DISSERTATION

QoS-Aware Composition of Adaptive Service-Oriented Systems

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

unter der Leitung von

Univ.-Prof. Dr. Schahram Dustdar
Distributed Systems Group
Institut für Informationssysteme (E184)
Technische Universität Wien

und

Prof. M. Brian Blake, PhD
Department of Computer Science
University of Notre Dame, IN, USA

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Florian Rosenberg
Matr.Nr.: 9955548
Hippgasse 5/13
A-1160 Vienna

Wien, Mai 2009

Abstract

Service-Oriented Computing (SOC) increasingly gains momentum in academia and industry as a means to develop adaptive distributed software applications in a loosely coupled way. Software services, as the main entities in SOC, have some distinct properties such as platform-independence or a uniform interface description enabling an easier integration and use within and across organizational boundaries. One of the main assets of service-orientation is composability to develop higher-level services, so-called composite services, by re-using well-known functionality provided by other services in a low-cost and rapid development process. However, in distributed environments, the use of services without any quality guarantees from the service providers can negatively affect a composite service by raising intermittent failures or having a slow performance of one of the services. One of the main problems is the lack of an integrated Quality of Service (QoS) model combined with an automated monitoring technique. The availability of accurate and up-to-date QoS information enables a QoS-aware composition and optimization of composite services by automatically selecting well-performing services and dynamically replace services that reduce the performance or lead to failures in a composition. However, existing QoS-aware composition approaches mainly focus on the optimization aspect to find the best composition in terms of QoS. Therefore, QoS should be seamlessly integrated into multiple layers of the SOC stack, such as choreography, orchestration and execution. This enables an end-to-end view on QoS and allows a better integration and optimization throughout the application lifecycle to achieve the vision of adaptive service-oriented systems.

This thesis contributes a set of methods and tools to address these issues. Firstly, it proposes an extensible multi-layer QoS model for services and an automated QoS monitoring approach. Secondly, it describes the integration of Service Level Agreements (SLAs) into choreographies and proposes an automated mapping to orchestrations annotated with QoS policies to enable SLA enforcement. Thirdly, it addresses a set of issues related to the overall development lifecycle of QoS-aware service composition. Specifically, a domain-specific language, called VCL, is introduced to enable the specification of QoS-aware composite services with a focus on hard and soft constraints in form of constraint hierarchies. Based on VCL, a set of methods and algorithms are presented to generate an executable composite service that is optimized with regard to the QoS constraints specified by the user. To this end, this thesis also introduces a novel Web service runtime environment, called VRESCo, which implements a number of important SOC concepts (such as dynamic binding or invocation) that are foundational for the presented QoS-aware service composition approach.

Kurzfassung

Service-Oriented Computing (SOC) gilt als aufstrebende Disziplin in der Forschung sowie Industrie und befasst sich mit den Möglichkeiten lose-gekoppelte, verteilte und adaptive Softwaresysteme zu entwickeln. Software Services bilden die Kernbausteine im SOC und zeichnen sich u.a. durch ihre Plattformunabhängigkeit und einer einheitlichen Schnittstelle aus. Dadurch wird eine einfachere Integration und Benutzung über Unternehmensgrenzen hinweg ermöglicht. Eine der wichtigsten Eigenschaften ist die Komponierbarkeit (composability) um funktional höherwertige Services durch Wiederverwendung existierender Services einfach und rasch zu implementieren. In verteilten Szenarien besteht allerdings das Problem, dass Serviceanbieter keine Qualitätsgarantien in Form von Dienstgüteattributen – im folgenden auch Quality of Service oder abgekürzt QoS genannt – zur Verfügung stellen (wie Antwortzeit, Verfügbarkeit, usw.). Dies kann vor allem in Kompositionen, die verschiedenste Services diverser Anbieter verwenden, zum Problem werden, da die Performance und Verfügbarkeit der einzelnen Services auch die gesamte Komