios. The two upper diagrams display the average response time. The diagrams on the bottom highlight the transactions per second. Table 14 summarizes the numbers for the 100 users scenario, as well as the normalized costs incurred by the roaming partner service invocations that have to be paid by Phonyfone.

```
1 selection {
2     inCase(responseTimeMs5Min, less, 400.0) {
3         min(callSetupFeeEur, 0.4)
4         min(voiceTrafficRateEur, 0.6)
5     } {
6         min(responseTime5MinMs, 0.49)
7         max(availabilityPercent, 0.51)
8     }
9 }
```

Listing 14: VieDASSL Selector for ManagedRoaming

8.2.3 Results and Discussion

The results before the service outage at 25:00 show, as expected, a processing overhead introduced by VieDAME. However, during the service outage (from 25:00 on), the results clearly

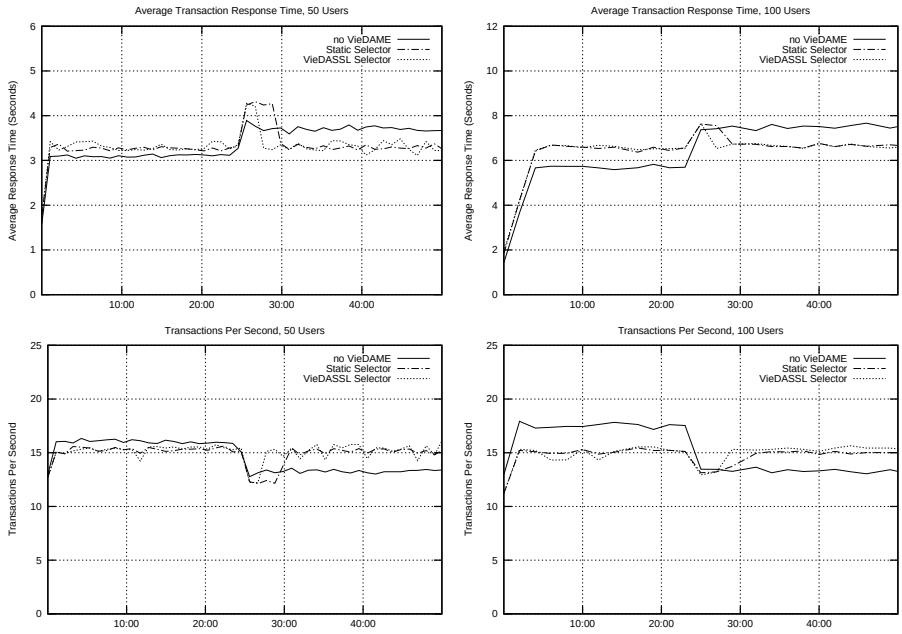


Figure 32: Response Times and Transactions per Second during VieDAME Loadtest

Performance Metric	**Stage 1**	**Stage 2**	**Stage 3**
Transactions per second	13.98	15.02	15.13
Average response time	6.18	5.82	5.76
CallSetupFee	2097	2097	1677.6

Table 14: 100 Users Scenario Results Summary

show the performance optimizing characteristics of the VieDAME setups. The high response time of the IPlus roaming partner results in a higher average response time and a lower number of transactions per second in the default Apache ODE environment for both the 50 and 100 users scenarios during the remainder of the test runs. In contrast, both the stage (2) well as the stage (3) results show only a short period of lower process performance caused by the outage. For both VieDAME scenarios, overall process performance is brought back to where it was for the rest of the load test. Though, the stage (3) based process executions have two advantages over the stage (2) executions. First, the domain-specific selector minimizes the domain-specific QoS aspects associated with the roaming partners, while tracking both the response time and availability of the alternative roaming service providers. Thus, the domain-specific selector effectively saves costs while not compromising process performance in terms of availability and execution time. Second, when comparing the time needed for the selector to switch over to a better performing alternative, the time-window based QoS features that were enabled in the stage (3) setup can

considerably minimize the critical time frame where process performance is affected by the service outage. The domain expert can also adjust the selector thresholds and observation periods, such as, the last *n* invocations instead of a fixed time period, in case of changing requirements.

8.3 Qualitative Discussion

Besides the performance related aspects that were presented in the previous section, there are qualitative aspects to our approach that need consideration. In the following, we provide an overview of the strengths and limitations of our approach. Additionally, we present an informal stakeholder analysis to show how different stakeholders of our system are supported by our approach.

8.3.1 Strengths

Non-Intrusiveness. As stressed in Chapter 5, we tap the message exchange within a composite service by the means of the interception and adaption layer (IAL). Together with the composition engine adapter (CEA), they provide the means to transparently intercept and adapt the service interaction in composite services. The strength of this approach lies in the abstraction of the underlying engine and messaging stack specifics. Considering the implementation of the IAL in our VieDAME system, we leverage AOP to this end, which enables us to weave the code required to access the message context provided by the engine's messaging stack into the system at runtime. Hence, this approach effectively minimizes the coupling between the base-system, which is, the composition engine, and our VieDAME runtime. Another benefit of using AOP to extend an arbitrary composition engine is the ability to enable and disable the related AOP advices at runtime. This, in turn, allows us to selectively enable or disable certain features of our VieDAME runtime, such as the adaptation capabilities. The CEA, on the other hand, covers the specific handling of the messaging stack that a composition engine uses. Therefore, the CEA for a particular messaging stack, such as Apache Axis, can be reused for the integration with additional composition engines that also rely on this particular messaging stack. Taken together, the IAL and CEA make our approach applicable for a variety of current and future composition engines.

Inherent Simplicity. Due to the reduction of a composite service to a corresponding message exchange, we believe that our approach does not suffer various issues that may arise when leveraging a more detailed model of composite service executions. Other abstraction levels, such as operating on the in-memory representations of composite services, clearly have more adaptation capabilities. These may include the replacement of process fragments or state manipulations. However, our simple view on matters minimizes the potential side effects that may arise when relying on lower-level abstractions.

Holistic and Generic Model of Monitoring Requirements. The event-based model that we leverage on top of the message based abstraction provides a generic tool to model the monitoring requirements of composite services. This event-based model enables us to cover monitoring

requirements from various stakeholders and business domains, since it acts as a common base of understanding of the situations of interest within a particular business domain. Moreover, and important for the adoption of a monitoring system, our system enables an holistic view on all monitoring data that is relevant for a specific monitoring requirement. That is, we provide various event source adapters that can be used to include information that stems from outside the composite service and convert this information into events that can be considered and correlated within our monitoring system.

Third Party Integration The design of the system architecture that we described in Chapter 5 and Chapter 6 proposes a management service on top of the core components. This management service can be leveraged to integrate the monitoring and adaptation capabilities into third party software. To this end, VieDAME provides a REST based implementation of this management service that allows a distributed access to the core functionality.

Domain Specific Service Selection. The various reasons for a domain specific language (DSL) that enables the definition of service selection strategies have already been stressed extensively in Chapter 6. However, we want to emphasize here that such a DSL is highly important for the acceptance of such service selection mechanisms, as it enables so called *domain experts* to define the service selection strategy. Thereby, system engineers, who previously were the target audience responsible for such tasks, are not required to be instructed how service selection strategy should be implemented. Domain experts themselves find the tool support they require in such a DSL, which enables them to specify their requirements in a natural fashion. Hence, we argue that our DSL based service selection approach effectively supports a separation of concerns in this regard within an enterprise.

From a technical point of view, one feature that sets us apart from various related service selection systems is the runtime adaptivity of the foundational QoS model and the corresponding mapping of such modifications in our DSL. Hence, our system might have an advantage over related work in mission critical zero-downtime scenarios.

Expressive Event Processing Language. As aforementioned, we leverage the OpenSource engine Esper to cover the event stream processing that our approach requires. In this regard, Esper provides an event processing language (EPL) that allows to model event stream queries and patterns. The use of this EPL within our system allows us to equip VieDAME users with a powerful, domain-specific tool that covers monitoring requirements in a declarative fashion. In the following, we will provide some short samples that should give the reader an impression of the power of Esper's EPL.

As an example how we can leverage the EPL to cover complex monitoring requirements in a declarative fashion, consider the time constraints associated with the execution of the AddSubscriber and ManagedRoaming compositions. In this regard, it is important that the AddSubscriber and ManagedRoaming compositions are invoked in the correct order. Listing 15 displays the related EPL statement.

```
every addSub =
  AddSubscriber ->
    (timer:interval(10 sec)
  and not
    ManagedRoaming(msisdn = addSub.msisdn)
```

Listing 15: Multi-Composition Time Constraint

The above pattern is triggered upon an AddSubscriber message that is not followed by a related ManagedRoaming message within a time frame of 10 seconds. The two corresponding messages are correlated via their `msisdn` tag that uniquely identifies subscribers in our scenario. However, this pattern assumes that ManagedRoaming messages, that do not follow related AddSubscriber messages within 10 seconds, will never arrive. If we want to detect situations where a corresponding ManagedRoaming message arrives too late *or* does not arrive at all - within an additional, longer time frame (5 minutes) than the constraint violating 10 seconds time frame - we can use the pattern from Listing 16.

```
every addSub =
 AddSubscriber ->
  (
   (timer:interval(5 min) and not ManagedRoaming(msisdn=addSub.msisdn))
    or
   (
    (timer:interval(10 sec) and not ManagedRoaming(msisdn=addSub.msisdn))
     ->
    AddSubscriber(msisdn=addSub.msisdn) where timer:within(10 seconds)
   )
  )

```

Listing 16: Detecting Absent Messages

Another important matter in the SubscriberActivation scenario is the sending of the confirmation email that contains important account details for the new customer. However, this email message is not part of the monitored service compositions, but originates from a system that lives outside of the composition engine. Therefore, we are required to configure an appropriate event source adapter that accepts this email message and converts it to an event that can be considered within an EPL statement. The statement in Listing 17 shows how this external event can be leveraged to describe the fact that the SubscriberActivation process is not complete until both the AddSubscriber and ManagedRoaming compositions have completed successfully as well as the customer confirmation email has been sent out. For brevity, the corresponding event source adapter configuration can be found in the Appendix and is not shown here.

```
every addSub =
  AddSubscriber ->
    (timer:interval(10 min)
  and not
     NotificationEmail(msisdn = addSub.msisdn)
```

Listing 17: Customer Email Confirmation

As a last example, we will consider a case where we want to find out the best selling products within the last three days. The corresponding EPL statement is provided in Listing 18.

```
1 select
2   tariff, count(tariff)
3 from
4   AddSubscriber.win:time(24 hours)
5 group by
6   tariff
```

Listing 18: Finding Popular Tariffs

In summary, we believe that the ESP platform that we integrated into VieDAME is capable of covering most of the monitoring requirements that may arise in typical enterprise scenarios.

8.3.2 Limitations

We will now emphasize the potential limitations that are inherent to our approach, both with regard to the monitoring as well as the adaptation capabilities.

Strict Message Orientation. Besides the advantages that we already stressed enough, the message based abstraction that we use has also an inherent downside. In this regard, functionality specific to a composition technology or composition engine cannot be addressed using our approach. As an example, we cannot cover the alarm events supported by WS-BPEL when considering solely the message exchange. These time related events can be leveraged in WS-BPEL compositions to trigger the instantiation of a composition either after a certain duration or at a specific time. In this case, no incoming message triggers the composition, hence, our approach cannot generate a ServiceInvocationEvent in this case. Nevertheless, the service interactions within such compositions are in most but the rarest cases realized by the means of a message exchange, and hence, are covered by both the monitoring and adaptation features of our approach.

Stateful Services So far, we did not consider the aspect of *state* associated with service invocations. That is, the interaction between the service consumer and the service provider does not depend on the history of previous interactions between these two parties. Such service are called *stateless services*. Contrary, *stateful services*, which are also referred to as *conversational services*, maintain certain state information between operation invocations issued by service consumers [125]. Therefore, stateful services impose special requirements on runtime service selection mechanisms as implemented in the VieDAME system. In short, there is no comprehensive support for stateful services in our approach. Nevertheless, simple state information that is part of the message payload, such as transaction identifiers, can be extracted or even rewritten by our system.

Number of Alternative Service Candidates. As we stated in Chapter 6, we assume a relatively small number of alternative services that are considered as a replacement for a service originally defined in a service composition. We justified this assumption by the contractual obligations that are present in most enterprise scenarios when external partners are involved. Nevertheless, we did not consider scenarios where more than ten alternative services are avail-

able for an original service, and therefore cannot quote on the scalability of this aspect of the service selection components.

Performance Impact. Clearly, when adopting our approach in high load scenarios, one must consider the performance penalty that is imposed by the VieDAME system. In this regard, the experiments in this chapter proved that the performance of a composite service is affected by our VieDAME runtime, both in terms of the average response time as well as the number of transactions per second. Moreover, the event processing system that we use requires a thoughtful configuration. What we mean here is that it can be the source of a dramatically increased memory load when using event streams that retain event objects for a long time, especially in high load scenarios where a huge number of event objects are generated.

8.3.3 Stakeholder Analysis

When considering the approach presented in this thesis from a corporate point of view, there are several parties that are affected by the introduction of the VieDAME system into the IT landscape of an enterprise. In the following, we identify these *stakeholders* and provide an *informal analysis* of their position towards a monitoring and adaptation system as manifested as in our VieDAME runtime.

System Engineer Perspective. From a system engineer's perspective, who leverages VieDAME as a framework, choosing the message level as the layer for interception and adaptation of service interaction results in broad compatibility with existing and future composition platforms. This design decision also minimizes potential side effects compared to operating on other abstraction levels, such as the in-memory process representation or the schema level. Additionally, the service oriented architecture of our approach can be easily extended to cover additional requirements and enables seamless integration into third party applications.

System Administrator Perspective. As described previously, setting up the VieDAME runtime system to enhance an existing composition infrastructure is non-invasive. Nevertheless, the AOP based approach that was chosen for the integration of the monitoring and adaptability capabilities requires access to the hosting environment of the composition engine. This might be a problem in highly restrictive environments. System administrators will further notice the processing overhead introduced by the runtime features of our system, which will result in a higher system load. On the upside, our holistic approach with regard to composite service monitoring enables system administrators to cover monitoring requirements in a centralized and powerful fashion.

Domain Expert Perspective We argue that the primary stakeholders of the VieDAME runtime system are *domain experts*. Similar to a business analyst, a domain expert has solid knowledge of the business domain, such as the telco domain from our case study. Moreover, and different from the business analyst, a domain expert has more in depth know-how of the technical details of the underlying services and systems. Domain experts only have basic programming

skills, which separates them from developers or system engineers. Considering their skill set and position within the corporation, we argue that domain experts are primarily entrusted with and responsible for tasks that can be effectively supported by the capabilities of our VieDAME system. Their duties can include the definition and implementation of monitoring requirements, as well as identifying the optimal set of individual services in service composition scenarios where more than one candidate service is available for a specific task. Nevertheless, they require special tool support to cover these requirements, or need to delegate the implementation of their requirements to system administrators or system engineers. However, such a delegation bears potential problems, such as its impact on time to market or cognitive dissonances between the requester and the implementer.

To this end, our approach provides an intuitive and goal oriented language to describe domain-specific service selection and monitoring requirements. The main advantage of leveraging VieDASSL for the definition of service selection strategies is the achievement of a clear separation of concerns in large enterprises. Domain experts should not be required to learn a programming language to implement monitoring requirements or service selection strategies. In fact, they should focus on the business related aspects of these concerns. Moreover, the runtime adaptability features of VieDASSL and the underlying QoS model implementation address very short-term market requirements. These requirements might be imposed by a regulator or law enforcement, but also by customer feedback or a competitor's new offering. Minimizing time to market is crucial, not only for telco enterprises as shown in this work, but for the majority of other businesses. Our approach can be leveraged to effectively minimize the time to market, resulting in a key advantage in today's highly competitive businesses.

8.4 Summary

In this chapter, we examined the performance and qualitative aspects of our approach and the related proof of concept implementation in the form of the VieDAME runtime system. A WS-BPEL implementation of the ManagedRoaming composition from our running example was used as the foundation for the experiments that were conducted. These performance experiments relied on a multi-stage setup where in each stage an individual feature of our VieDAME system was stressed to highlight its impact on the overall performance. To this end, we can conclude that there is a certain performance penalty imposed by the VieDAME runtime system, both in terms of the composite service performance as well as system load. However, we further presented an end-to-end performance evaluation where we showed that the service selection features of VieDAME can be used to effectively improve certain QoS aspects, including performance and cost related QoS. Along with these performance aspects, we also discussed the qualitative aspects of our approach. That is, we listed the general strengths and limitations of our system, as well as identified its stakeholders and their perspective towards our method. Prior to adopting the VieDAME system, the applicability of our system should be considered with regard to the strengths and potential limitations that we discussed.

CHAPTER 9

Related Work

In this chapter, we give an overview of work related to the two main subjects of this thesis, that is, monitoring and adaptation of composite services. Work related to service monitoring will be covered in the first section. Due to the large variety of different approaches, the section on monitoring is split into a general section that discusses various techniques to acquire monitoring related data, and three specialized sections that encompass *assertion based monitoring*, *monitoring of temporal properties* and approaches related to *event based monitoring*.

The second section focuses on work in the area of *runtime adaptation of composite services*. In this section, we compare our approach with related work in terms of several criteria, such as the underlying QoS model or the specification of adaptation rules. We also present a comparison matrix that provides a quick overview of related adaptation work.

9.1 Monitoring of Service Oriented Systems

Various approaches for monitoring QoS properties exist. In this work, we present a technique that intercepts outgoing and incoming requests to capture QoS related information for composite services as well as its constituent services. However, this is not the only viable approach to realize a service monitoring system. In the following, we will give a short overview of other techniques that can be applied to implement service monitoring. Interested readers might refer to [146, 131] for more information.

Following the taxonomy provided in [131], four principle methods for QoS measurement can be distinguished. *Provider-* or *server-side instrumentation*, as presented in [17, 31], describes techniques that alter the service provider implementation to gather QoS related data. In its simplest form, this monitoring technique wraps the method invocation that is subject to the QoS observation inside a "stopwatch" method that determines, for instance, the execution time of the method invocation. However, such an approach is rather invasive as it requires direct modifications in the source code of the service implementation. This drawback can be addressed by applying modern tooling approaches, such as aspect oriented programming (AOP), to weave the

required monitoring code into the base system in a unobtrusive fashion. On the upside, monitoring methods that rely on provider-side instrumentation allow for fine-grained measurement of QoS related attributes, as the instrumentation of the server side code can happen on various levels of the request processing logic. Such attributes include, for instance, the wrapping time or execution time of service invocations [139].

In [155], 15 different factors that influence service performance have been identified. However, not all of these factors can be covered using pure provider-side instrumentation approaches. In fact, client-side attributes, such as response time or availability that we use in our QoS model from Chapter 4, cannot be determined using provider-side instrumentation. Therefore, various client-side approaches exist. *Intermediaries* are computational elements that lie along the path of web transactions [28, 29]. With regard to monitoring, such intermediaries may be used to reroute service invocations to enable the collection of QoS related data. SOAP intermediaries [96] bring the idea of intermediaries to Web services. Effectively, such an intermediary acts as a proxy that routes messages to a service provider on behalf of the service consumer. However, such a proxy based approach inherently introduces a single point of failure, as well as the problem that service providers and service consumers have to agree on the intermediary to enable the monitoring capabilities.

The last QoS monitoring techniques that we will consider fall into the category of *client-side* approaches. *Probing* is a monitoring technique where the computation of QoS attribute values is conducted by sampling requests towards a service. Such probing requests are intended to simulate typical service consumer-producer interaction patterns in order to determine the performance or dependability of a service. The foremost advantage of the probing approach is its location transparency and its flexibility. Such probes are fully decoupled from the system that they observe, and can be easily extended to support additional probing capabilities. However, not all service operations might be eligible for probing to determine related QoS values. Service operations that behave in a non read-only manner might not be observable as operation invocations are not idempotent and may introduce side effects on the service provider side. *Sniffing*, on the other hand, is a technique that relies on the low-level inspection of network packets that are transmitted during a service invocation. This technique allows to accurately calculate various QoS related values, such as response time or latency, by capturing, correlating and aggregating network packets. Certain approaches, such as the one presented by Rosenberg et al. in [139], combine sniffing with other monitoring techniques to get a more accurate view on monitoring data. Furthermore, Michlmayr et al. [101] present a framework that combines the advantages of client and server side monitoring and informs interested subscribers of QoS values using event notifications.

When considering the work presented in this thesis, our method represents a hybrid approach to QoS monitoring that incorporates both *client side* as well as *server side* techniques. This is due to the fact that when considering the monitoring of the individual services, our approach implements a client side method. The monitoring of QoS related attributes of the whole composite service, however, can be seen as server side monitoring. In [111], Oberortner et al. present various client-side patterns that can be applied to measure QoS related service properties. Our method uses the *QoS interceptor* pattern from this catalogue. Such a QoS interceptor can implemented by either leveraging certain middleware provided capabilities, that is, invocation hooks

that are provided by the messaging stack, or can be realized by the means of aspect oriented programming (AOP). Our approach uses AOP to weave the monitoring code into the composition engine, and hence, we do not require that the underlying messaging stack provides such invocation hooks.

9.1.1 Assertion based Monitoring

A large body of work exists that can be classified as assertion-based monitoring. Baresi and Guinea [22, 23] propose WSCoL, a constraint language for the supervision of WS-BPEL based service compositions. Monitoring information is specified as assertions on the composite service definition, that is, the WS-BPEL process definition. Their approach uses Aspect-Oriented Programming (AOP) to check the assertions at runtime. In [70], Baresi et al. propose SECMOL, a general monitoring language, that sits on top of concrete monitoring languages such as WSCoL. Thus, it is flexible by separating data collection, aggregation and computation from the actual analysis. Using the expressiveness of an event processing language enables us to model monitoring requirements similar to those that are supported by WSCoL. Additionally, WSCoL provides a much more integrated way of accessing the composite service definition, and hence, is more expressive in this regard. However, their approach is tightly integrated with the underlying composition technology, that is, WS-BPEL. In contrast, our approach abstracts from composition technology related details and thus can be applied in scenarios where WS-BPEL does not represent the target platform. No information is provided with regard to service interface compatibility checking, which is an requirement for our method.

Mahbub et al. [95] early work presents a framework for managing the monitoring requirements of WS-BPEL based compositions based on event calculus (EC). To this end, such monitoring requirements truly specify behavioral properties of the service composition, as well as assumptions about the behavior of the system. The behavioral properties are automatically extracted from the composite service specification and transformed into EC formulas. Assumptions are, on the other hand, manually specified by the provider of the service composition. For their approach, they assume that the composition engine emits events which are sent as string streams to an event receiver, which, in turn, persists relevant events to an event database. A monitor component then checks the compliance of these events against the behavioral properties and the assumptions that were made, eventually generating a deviation report in case of violations. Hence, and contrary to our approach, they are using an asynchronous approach to detect deviations in these assumptions and behavioral properties of a service composition. Our synchronous approach allows to detect similar situations of interest by leveraging event processing techniques, and might be more appropriate in situations where a timely response to such situations is required.

Sun et al. [144] also propose a monitoring approach based on AOP. Their goal is to check business process conformance with the requirements that are expressed using WS-Policy. The properties of a Web service, including temporal properties as well as behavioral properties, are described as an Extended Message Sequence Graph (EMSG). This EMSG is a graphical representation of the Web service properties that helps to understand service behavior. To check various runtime message sequence information, this EMSG is transformed into a Message Event Transferring Graph (METG). A runtime monitoring framework is then used to monitor the cor-

responding properties that are analyzed and checked against the METG graphs. Similar to our approach, Sun et al. use AOP to enhance a composition platform with monitoring capabilities. They use a very intuitive, visually supported method to express monitoring requirements using diagrams that are similar to UML sequence diagrams. Their approach, however, requires WS-BPEL as the underlying implementation technology as their EMSC directly maps to WS-BPEL elements. Moreover, they do not support monitoring data that originates from outside the monitored service composition.

9.1.2 Monitoring of Temporal Properties

Another group of work focuses on monitoring temporal properties of Web service compositions. In this regard, Kallel et al. [86] propose an approach to specify and monitor temporal constraints based on a novel formal language called XTUS-Automata. XTUS-Automata combines Timed Automata (TA), which are automata extended with time constraints, and the Time Unit System (TUS), a means to specify time instants and intervals, into a novel language that supports relative and absolute temporal properties. The definition of these temporal constraints is supported by generic specification patterns, which represent a generalized description of commonly occurring requirements. Monitoring itself is based on AO4BPEL [40], an aspect oriented extension of WS-BPEL, which enables more modular and dynamically adaptable WS-BPEL processes. Similar to our approach, their monitoring method supports the specification of temporal aspects of service interactions within composite services. Moreover, this specification of temporal aspects is not limited to service interactions but can be used with arbitrary WS-BPEL activities. However, and in contrast to our approach, their method directly alters the composite service definitions by augmenting it with AOP aspects, which might not be possible in certain enterprise environments.

Barbon et al. [21] propose RTML (Runtime Monitoring Specification Language) to monitor temporal properties. Their approach translates monitoring specifications defined in RTML to Java code to monitor instances of services and process classes. For their implementation they extend an OpenSource WS-BPEL engine to access process relevant data, which can be referenced in RTML specifications. In this regard, their approach is tightly integrated into the composition engine. Contrary to this approach, we do not need to translate a formal language to specific monitoring code. We simply represent the message exchange in a composite service as a stream of events that inherently have a temporal order. This event stream can then be queried or observed to detect patterns of interest that resemble monitoring requirements.

Zhang et al. [166] present an approach to represent the temporal properties of service compositions using Property Sequence Charts (PSC). PSCs are realized as a subset of UML 2.0 sequence diagrams. They also leverage AOP technology to tap the execution of WS-BPEL based compositions. To some extent, their approach is similar to the work of Sun et. al [144], since they rely on a graphical notation to define composite service properties. However, Zahng et al. focus on the temporal properties of service compositions. The use of visual support tools to define service execution constraints enables the application of these approaches where users have no knowledge of a formal language. In comparison with our approach, their method only provides offline analysis of traces that have been recorded, and is not capable of integrating external data that might be relevant for such temporal constraints.

9.1.3 Event Based Approaches to Service Monitoring

Finally, we present some examples of related work in the area of service monitoring that cannot be strictly categorized as assertion based approaches or targeted at monitoring the temporal properties of service based systems. However, these research projects follow an event based methodology towards service monitoring, which is similar to our method.

Suntinger et al. [145] provide a visualization approach using the notion of an Event Tunnel. This Event Tunnel framework provides interactive and visual support for business analysts in exploring business incidents. It allows to visualize complex event streams of historical information and enables the detection and analysis of business patterns. This can be used, for example, to detect and prevent fraud. In our monitoring approach, we also leverage an event based abstraction to find anomalies in service behavior. However, for the time being, we do not provide visual support for analyzing event streams. Our approach, on the contrary, provides seamless integration into existing composite service infrastructures, whereas the Event Tunnel framework requires that event data is preloaded into an event space, which is a specialized mass storage for events.

Beeri et al. [32, 33] propose a query-based monitoring approach and system. It consists of a high level query language allowing to visually design monitoring specifications. These monitoring specifications are then compiled into WS-BPEL compliant process descriptions, which can be deployed along with the WS-BPEL process that requires monitoring. For their approach, they assume that the WS-BPEL process instance that is under consideration for monitoring generates events. These events are then used by the generated monitoring process for performing the specified queries. The visual query designer also allows to define customized reports based on those queries. Complementary to our work, we also allow a query based approach based on an event query language. However, we do not generate distinct query processes, but rather leverage event stream queries that operate on the data provided by the message exchange within a service composition. We also support external data to be integrated into our queries, which is not explicitly supported by their monitoring system.

Michlmayr et al. [101] combine client- and server-side QoS monitoring to achieve a higher accuracy of QoS monitoring. Based on the monitoring data, their approach allows to detect simple performance-specific SLA violations using an event query language. SLA obligations are expressed using a simple syntax and are then transformed to event processing queries. Similar to our approach, they leverage event stream processing technology to deal with SLA obligations. However, their approach is mainly targeted at performance and dependability related QoS, while our approach also enables the detection of business related QoS deviations.

Wetzstein et al. [154] use a dependency analysis framework based on machine learning to identify factors that influence business process performance. Their approach represents the QoS of a composite service on different levels. Key Performance Indicators (KPI) are defined by business analysts and measure the success of a business process as a whole, while Process Performance Metrics (PPM) capture single facets of business process performance. Together with performance and dependability related QoS metrics, PPMs directly influence a business process KPIs. Their framework relies on events published by the composition engine to capture PPMs, while QoS metrics are captured by dedicated QoS monitors or instrumentation of the involved systems. The fundamental idea of their approach is to perform dependency analysis of historical

process instance data. The result of this dependency analysis is a decision tree that represents the most influential factors on process performance. This decision tree can then be used by business analysts to determine potential issues in a business process. In contrast to our monitoring system, they provide a visualization of potential "hot-spots" in composite services, which indeed helps business analysts to conduct optimizations. Our approach is more targeted at a holistic view on monitoring data that is produced in enterprise environments, as our system allows to integrate data that originates from outside the monitored composite service.

9.1.4 Discussion

In contrast to the aforementioned approaches, we do not use assertions to define what needs to be monitored using a proprietary language. We propose a holistic and flexible monitoring system based on ESP technology. Our system measures various QoS statistics by using a non-intrusive mechanism and has access to message payloads of in- and outbound requests of a composition engine. Additionally, our approach allows to connect external event sources. This enables us to integrate information that does not originate from the monitored composite service itself, but rather from external systems. Such information might be of interest, especially in scenarios where composite services are not the only means to realize enterprise business processes. From a technological point of view, ESP technology is very powerful, since it enables us to operate on event streams. In our case, these event streams contains the monitoring data. Various event processing operators, for instance, for temporal or causal event correlation, can then be used on event streams to define what monitoring information needs to be retrieved. All of these steps can be conducted at runtime. Domain experts can define event queries in an SQL-like declarative style, without the need for programming expertise. Moreover, our monitoring system provides the means to verify the compatibility of service interfaces that are relevant for the message exchange within a particular service compositions. To the best of our knowledge, none of the approaches discussed above provide such means. Our service interface surveillance method is seamlessly integrated into the monitoring runtime.

9.2 Adaptation of Composite Services

There is a recognizable trend that SOA is becoming mainstream and the underlying infrastructures, such as composition middleware, cloud platforms and services, increasingly adopt principles of self-adaptive software systems [118]. The benefits are software applications that are better manageable and more robust towards requirements changes during its lifetime, which in turn reduces the operational costs. Software applications adopting self-adaptation mechanisms typically implement an adaptation loop, consisting of several process, sensors and effectors [140]. Variants of this adaptation loop are the MAPE-K loop from the Autonomic Computing space [89, 77], as introduced in Chapter 2, or adaptation management introduced by Oreizy et al [118].

Another noteworthy trend is the adoption of AOP techniques to realize adaptivity in composite services [40, 87, 45]. In the majority of cases, AOP is used as a tool to extend a base system, which is, the composition engine, with the mechanisms that are required to implement various

adaptations, for instance, service rebinding methods as in our approach. However, AOP technology is also leveraged by certain research projects to truly manage cross cutting concerns [40]. That is, functionality that (re-)occurs at several points in the execution of a composite service. Prime examples for such cross cutting concerns are logging, security or transaction handling.

In the following discussion of work related to adaptation techniques, we focus on approaches from the Service-Oriented Computing [121] context that implement parts or the complete adaptation loop and compare it to our work. We categorize them according to five criteria as illustrated in Table 15. For each criterion, we additionally show the specific MAPE phase to illustrate where the proposed approach fits into the adaptation loop. We use the abbreviations M, A, P, E as described in Chapter 2. The *QoS Model* criteria identifies whether deterministic and non-deterministic QoS attributes are supported. The remaining sub-criteria classify whether the QoS model implementation is extensible at design- and/or run-time. *QoS Monitoring* identifies whether an approach supports monitoring of real-time QoS data within a composition and whether temporal aspects are considered, such as the average response time over the last 5 minutes. Moreover, interface compatibility refers to support for checking service interface compatibility as described in Chapter 5. The next two criteria address the *Selector Specification and Adaptation*. The classification of the selector specification roughly follows the taxonomy from Delgado et al. [47]. We leverage language-based approaches, that is, based on an imperative, object-oriented or functional language, or approaches based on a logic. The adaptation sub-criteria define whether a selector can be dynamically adapted at runtime or is just simply adaptable at design-time, thus requiring a redeploy and reload of the process. Finally, the *Service Mediation* describes whether the approach is able to deal with service interface heterogeneity. We distinguish between (a) simple message-based mediation, where the original message can be transformed to the message expected by the target service (e.g., using XLST); and (b) Mediation flows where the mediation can have multiple steps and even call other services as part of the mediation execution, similar to what is supported by an Enterprise Service Bus [13]. Please note that due to space restrictions, not all related work that we will discuss in the following is listed in Table 15.

The approach presented by Baresi et al. [27, 24] shares many commonalities with our approach. They propose the Web Service Recovery Language (WSReL) which allows to define complex recovery strategies that include retry and rebinding of Web services. These recovery strategies can be seen as some form of selectors that can be defined at design-time, however, they are not adaptable at runtime. In [26], the same authors focus on service selection approaches where self-healing of service compositions is the primary concern. Finally, in [25], Baresi et al. present an integrated supervision framework for WS-BPEL based service compositions. In particular, they present the idea of self-supervising WS-BPEL compositions, which are compositions that autonomously assess their behavior and react via user defined rules. To this end, they combine WSCoL and WSReL into an integrated framework. As an extension over [27], they added backward recovery to WSReL as well as improved the interplay between WSReL and WSCoL.

Charfi and Mezini [40] propose an AOP extension for BPEL, called AO4BPEL. The main motivation behind their work is the lack of support for modularizing cross cutting concerns in WS-BPEL based service compositions. Such cross cutting concerns include logging, security

MAPE Phase	Criteria		Baresi et al. [25]	Charfi et al. [40]	Erradi et al. [56]	Zeng et al. [164]	Moscinat et al. [107]	Michlm. et al. [102]	Our Work
M	QoS Model	Deterministic Attr.	+	∼	+	+	+	+	+
		Non-deterministic Attr.	+	∼	–	+	∼	+	+
		Design-time Ext.	+	+	∼	–	+	+	+
		Runtime Ext.	–	–	–	–	–	–	+
M, A	QoS Monitoring	Real-time Data	+	–	+	∼	∼	+	+
		Temporal Aspects	–	–	–	–	–	–	+
		Interface Compatibilty	–	–	–	–	–	–	+
P	Selector Specification	Language-based	–	+	+	–	∼	+	+
		Logic-based	+	–	–	–	–	–	–
E	Selector Adaptation	Design-time	+	+	+	–	+	+	+
		Runtime	–	–	–	–	–	–	+
E	Service Mediation	Message-based	+	+	+	-	-	+	+
		Mediation Flows	∼	+	∼	-	-	-	+

– not supported ∼ partly supported + fully supported

Table 15: Comparison of Existing Approaches

or transaction handling. On the other hand, their framework also enables certain dynamic adaptations of WS-BPEL compositions. Their approach is based on a graph formalism to represent composite services, where the nodes represent activities, and the edges between nodes represent conditional transitions between activities. The join points in their graph formalism are defined by the execution of these activities. Their pointcut language is based on XPath expressions that describe the joinpoints where cross cutting functionality should be executed. An advice in terms of AO4BPEL is a BPEL activity that implements a crosscutting concern or a workflow change. AO4BPEL also allows for dynamically changing the deployed composite service by simply activating or deactivating aspects. Although they focus on different QoS attributes, which are specified by the Web service stack, such as transactions, reliable messaging and security, their AOP approach can be used to implement selectors. However, and contrary to our method which uses a tailored specification language to describe service selection strategies, AO4BPEL based selectors can be adapted at design-time only.

Erradi et al. [56] propose an adaptive middleware with a policy-aware selection mechanism, called Manageable and Adaptive Service Compositions (MASC). Besides technical QoS metrics, and similar to our approach, their MASC platform also considers business related metrics. Policies are specified in an event-condition-action (ECA) style, using an XML format based on WS-Policy. They have a fixed set of QoS attributes that they can measure and use in the adaptation policies. MASC currently supports static and dynamic customizations, as well as corrective adaptation at the messaging layer. In contrast to our work, their focus lies in the area of corrective adaptations, while our system also provides for optimizations with regard to QoS and business related metrics. Moreover, we provide an easy to understand DSL to describe service selection rules, while MASC relies on an extension to WS-Policy.

Ezenwoye et al. provide an approach [62] to transparently adapt WS-BPEL based compositions to compensate runtime faults and to improve performance with respect to the performance

of the constituent services. Their framework, called RobustBPEL2, provides a method that allows runtime discovery of replacement services, and adds self-optimizing behavior to existing WS-BPEL based compositions. While their work is similar to ours with respect to their aim to improve reliability and performance of composite services, they are using a proxy based approach to monitor process execution and improve process performance. On the contrary, our method leverages a lightweight adaptation and monitoring layer based on AOP to achieve these goals. RobustBPEL2 uses service registry to discover alternative services upon failure, but it does not incorporate selection criteria when multiple services are found. Our system chooses the most adequate service in advance to optimize performance and business related requirements.

Zeng et al. [164] proposed a foundational approach for QoS-aware service selection and optimization of composite services. Their approach uses a small set of given QoS attributes that can be used as part of the optimization. They provide two alternative QoS driven service selection approaches, where one is based on local optimizations, and the other on global planning. Conceptually similar approaches have been proposed in [15, 138, 163, 5, 6]. However, most approaches use different service selection techniques to optimize the overall QoS of a composition (e.g., skyline computation [6]). Similar to our method, their approach is not tailored for a particular composition technology. However, the selection of a specific service is determined by the optimization algorithm and cannot be adapted without re-executing the optimization. Moreover, our approach allows to reflect domain specific quality attributes within a service selection specification that is declared in a declarative style.

Moscinat et al. [107] focus on transparent runtime adaptability of WS-BPEL processes based on a fixed set of QoS attributes. Their main approach is to dynamically adapt the services in a WS-BPEL process by augmenting the WS-BPEL code with service selection code at deployment time. However, this implies that they cannot dynamically adapt the service selection logic. On the upside, their approach provides explicit support for adapting WS-BPEL based compositions that orchestrate stateful services, which is currently not supported by our method.

Michlmayr et al. [102] present VRESCO, a QoS-aware middleware that addresses adaptability of service-oriented applications. They propose several infrastructure components and services including dynamic binding and invocation, querying and composition that are based on a unified metadata model. Service selection can be specified by formulating service queries that return a rebinding proxy which implements different QoS-based rebinding strategies (e.g., on demand, on invocation, etc.). The service selection logic is predefined by choosing among a range of selectors depending on the application scenario and its need for continuous rebinding.

9.2.1 Discussion

Table 15 clearly shows that none of the related approaches supports runtime adaptation of the QoS model and selector implementations. This provides the means to add, remove or change attributes within the QoS model implementation and use the newly added QoS attributes in a selector logic without any downtime of the composition infrastructure. This is crucial especially for high-availability environments such as the telecommunications example introduced in Chapter 3. Moreover, our system incorporates a domain specific approach to runtime service selection in composite services. To this end, we extend the runtime adaptivity of the underlying QoS model to our selection strategy specification language. That is, changes in the QoS model are immediately reflected in our domain specific service selection language.

CHAPTER 10

Conclusion

This thesis proposed a novel and integrated approach for holistic monitoring and domain specific runtime adaptation of composite services. The main contributions of this work are threefold. First, we presented an adaptive QoS model (Chapter 4) that provides the foundation for the two remaining contributions. Such an adaptive QoS model enables us to effectively tackle short-term requirements with regard to additionally needed QoS attributes. Second, we introduced our approach for holistic monitoring in composite services (Chapter 5). To this end, we illustrated the fundamental assumptions that our approach is based upon, and derived an appropriate abstraction that makes our method inherently unobtrusive and technology agnostic. Third, the need for dynamic adaptations in composite services is addressed by a domain specific service selection method (Chapter 6). Our approach equips users with an easy to learn domain specific language. This language is tailored for describing service selection strategies in a natural style. That is, only domain know how is required to leverage our approach, hence, we argue that domain experts are the primary stakeholders of our service selection system. The identification of the requirements for such a system were driven by a genuine case study from the telco domain (Chapter 3). We used this case study to motivate the requirements that must be addressed when dealing with service composition scenarios that require monitoring and runtime adaptation. Additionally, we presented VieDAME, which is a proof of concept implementation of our approach (Chapter 7). This proof of concept implementation, together with a WS-BPEL based implementation of the case study, was used to validate the methods that we proposed (Chapter 8). Along with a performance validation, we provided an in-depth qualitative discussion of the strengths and limitations of our system. Chapters on related work (Chapter 9) as well as a detailed presentation of background information on concepts and technologies used in this thesis (Chapter 2) top off our work.

In the following, we will briefly review the research questions that we stated in Chapter 1, and discuss whether our contributions provide viable solutions to these questions. Thereafter, we conclude this thesis by providing an outlook on future work.

10.1 Research Questions Revisited

We derived two overarching research questions from the problem statement that we provided in Chapter 1. We will now briefly assess those questions. The first question was:

> *What is a method for managing holistic monitoring requirements in heterogeneous composite service infrastructures?*

In particular, this question is targeted at determining (1) a method to integrate platform and technology agnostic monitoring capabilities into existing composite services, (2) an abstraction to model QoS and business related monitoring requirements, as well as (3) the performance penalty introduced by such a method.

Considering (1), this characteristic is achieved by the *simple assumption* that interaction within a service composition is realized solely by the means of a *message exchange*. Together with an *AOP based approach* to integrate this functionality into an existing composition platform, there is only a minimal integration effort to use our system. Additionally, our simple abstraction frees us from potential issues that might arise when implementing monitoring and adaptation capabilities based on different abstractions, such as the in-memory representation of the composite service.

When we consider the abstraction that we use to model monitoring requirements (2), our method uses an *event based approach to composite service monitoring*. To this end, we interpret the message exchange in a composition as a stream of events. An event processing language is consequently used to operate on this event stream and allows to express complex monitoring requirements that span across several composite services. Moreover, our system treats monitoring data that originates from *external systems as first class citizens*. We achieve this by providing external event source adapters that abstract from the technical details of external systems, and transform such external events into a representation that can be used in monitoring statements. This external event support enables an *holistic view on monitoring data* that incur in heterogeneous enterprise environments. In our approach, such an holistic view also includes a perspective on service interface compatibility. In this regard, our system provides the means for automatic service interface surveillance. This allows us to determine interface incompatibilities that may arise when the constituent services of a service composition evolve independently.

Regarding the performance impact (3), we can conclude that a certain performance penalty is inevitable when opting for our method. This performance penalty can be explained when considering the deployment model of our VieDAME system. In this regard, we use a service based decoupling between the composition engine and the VieDAME system. We achieve this by installing only a small component of the VieDAME system, that is, the composition engine adapter, on-site with the composition engine. In fact, the capabilities of a VieDAME instance are delivered as a service on a remote host. On the upside, the evaluation showed also the performance optimizing characteristics of our system when leveraging the runtime service selection features. That is, the VieDAME system can effectively optimize several performance and dependability related properties, while it additionally tracks certain business related values, such as the call setup fee that we presented in the case study.

The second question was:

How can self-adaptation features be integrated into composite service infrastructures?

Similar to the first question that we revisited, this question has also several dimensions to it. In general, an approach for self-adaptation in composite services that improves performance and dependability was to be determined (1), along with a perspective on the right amount of coupling between the composite service and the adaptation system (2). Additionally, we had to find a method to make the features of corresponding supporting system available to non-technical users (3).

With regard to (1), we found that runtime service selection can improve the performance of composite services. To this end, our approach identifies individual services within a composite service that have performance problems or do not fulfill certain business related requirements, such as cost limits. Domain experts can assign several alternative services for each service that is part of a composite service. If such an alternative service performs better in terms of QoS or business related attributes, our system dynamically replaces the service that was originally defined in the composite service description with this alternative service. Thereby, we can immediately improve the performance and dependability of composite services, as well as minimize various business related properties, such as the costs that incur for the invocation of a service. Additionally, our system includes support for service interface mismatch compensation. This means that the interface descriptions of the service that was originally defined in the composite service description and the interface descriptions of its alternative services do not necessarily have to be syntactically identical. Truly, only semantic equivalence between the originally defined and its alternative services is required.

When considering (2), we leverage the basic abstraction that we also use for our holistic monitoring concept. That is, we reduce a composite service to its message exchange, to free our method from the intrinsics of arbitrary composition engines. Another upside of such an approach is that it supports various available composition platforms. Support for additional composition platforms can be realized with little effort, as we abstract from the implementation details of a composition engine by the means of an adapter. With regard to VieDAME, our proof of concept implementation of our approach, we provide a high degree of decoupling between the composition engine and VieDAME by delivering the adaptation features as a service. Truly, only a rather small component of our architecture, that is, the composition engine adapter, is required to be installed in the existing composition engine environment. This indeed improves the unobtrusiveness of our approach as a whole.

Finally, considering (3), the domain specific language that we incorporate into our service selection mechanisms allows users that are not skilled with programming language knowledge to leverage our system. We believe that this service selection language is expressive enough to cover enterprise requirements, yet easy enough to be used by non-technical users. The underlying QoS model enables a fine grained representation of business related QoS attributes. This is required in cases where the usage of a domain agnostic attribute, such as service cost, may be too coarse grained to describe service selection requirements. Moreover, the dynamic nature of both the underlying QoS model as well as the service selection language makes our system applicable in high-availability scenarios where system downtimes have to be minimized.

10.2 Possible Future Work

With regard to the scientific contributions that we presented in this thesis, there are several aspects that leave room for future research. We will briefly consider these aspects in what follows.

Visual Support for Monitoring Statement Creation. For the time being, domain experts that define monitoring statements have to provide the EPL statement that represents a monitoring requirement. To this end, our prototype user interface features an EPL editor that provides syntax high-lighting and basic auto-completion features. However, we strive for a visual support tool that enables a more comfortable and intuitive way to define monitoring requirements. In particular, such a tool should support a point and click approach to build monitoring statements that reference parts of a request message's payload. Currently, the XPath expressions that might be required in case no XML schema definition exists for a message payload must be user-provided. Thus, XPath know-how or the use of an external tool to derive XPath expressions is required.

Automatic Generation of Transformer Rules. This thesis presented the concept of transformer as a means to compensate the interface mismatch between a service originally defined in a composite service and its alternative services that may be used for a dynamic service replacement. Transformers currently rely upon XSLT to perform the required mediation on incoming and outgoing messages. However, the definition of such XSL documents is a rather tedious task. In certain scenarios, those definitions may be created automatically. We are currently working on an approach for automatic generation of XSL definitions based on term categorization techniques, similar to what Liang et al. present in [92].

Deployment Model. The deployment model of our proof of concept implementation is designed to provide a high degree of decoupling between the VieDAME system and the composition engine that is subject to monitoring and adaptation provided by our method. To this end, we presented an approach where a small component, that is, the composition engine adapter (CEA), is installed in an existing service composition environment. This CEA then provides monitoring information and requests information that is required for runtime service selection from a remotely installed VieDAME service instance. Nevertheless, in certain scenarios, VieDAME might be deployed on the same host where the composition engine in installed. In this case, the service based, inherent decoupling that we provide truly is not required and imposes an avoidable performance overhead. Hence, future work will evaluate various alternative deployment models that may improve the performance of VieDAME setups that are installed local to a composition engine.

Bibliography

[1] 3GPP. *GSM 03.03 - European digital cellular telecommunications system - Numbering, addressing and identification* , 2010. ftp://ftp.3gpp.org/specs/2000-12/Ph1/03_serie/0303-4a0.zip (Last accessed: Feb 25, 2010).

[2] 3GPP. *3GPP TS 11.11 Specification of the Subscriber Identity Module - Mobile Equipment (SIM-ME) Interface*, 2011. http://www.3gpp.org/ftp/specs/html-info/1111.htm (Last accessed: Jan 23, 2012).

[3] 3GPP. *3GPP TS 23.003 Numbering, addressing and identification*, 2011. http://www.3gpp.org/ftp/Specs/html-info/23003.htm (Last accessed: Jan 23, 2012).

[4] Active Endpoints. *ActiveBPEL Engine*, 2007. http://www.active-endpoints.com/ (Last accessed: May 07, 2007).

[5] Mohammad Alrifai and Thomas Risse. Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In *Proc. of the 18th International World Wide Web Conference (WWW'09), Madrid, Spain*, pages 881–890. ACM Press, April 2009.

[6] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. Selecting Skyline Services for QoS-based Web Service Composition. In *Proc. of the 19th International World Wide Web Conference (WWW'10), Raleigh, NC, USA*. ACM Press, April 2010.

[7] Ross Altman. SOA Overview and Guide to SOA Research, July 2010. Gartner Research Report (ID Number: G00201650).

[8] Vasilios Andrikopoulos, Salima Benbernou, and Mike P. Papazoglou. Evolving services from a contractual perspective. In *Proceedings of the 21st International Conference on Advanced Information Systems Engineering*, CAiSE '09, pages 290–304, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] Apache Foundation. *Apache CXF*, 2012. http://cxf.apache.org (Last accessed: Mar. 5 , 2012).

[10] Apache Foundation. *Apache Service Mix*, 2012. http://servicemix.apache.org (Last accessed: Mar. 5 , 2012).

[11] Apache Foundation. *Apache Tuscany*, 2012. `http://tuscany.apache.org` (Last accessed: Mar. 5 , 2012).

[12] Apache Software Foundation. *Apache ODE*, 2007. `http://ode.apache.org/` (Last accessed: Oct 21, 2007).

[13] Apache Software Foundation. *Apache Synapse*, 2007. `http://ws.apache.org/synapse/` (Last accessed: Oct 21, 2007).

[14] Apache Software Foundation. *Tomcat 7.0*, 2012. `http://http://tomcat.apache.org/tomcat-7.0-doc/index.html/` (Last accessed: Mar. 5 , 2012).

[15] Danilo Ardagna and Barbara Pernici. Adaptive Service Composition in Flexible Processes. *IEEE Trans. Softw. Eng.*, 33(6):369–384, 2007.

[16] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Gariapathy, and K. Holley. Soma: a method for developing service-oriented solutions. *IBM Syst. J.*, 47:377–396, July 2008.

[17] Natee Artaiam and Twittie Senivongse. Enhancing service-side qos monitoring for web services. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, ACIS International Conference on*, 0:765–770, 2008.

[18] L. Aversano, G. Canfora, A. Cimitile, and A. De Lucia. Migrating legacy systems to the web: an experience report. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 148 –157, 2001.

[19] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions Dependable Secure Computing*, 1(1):11–33, 2004.

[20] D. F. Bantz, C. Bisdikian, D. Challener, J. P. Karidis, S. Mastrianni, A. Mohindra, D. G. Shea, and M. Vanover. Autonomic personal computing. *IBM Systems Journal*, 42(1):165 –176, 2003.

[21] Fabio Barbon, Paolo Traverso, Marco Pistore, and Michele Trainotti. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In *IEEE Intl. Conf. on Web Services (ICWS'06)*, pages 63–71. IEEE Computer Society, 2006.

[22] L. Baresi and S. Guinea. Self-supervising bpel processes. *Software Engineering, IEEE Transactions on*, 37(2):247 –263, march-april 2011.

[23] Luciano Baresi and Sam Guinea. Dynamo: Dynamic Monitoring of WS-BPEL Processes. In *Proc. of the International Conference on Service-Oriented Computing (ICSOC'05), Amsterdam, The Netherlands*, pages 478–483. Springer, 2005.

[24] Luciano Baresi and Sam Guinea. A dynamic and reactive approach to the supervision of BPEL processes. In *ISEC '08: Proceedings of the 1st conference on India software engineering conference*, pages 39–48, New York, NY, USA, 2008. ACM.

[25] Luciano Baresi and Sam Guinea. Self-supervising bpel processes. *IEEE Trans. Software Eng.*, 37(2):247–263, 2011.

[26] 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 (ESSPE '07)*, pages 11–20. ACM, 2007.

[27] Luciano Baresi, Sam Guinea, and Pierluigi Plebani. Policies and Aspects for the Supervision of BPEL Processes. In *Proc. of the 19th International Conference on Advanced Information Systems Engineering (CAiSE'07), Trondheim, Norway*, pages 340–354. Springer, 2007.

[28] R. Barrett and P. P. Maglio. Intermediaries: An approach to manipulating information streams. *IBM Systems Journal*, 38(4):629 –641, 1999.

[29] Rob Barrett and Paul P. Maglio. Intermediaries: new places for producing and manipulating web content. In *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, pages 509–518, 1998.

[30] BBC. *Finland makes broadband a 'legal right'*, 2010. `http://www.bbc.co.uk/news/10461048` (Last accessed: Feb. 21, 2012).

[31] Christoph Becker, Hannes Kulovits, Michael Kraxner, Riccardo Gottardi, and Andreas Rauber. An extensible monitoring framework for measuring and evaluating tool performance in a service-oriented architecture. In *Proceedings of the 9th International Conference on Web Engineering*, ICWE '9, pages 221–235, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] Catriel Beeri, Anat Eyal, Tova Milo, and Alon Pilberg. Query-based monitoring of BPEL business processes. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD '07)*, pages 1122–1124. ACM, 2007.

[33] Catriel Beeri, Anat Eyal, Tova Milo, and Alon Pilberg. BP-Mon: query-based monitoring of BPEL business processes. *SIGMOD Rec.*, 37(1):21–24, 2008.

[34] Alex E. Bell. Death by uml fever. *Queue*, 2(1):72–80, March 2004.

[35] K. Bennett. Legacy systems: coping with stress. *Software, IEEE*, 12(1):19 –23, jan 1995.

[36] F.M.T. Brazier, J.O. Kephart, H. Van Dyke Parunak, and M.N. Huhns. Agents and service-oriented computing for autonomic computing: A research agenda. *Internet Computing, IEEE*, 13(3):82 –87, may-june 2009.

[37] Caucho Technology, Inc. *Hessian binary web service protocol*, 2011. `http://hessian.caucho.com/` (Last accessed: Feb. 1, 2012).

[38] K. Mani Chandy and W. Roy Schulte. *Event Processing: Designing IT Systems for Agile Companies*. McGraw-Hill Osborne Media, 2009.

[39] K.Mani Chandy and W.Roy Schulte. *Event Processing - Designing IT Systems for Agile Companies*. McGraw Hill Professional, 2010.

[40] Anis Charfi and Mira Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web*, 10(3):309–344, 2007.

[41] Lawrence Chung and Julio do Prado Leite. On non-functional requirements in software engineering. In Alexander Borgida, Vinay Chaudhri, Paolo Giorgini, and Eric Yu, editors, *Conceptual Modeling: Foundations and Applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379, 2009.

[42] Codehaus. *ExpandoMetaClass Domain Specific Language with Groovy*, 2009. `http://groovy.codehaus.org/ExpandoMetaClass+Domain-Specific+Language` (Last accessed: Oct 25, 2009).

[43] Codehaus. *Groovy*, 2009. `http://groovy.codehaus.org` (Last accessed: Oct 25, 2009).

[44] S. Comella-Dorda, K. Wallnau, R.C. Seacord, and J. Robert. A survey of black-box modernization approaches for information systems. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 173 –183, 2000.

[45] Carine Courbis and Anthony Finkelstein. Towards aspect weaving applications. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 69–77, New York, NY, USA, 2005. ACM.

[46] Umeshwar Dayal. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[47] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. A Taxonomy and Catalog of Runtime Software-Fault Monitoring Tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, December 2004.

[48] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15:313–341, 2008.

[49] The Eclipse Foundation. *AspectJ*, 2012. `http://www.eclipse.org/aspectj/` (Last accessed: Oct 21, 2011).

[50] ECMA International. *ECMA-376 Office Open XML File Formats*, 2011. `http://www.ecma-international.org/publications/standards/Ecma-376.htm` (Last accessed: Feb 2, 2012).

[51] M. Endrei, M. Gaon, J. Graham, K. Hogg, and N. Mulholland. Moving forward with web services backward compatibility, 2006. `http://www.ibm.com/developerworks/java/library/ws-soa-backcomp/index.html?ca=drs-` (Last accessed: Mar. 1, 2012).

[52] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology & Design*. Prentice Hall PTR, 2005.

[53] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall PTR, 2007.

[54] Thomas Erl, Anish Karmarkar, Priscilla Walmsley, Hugo Haas, L. Umit Yalcinalp, Kevin Liu, David Orchard, Andre Tost, and James Pasley. *Web Service Contract Design and Versioning for SOA*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2009.

[55] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17 –23, may/jun 2000.

[56] Abdelkarim Erradi, Piyush Maheshwari, and Vladimir Tosic. Policy-Driven Middleware for Self-Adaptation of Web Services Compositions. In *Proc. of the ACM/IFIP/USENIX International Conference on Middleware (Middleware'06), Melbourne, Australia*, pages 62–80, 2006.

[57] EsperTech. *Esper*, 2009. `http://esper.codehaus.org` (Last accessed: Oct 25, 2009).

[58] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[59] Europen Union 6th Framework Programme. *Semantics Utilised for Process Management within and between Enterprises (SUPER)*, 2012. `http://www.ip-super.org` (Last accessed: Mar. 5 , 2012).

[60] John Evdemon. Principles of service design: Service versioning, 2005. url http://msdn.microsoft.com/en-us/library/ms954726.aspx (Last accessed: Mar. 1, 2012).

[61] Eviware. *SoapUI*, 2009. `http://www.soapui.org/` (Last accessed: Oct 25, 2009).

[62] Onyeka Ezenwoye and S. Masoud Sadjadi. RobustBPEL2: Transparent Autonomization in Business Processes through Dynamic Proxies. In *Proc. of the 8th International Symposium on Autonomous Decentralized Systems (ISADS'07), Sedona, Arizona*, 2007.

[63] Ted Faison. *Event-Based Programming: Taking Events to the Limit*. Apress, 2006.

[64] Simon Gaele. Knowledge acquisition and modeling for corporate memory: Lessons learnt from experience. In *Proc. of KAW'96, Banff, Canada*, pages 1–18, November 1996.

[65] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissidesi. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[66] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5 –18, 2003.

[67] Google. *Protocol Buffers (Protobuf)*, 2011. http://code.google.com/p/protobuf/ (Last accessed: Feb. 1, 2012).

[68] Daniel Gredler. *Java Remoting: Protocol Benchmarks*, 2008. http://daniel.gredler.net/2008/01/07/java-remoting-protocol-benchmarks/ (Last accessed: Oct 21, 2011).

[69] GSM World. *IR.50.4.0 - 2G/2.5G/3G Roaming*, 2009. http://www.gsmworld.com/documents/IR50_4_0.pdf (Last accessed: Oct 25, 2009).

[70] Sam Guinea, Luciano Baresi, George Spanoudakis, and Olivier Nano. Comprehensive Monitoring of BPEL Processes. *IEEE Internet Computing*, (99):1, 2009.

[71] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[72] Paul Horn. Autonomic computing : Ibm's perspective on the state of information technology. *Computing Systems*, 15(Jan):1–40, 2001.

[73] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40:1–28, August 2008.

[74] C. Huemer, P. Liegl, R. Schuster, H. Werthner, and M. Zapletal. Inter-organizational systems: From business values over business processes to deployment. In *DEST 2008. 2nd IEEE Intl. Conf. on Digital Ecosystems and Technologies (DEST'08)*, pages 294 – 299, 2008.

[75] M.N. Huhns and M.P. Singh. Service-oriented computing: key concepts and principles. *Internet Computing, IEEE*, 9(1):75 – 81, jan-feb 2005.

[76] Ching-Lai Hwang and Kwangsun Yoon. *Multiple Attribute Decision Making. Methods and Applications.* Springer-Verlag GmbH, 1981.

[77] IBM Corporation. *An Architectural Blueprint for Autonomic Computing*, 2006. http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf (Last accessed: Oct. 21, 2009).

[78] IETF. *Introduction to Accounting Management*, 2000. http://tools.ietf.org/html/rfc2975 (Last accessed: Feb. 1, 2012).

[79] IETF. *The Transport Layer Security (TLS) Protocol Version 1.2*, 2008. http://tools.ietf.org/html/rfc5246 (Last accessed: Feb. 1, 2012).

[80] IETF. *Key words for use in RFCs to Indicate Requirement Levels*, 2012. http://www.ietf.org/rfc/rfc2119.txt (Last accessed: Mar. 5 , 2012).

[81] Intalio, Inc. *Intalio BPMS Designer*, 2011. http://www.intalio.com/bpms/designer.

[82] Intalio, Inc. *Intalio BPMS*, 2012. http://www.intalio.com/bpm (Last accessed: Mar. 5 , 2012).

[83] International Telecommunication Union (ITU). *Service-Oriented Architecture Ontology*, 2008. http://www.itu.int/rec/T-REC-E.800-200809-I/en (Last accessed: Feb. 1, 2012).

[84] Michael C. Jaeger, Gero Mühl, and Sebastian Golze. QoS-aware Composition of Web Services: An Evaluation of Selection Algorithms. In Robert Meersman and Zahir Tari, editors, *Proc. of the Confederated International Conferences CoopIS, DOA, and ODBASE 2005 (OTM'05), Agia Napa, Cyprus*, volume 3760 of *Lecture Notes in Computer Science (LNCS)*, pages 646–661. Springer, November 2005.

[85] Michael C. Jaeger, Gregor Rojec-Goldmann, and Gero Mühl. QoS Aggregation for Service Composition using Workflow Patterns. In *Proc. of the 8th International Enterprise Distributed Object Computing Conference (EDOC'04)*, pages 149–159. IEEE Computer Society, September 2004.

[86] S. Kallel, A. Charfi, T. Dinkelaker, M. Mezini, and M. Jmaiel. Specifying and Monitoring Temporal Properties in Web Services Compositions. In *Proc. of the 7th IEEE European Conf. on Web Services (ECOWS'09)*, pages 148–157, 2009.

[87] D. Karastoyanova and F. Leymann. Bpel'n'aspects: Adapting service orchestration logic. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 222 –229, july 2009.

[88] Doug Kaye. *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003.

[89] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41 – 50, January 2003.

[90] J.O. Kephart and W.E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3 – 12, june 2004.

[91] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[92] Qianhui Althea Liang and Herman Lam. Web service matching by ontology instance categorization. In *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 1*, SCC '08, pages 202–209, Washington, DC, USA, 2008. IEEE Computer Society.

[93] Yutu Liu, Anne H.H. Ngu, and Liangzhal Zeng. QoS Computation and Policing in Dynamic Web Service Selection. In *Proc. of the 13th International Conference on World Wide Web (WWW'04), New York, NY, USA*, pages 66–73. ACM Press, May 2004.

[94] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[95] Khaled Mahbub and George Spanoudakis. A framework for requirents monitoring of service based systems. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 84–93, New York, NY, USA, 2004. ACM.

[96] A. Mani and A Nagarajan. *Use SOAP-based intermediaries to build chains of Web service functionality*. IBM, 2002. `http://www.ibm.com/developerworks/webservices/library/ws-soapbase/` (Last accessed: Mar. 5 , 2012).

[97] Anbazhagan Mani and Arun Nagarajan. Understanding Quality of Service for Web Services, January 2002.

[98] Robert C. Martin. The dependency inversion principle. *C++ Report*, 1996.

[99] D.A. Menasce. Response-time analysis of composite web services. *Internet Computing, IEEE*, 8(1):90 – 92, 2004.

[100] Daniel A. Menasce. QoS issues in Web services. *IEEE Internet Computing*, 6(6):72–75, November/December 2002.

[101] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Comprehensive QoS Monitoring of Web Services and Event-Based SLA Violation Detection. In *Proc. of the 4th Intl. Workshop on Middleware for Service Oriented Computing (MWSOC'09)*, pages 1–6. ACM, 2009.

[102] 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*, 3:193–205, July-Sept. 2010.

[103] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards Recovering the Broken SOA Triangle - A Software Engineering Perspective. In *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering (IW-SOSE'07), co-located with the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07), Dubrovnik, Croatia.*, pages 22–28. ACM Press, September 2007.

[104] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken soa triangle: a software engineering perspective. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, IW-SOSWE '07, pages 22–28, New York, NY, USA, 2007. ACM.

[105] Microsoft Corporation. *The Business Value of Legacy Modernization*, 2007. http://download.microsoft.com/download/e/9/7/e9734f87-c581-482a-aaca-2835df48d40e/business_value_legacy_modernization.pdf (Last accessed: Feb. 1, 2012).

[106] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proceeding of the 17th International Conference on World Wide Web (WWW'08), Beijing, China*, pages 815–824. ACM, 2008.

[107] Adina Mosincat and Walter Binder. Transparent Runtime Adaptability for BPEL Processes. In *Proc. of the 6th International Conference Service-Oriented Computing (ICSOC'08), Sydney, Australia*, pages 241–255, 2008.

[108] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: a process-oriented approach. *Software Engineering, IEEE Transactions on*, 18(6):483 –497, jun 1992.

[109] OASIS. *OASIS Web Services Security (WSS) TC*, 2006. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss (Last accessed: Feb. 1, 2012).

[110] OASIS. *Web Service Business Process Execution Language 2.0*, 2006. URL: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel (Last accessed: Apr. 17, 2007).

[111] E. Oberortner, Uwe Zdun, and S. Dustdar. Patterns for measuring Performance-Related QoS properties in distributed systems. In *17th Conference on Pattern Languages of Programs*, October 2010.

[112] Object Management Group – Business Process Management Initiative. *Business Process Modeling Notation (BPMN) Specification, Version 1.0*, 2006. http://www.bpmn.org/ (Last accessed: Oct. 21, 2009).

[113] Object Management Group (OMG). *Common Object Request Broker Architecture (CORBA)*, 2011. http://www.omg.org/spec/CORBA/3.2/ (Last accessed: Feb. 1, 2012).

[114] Object Management Group (OMG). *Unified Modelling Language (UML)*, 2012. http://www.uml.org/ (Last accessed: Mar. 5 , 2012).

[115] Open Source Initiative. *Open Source Initiative*, 2012. http://www.opensource.org/ (Last accessed: Mar. 5 , 2012).

[116] OpenSymphony. *Quartz*, 2007. http://www.opensymphony.com/quartz/ (Last accessed: Oct 21, 2007).

[117] Oracle. *JConsole*, 2012. `http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html` (Last accessed: Mar. 5 , 2012).

[118] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[119] Justin O'Sullivan, David Edmond, and Arthur ter Hofstede. What's in a service? *Distributed and Parallel Databases*, 12:117–133, 2002. 10.1023/A:1016547000822.

[120] M. P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *Proc. of the Fourth International Conference on Web Information Systems Engineering*, pages 3–12, December 2003.

[121] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer*, 40(11):38 –45, 2007.

[122] Michael Papazoglou. *Web services & SOA: principles and technology*. Pearson, second edition, 2011.

[123] Mike P. Papazoglou. The challenges of service evolution. In *CAiSE*, pages 1–15, 2008.

[124] Mike P. Papazoglou, Vasilios Andrikopoulos, and Salima Benbernou. Managing evolving services. *IEEE Software*, 28(3):49–55, 2011.

[125] M.P. Papazoglou and J. Dubray. A survey of web service technologies. Technical Report DIT-04-058, University of Trento, 2004.

[126] M.P. Papazoglou and G. Georgakapoulos. Introduction to the Special Issue about Service-Oriented Computing. *Communicatios of the ACM*, 46(10):24–29, October 2003.

[127] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[128] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, March 1999.

[129] M. Perepletchikov and C. Ryan. A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software. *Software Engineering, IEEE Transactions on*, 37(4):449 –465, july-aug. 2011.

[130] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari. Coupling metrics for predicting maintainability in service-oriented designs. In *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*, pages 329 –340, april 2007.

[131] Panos Periorellis and Panos Periorellis. *Securing Web Services: Practical Usage of Standards and Specifications*. IGI Publishing, Hershey, PA, USA, 2007.

[132] The Huffington Post. *Finland: Broadband Access Made Legal Right In Landmark Law*, 2010. http://www.huffingtonpost.com/2009/10/14/finland-broadband-access_n_320481.html (Last accessed: Dec. 12, 2011).

[133] Postgres. *PostgreSQL*, 2012. http://www.postgresql.org/ (Last accessed: Oct 25, 2011).

[134] Maseud Rahgozar and Farhad Oroumchian. An effective strategy for legacy systems evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(5):325–344, 2003.

[135] Red Hat. *Hiberante ORM*, 2007. http://www.hibernate.org (Last accessed: Oct 24, 2007).

[136] Red Hat. *JBoss RiftSaw*, 2012. http://www.jboss.org/riftsaw (Last accessed: Mar. 5 , 2012).

[137] C. Roblee, V. Berk, and G. Cybenko. Implementing large-scale autonomic server monitoring using process query systems. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 123 –133, june 2005.

[138] F. Rosenberg, P. Celikovic, A. Michlmayr, P. Leitner, and S. Dustdar. An End-to-End Approach for QoS-Aware Service Composition. In *EDOC'09: Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference*, pages 128–137, 2009.

[139] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proc. of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, pages 205–212. IEEE Computer Society, 2006.

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

[141] SmartBear Software. *LoadUI*, 2012. http://www.loadui.org/ (Last accessed: Mar. 5 , 2012).

[142] SpringSource. *Spring Framework: Extensible XML authoring*, 2002. http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/extensible-xml.html (Last accessed: Mar. 5 , 2012).

[143] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115 –139, 1974.

[144] Mingjie Sun, Bixin Li, and Pengcheng Zhang. Monitoring BPEL-Based Web Service Composition Using AOP. In *Proc. of the 8th IEEE/ACIS Intl. Conf. on Computer and Information Science (ICIS'09)*, pages 1172–1177, 2009.

[145] M. Suntinger, J. Schiefer, H. Obweger, and M.E. Groller. The Event Tunnel: Interactive Visualization of Complex Event Streams for Business Process Pattern Analysis. In *IEEE Pacific Visualization Symposium (PacificVIS'08)*, pages 111–118, 2008.

[146] N. Thio and S. Karunasekera. Automatic measurement of a qos metric for web service recommendation. In *Software Engineering Conference, 2005. Proceedings. 2005 Australian*, pages 202 – 211, march-1 april 2005.

[147] Thomas Erl. www.soaprinciples.com, 2011. `http://www.soaprinciples.com/p3.php` (Last accessed: Dec. 12, 2011).

[148] VMWare. *Spring Framework*, 2012. `http://www.springframework.org` (Last accessed: Oct 24, 2011).

[149] Jonathan W. Strickland, Vincent W. Freeh, and Sudharshan S. Vazhkudai. Governor: Autonomic throttling for aggressive idle resource scavenging. In *Proceedings of the Second International Conference on Automatic Computing*, pages 64–75, Washington, DC, USA, 2005. IEEE Computer Society.

[150] W3C. *Web Services Architecture*, 2004. `http://www.w3.org/TR/ws-arch/#technology` (Last accessed: Feb. 1, 2012).

[151] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008. `http://www.w3.org/TR/REC-xml/` (Last accessed: Oct 21, 2009).

[152] W3C. *Voice Extensible Markup Language (VoiceXML) 3.0*, 2008. `http://www.w3.org/TR/voicexml30/)` (Last accessed: Feb 3, 2012).

[153] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[154] Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Ivona Brandic, Frank Leymann, and Schahram Dustdar. Monitoring and Analyzing Influential Factors of Business Process Performance. In *Proc. of the 13th IEEE Intl. Enterprise Distributed Object Computing Conf. (EDOC'09)*, pages 141–150. IEEE Computer Society, 2009.

[155] Narada Wickramage and Sanjiva Weerawarana. A benchmark for web service frameworks. In *Proceedings of the 2005 IEEE International Conference on Services Computing - Volume 01*, SCC '05, pages 233–242, Washington, DC, USA, 2005. IEEE Computer Society.

[156] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(02):115–152, 1995.

[157] World Wide Web Consortium (W3C). *SOAP Version 1.1*, 2000. URL: `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/` (Last accessed: Apr. 17, 2007).

[158] World Wide Web Consortium (W3C). *Web Services Description Language (WSDL) 1.1*, 2001. URL: http://www.w3.org/TR/wsdl (Last accessed: Apr. 17, 2007).

[159] World Wide Web Consortium (W3C). *SOAP Version 1.2*, 2003. URL: http://www.w3.org/TR/soap/ (Last accessed: Apr. 17, 2007).

[160] World Wide Web Consortium (W3C). *XML Schema Part 0: Primer Second Edition*, 2004. URL: http://www.w3.org/TR/xmlschema-0/ (Last accessed: Apr. 17, 2007).

[161] Guoquan Wu, Jun Wei, and Tao Huang. Flexible Pattern Monitoring for WS-BPEL through Stateful Aspect Extension. In *Proc. of the IEEE Intl. Conf. on Web Services (ICWS'08)*, pages 577–584, 2008.

[162] Jing Xu, Sumalatha Adabala, and J.A.B. Fortes. Towards autonomic virtual applications in the in-vigo system. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 15 –26, june 2005.

[163] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(6):1–26, 2007.

[164] Liangzhao Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *Software Engineering, IEEE Transactions on*, 30(5):311 – 327, may 2004.

[165] ZeroC. *Internet Communication Engine (Ice)*, 2011. http://www.zeroc.com (Last accessed: Feb. 1, 2012).

[166] Pengcheng Zhang, Bixin Li, Henry Muccini, and Mingjie Sun. Advanced web and networktechnologies, and applications. chapter An Approach to Monitor Scenario-Based Temporal Properties in Web Service Compositions, pages 144–154. Springer-Verlag, Berlin, Heidelberg, 2008.

[167] Zibin Zheng and Michael R. Lyu. A QoS-Aware Fault Tolerant Middleware for Dependable Service Composition. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks (DSN'09)*, pages 239 –248, July 2009.

APPENDIX A

List of Abbreviations

API	Application Programming Interface
AOP	Aspect Oriented Programming
AXIOM	AXis Object Model
BPM	Business Process Management
BPMN	Business Process Modelling Notation
CEP	Complex Event Processing
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DSL	Domain Specific Language
EAI	Enterprise Application Integration
EIS	Enterprise Information System
EJB	Enterprise Java Beans
ESB	Enterprise Service Bus
ESP	Event Stream Processing
EPL	Event Processing Language
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol

IT	Information Technology
JSON	JavaScript Object Notation
JAX-RPC	Java API for XML-based RPC
JAX-WS	Java API for XML-based Web Services
JEE	Java Enterprise Edition
JMS	Java Messaging Service
JRE	Java Runtime Environment
JSR	Java Specification Request
JVM	Java Virtual Machine
KPI	Key Performance Indicator
LAN	Local Area Network
MEP	Message Exchange Pattern
MOM	Message Oriented Middleware
OASIS	Organization for the Advancement of Structured Information Standards
OMG	Object Management Group
OOP	Object-Oriented Programming
PC	Personal Computer
QoS	Quality of Service
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SC	Sequencing Constraints
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service Oriented Computing
TCP	Transport Control Protocol
UDDI	Universal Description, Discovery and Integration

UML Unified Modelling Language

URI Universal Ressource Identifier

URL Universal Ressource Locator

VieDAME Vienna Dynamic Adaptation and Monitoring Environment

VUser Virtual User

WS-BPEL Web Service Business Process Execution Language

WS-CDL Web Service Choreography Description Language

WSDL Web Services Description Language

WWW World Wide Web

XML Extensible Markup Language

XSL Extensible Stylesheet Language

XSLT Extensible Stylesheet Language Transformations

APPENDIX B

Case Study Implementation

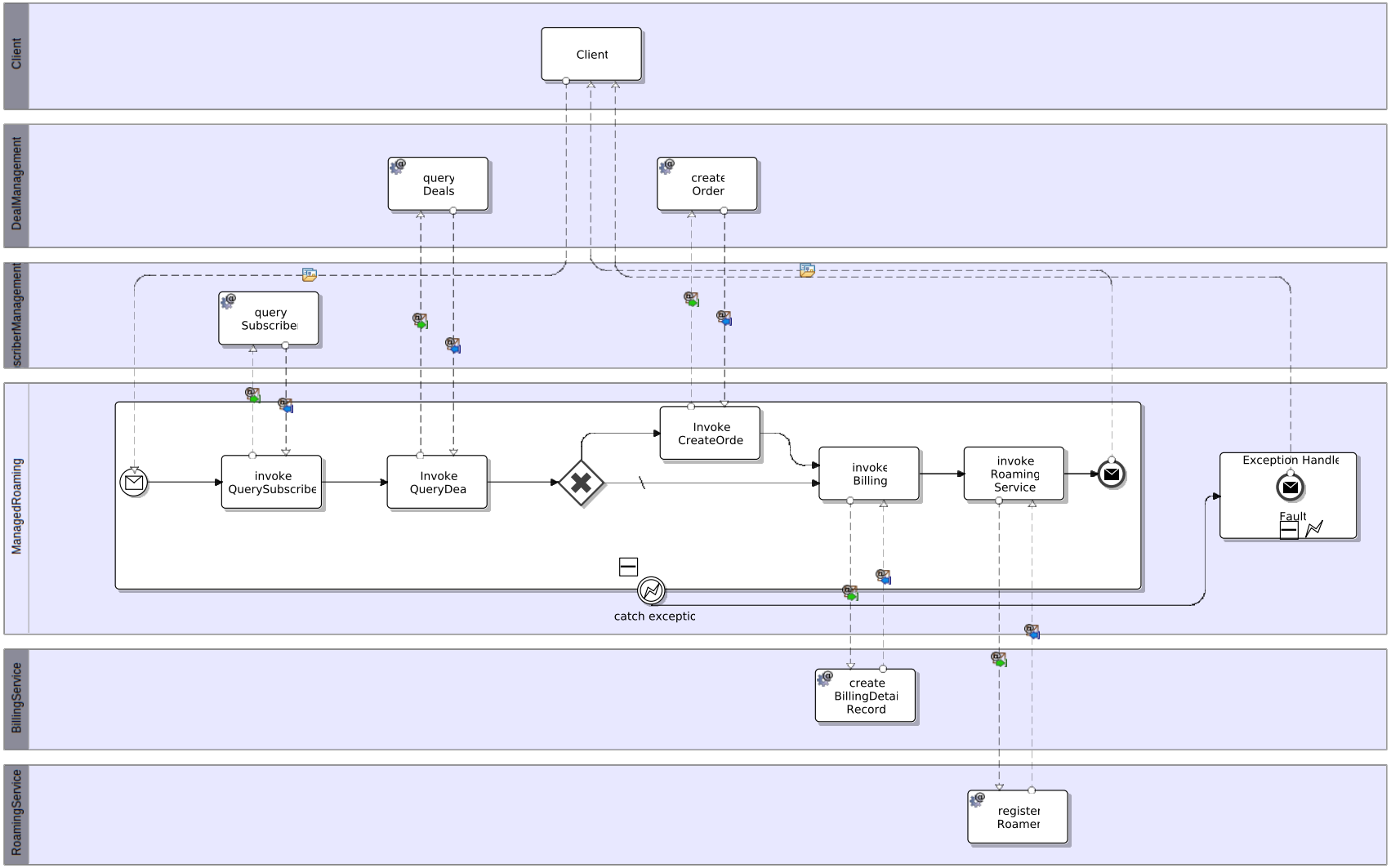


Figure 33: ManagedRoaming Composite Service Implementation

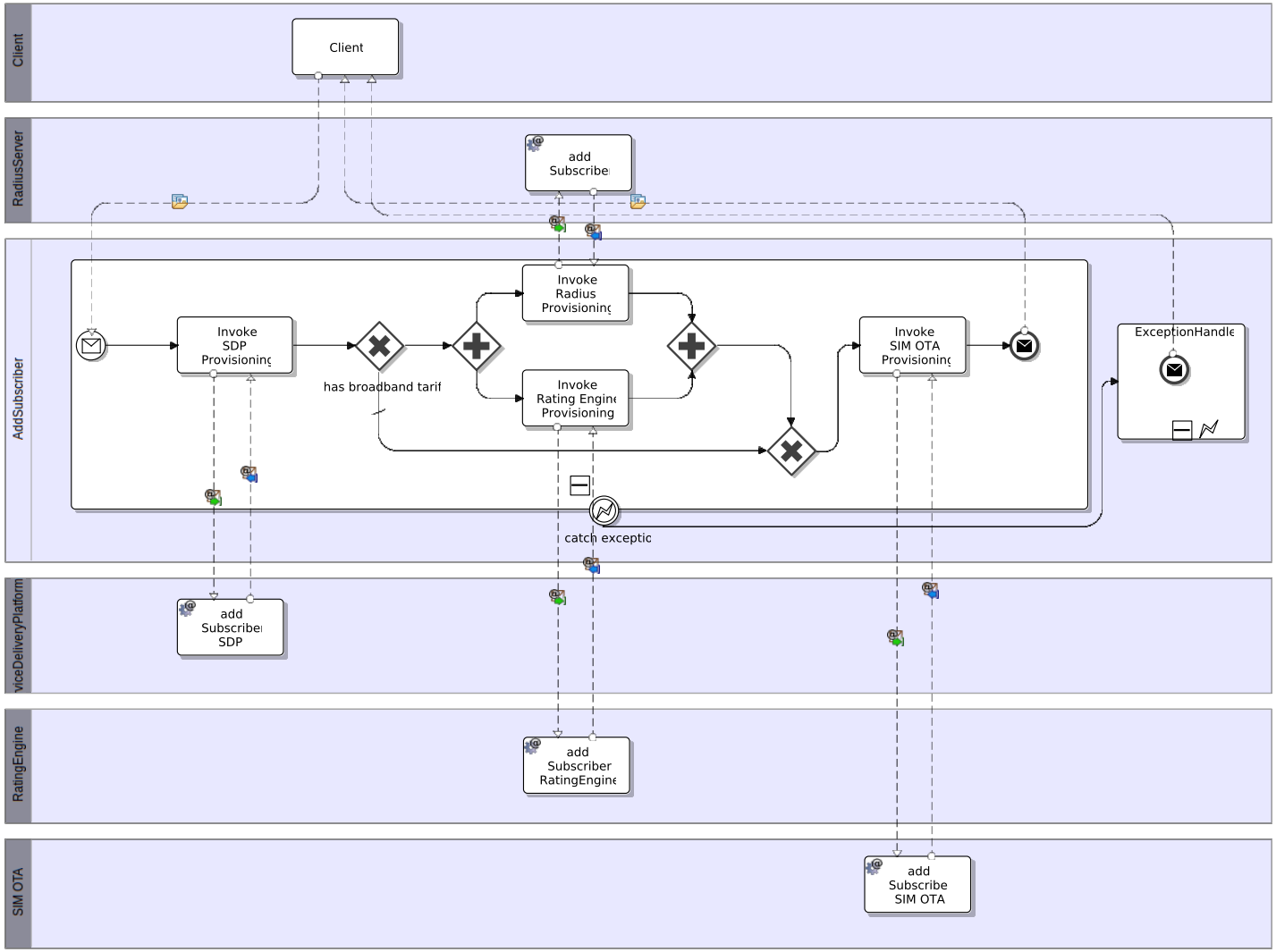


Figure 34: AddSubscriber Composite Service Implementation

```xml
<?xml version='1.0' encoding='utf-8'?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://vitalab.tuwien.ac.at/generatedService/billingService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://vitalab.tuwien.ac.at/generatedService/billingService"
name="billingServiceService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://vitalab.tuwien.ac.at/generatedService/billingService"
      schemaLocation="billingService.xsd" />
    </xsd:schema>
  </types>
  <message name="createBillingDetailRecord">
    <part name="parameters"
    element="tns:createBillingDetailRecord" />
  </message>
  <message name="createBillingDetailRecordResponse">
    <part name="parameters"
    element="tns:createBillingDetailRecordResponse" />
  </message>
  <message name="Exception">
    <part name="fault" element="tns:Exception" />
  </message>
  <portType name="billingService">
    <operation name="createBillingDetailRecord">
      <input message="tns:createBillingDetailRecord" />
      <output message="tns:createBillingDetailRecordResponse" />
      <fault message="tns:Exception" name="Exception" />
    </operation>
  </portType>
  <binding name="billingServicePortBinding"
  type="tns:billingService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
    <operation name="createBillingDetailRecord">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal" />
      </fault>
    </operation>
  </binding>
  <service name="billingServiceService">
    <port name="billingServicePort"
    binding="tns:billingServicePortBinding">
      <soap:address location="http://localhost:8060/WebServices/billingService" />
    </port>
  </service>
</definitions>
```

Listing 19: BillingService.wsdl

```
<?xml version='1.0' encoding='utf-8'?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://vitalab.tuwien.ac.at/generatedService/subscriberManagement"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://vitalab.tuwien.ac.at/generatedService/subscriberManagement"
name="subscriberManagementService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://vitalab.tuwien.ac.at/generatedService/subscriberManagement"
      schemaLocation="subscriberManagement.xsd" />
    </xsd:schema>
  </types>
  <message name="addSubscriber">
    <part name="parameters" element="tns:addSubscriber" />
  </message>
  <message name="addSubscriberResponse">
    <part name="parameters" element="tns:addSubscriberResponse" />
  </message>
  <message name="Exception">
    <part name="fault" element="tns:Exception" />
  </message>
  <message name="querySubscriber">
    <part name="parameters" element="tns:querySubscriber" />
  </message>
  <message name="querySubscriberResponse">
    <part name="parameters"
    element="tns:querySubscriberResponse" />
  </message>
  <portType name="subscriberManagement">
    <operation name="addSubscriber">
      <input message="tns:addSubscriber" />
      <output message="tns:addSubscriberResponse" />
      <fault message="tns:Exception" name="Exception" />
    </operation>
    <operation name="querySubscriber">
      <input message="tns:querySubscriber" />
      <output message="tns:querySubscriberResponse" />
      <fault message="tns:Exception" name="Exception" />
    </operation>
  </portType>
  <binding name="subscriberManagementPortBinding"
  type="tns:subscriberManagement">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
    <operation name="addSubscriber">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal" />
      </fault>
    </operation>
    <operation name="querySubscriber">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal" />
      </fault>
    </operation>
  </binding>
  <service name="subscriberManagementService">
    <port name="subscriberManagementPort"
    binding="tns:subscriberManagementPortBinding">
      <soap:address location="http://localhost:8060/WebServices/subscriberManagement" />
    </port>
  </service>
</definitions>
```

Listing 20: SubscriberManagement.wsdl

```xml
<?xml version='1.0' encoding='utf-8'?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://vitalab.tuwien.ac.at/generatedService/dealManagement"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://vitalab.tuwien.ac.at/generatedService/dealManagement"
name="dealManagementService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://vitalab.tuwien.ac.at/generatedService/dealManagement"
      schemaLocation="dealManagement.xsd" />
    </xsd:schema>
  </types>
  <message name="addOrder">
    <part name="parameters" element="tns:addOrder" />
  </message>
  <message name="addOrderResponse">
    <part name="parameters" element="tns:addOrderResponse" />
  </message>
  <message name="Exception">
    <part name="fault" element="tns:Exception" />
  </message>
  <message name="queryOrder">
    <part name="parameters" element="tns:queryOrder" />
  </message>
  <message name="queryOrderResponse">
    <part name="parameters" element="tns:queryOrderResponse" />
  </message>
  <portType name="dealManagement">
    <operation name="addOrder">
      <input message="tns:addOrder" />
      <output message="tns:addOrderResponse" />
      <fault message="tns:Exception" name="Exception" />
    </operation>
    <operation name="queryOrder">
      <input message="tns:queryOrder" />
      <output message="tns:queryOrderResponse" />
      <fault message="tns:Exception" name="Exception" />
    </operation>
  </portType>
  <binding name="dealManagementPortBinding"
  type="tns:dealManagement">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
    <operation name="addOrder">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal" />
      </fault>
    </operation>
    <operation name="queryOrder">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal" />
      </fault>
    </operation>
  </binding>
  <service name="dealManagementService">
    <port name="dealManagementPort"
    binding="tns:dealManagementPortBinding">
      <soap:address location="http://localhost:8060/WebServices/dealManagement" />
    </port>
  </service>
</definitions>
```

Listing 21: DealManagement.wsdl

```
<?xml version='1.0' encoding='utf-8'?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://vitalab.tuwien.ac.at/generatedService/radiusService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://vitalab.tuwien.ac.at/generatedService/radiusService"
name="radiusServiceService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://vitalab.tuwien.ac.at/generatedService/radiusService"
      schemaLocation="radiusService.xsd" />
    </xsd:schema>
  </types>
  <message name="addSubscriber">
    <part name="parameters" element="tns:addSubscriber" />
  </message>
  <message name="addSubscriberResponse">
    <part name="parameters" element="tns:addSubscriberResponse" />
  </message>
  <message name="Exception">
    <part name="fault" element="tns:Exception" />
  </message>
  <portType name="radiusService">
    <operation name="addSubscriber">
      <input message="tns:addSubscriber" />
      <output message="tns:addSubscriberResponse" />
      <fault message="tns:Exception" name="Exception" />
    </operation>
  </portType>
  <binding name="radiusServicePortBinding"
  type="tns:radiusService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
    <operation name="addSubscriber">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal" />
      </fault>
    </operation>
  </binding>
  <service name="radiusServiceService">
    <port name="radiusServicePort"
    binding="tns:radiusServicePortBinding">
      <soap:address location="http://localhost:8060/WebServices/radiusService" />
    </port>
  </service>
</definitions>
```

Listing 22: RadiusService.wsdl

```xml
<?xml version='1.0' encoding='utf-8'?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://vitalab.tuwien.ac.at/generatedService/ratingService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://vitalab.tuwien.ac.at/generatedService/ratingService"
name="ratingServiceService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://vitalab.tuwien.ac.at/generatedService/ratingService"
      schemaLocation="ratingService.xsd" />
    </xsd:schema>
  </types>
  <message name="addSubscriber">
    <part name="parameters" element="tns:addSubscriber" />
  </message>
  <message name="addSubscriberResponse">
    <part name="parameters" element="tns:addSubscriberResponse" />
  </message>
  <message name="Exception">
    <part name="fault" element="tns:Exception" />
  </message>
  <portType name="ratingService">
    <operation name="addSubscriber">
      <input message="tns:addSubscriber" />
      <output message="tns:addSubscriberResponse" />
      <fault message="tns:Exception" name="Exception" />
    </operation>
  </portType>
  <binding name="ratingServicePortBinding"
  type="tns:ratingService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
    <operation name="addSubscriber">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal" />
      </fault>
    </operation>
  </binding>
  <service name="ratingServiceService">
    <port name="ratingServicePort"
    binding="tns:ratingServicePortBinding">
      <soap:address location="http://localhost:8060/WebServices/ratingService" />
    </port>
  </service>
</definitions>
```

Listing 23: RatingService.wsdl

```
<?xml version='1.0' encoding='utf-8'?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://vitalab.tuwien.ac.at/generatedService/simOtaService"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://vitalab.tuwien.ac.at/generatedService/simOtaService"
name="simOtaServiceService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://vitalab.tuwien.ac.at/generatedService/simOtaService"
      schemaLocation="simOtaService.xsd" />
    </xsd:schema>
  </types>
  <message name="addSubscriber">
    <part name="parameters" element="tns:addSubscriber" />
  </message>
  <message name="addSubscriberResponse">
    <part name="parameters" element="tns:addSubscriberResponse" />
  </message>
  <message name="Exception">
    <part name="fault" element="tns:Exception" />
  </message>
  <portType name="simOtaService">
    <operation name="addSubscriber">
      <input message="tns:addSubscriber" />
      <output message="tns:addSubscriberResponse" />
      <fault message="tns:Exception" name="Exception" />
    </operation>
  </portType>
  <binding name="simOtaServicePortBinding"
  type="tns:simOtaService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
    <operation name="addSubscriber">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal" />
      </fault>
    </operation>
  </binding>
  <service name="simOtaServiceService">
    <port name="simOtaServicePort"
    binding="tns:simOtaServicePortBinding">
      <soap:address location="http://localhost:8060/WebServices/simOtaService" />
    </port>
  </service>
</definitions>
```

Listing 24: SimOta.wsdl

```
<?xml version='1.0' encoding='utf-8'?>
<definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://vitalab.tuwien.ac.at/generatedService/serviceDeliveryPlatform"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://vitalab.tuwien.ac.at/generatedService/serviceDeliveryPlatform"
name="serviceDeliveryPlatformService">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://vitalab.tuwien.ac.at/generatedService/serviceDeliveryPlatform"
      schemaLocation="serviceDelivery.xsd" />
    </xsd:schema>
  </types>
  <message name="addSubscriber">
    <part name="parameters" element="tns:addSubscriber" />
  </message>
  <message name="addSubscriberResponse">
    <part name="parameters" element="tns:addSubscriberResponse" />
  </message>
  <message name="Exception">
    <part name="fault" element="tns:Exception" />
  </message>
  <portType name="serviceDeliveryPlatform">
    <operation name="addSubscriber">
      <input message="tns:addSubscriber" />
      <output message="tns:addSubscriberResponse" />
      <fault message="tns:Exception" name="Exception" />
    </operation>
  </portType>
  <binding name="serviceDeliveryPlatformPortBinding"
  type="tns:serviceDeliveryPlatform">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
    <operation name="addSubscriber">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
      <fault name="Exception">
        <soap:fault name="Exception" use="literal" />
      </fault>
    </operation>
  </binding>
  <service name="serviceDeliveryPlatformService">
    <port name="serviceDeliveryPlatformPort"
    binding="tns:serviceDeliveryPlatformPortBinding">
      <soap:address location="http://localhost:8060/WebServices/serviceDeliveryPlatform" />
    </port>
  </service>
</definitions>
```

Listing 25: ServiceDelivery.wsdl

APPENDIX C

Curriculum Vitae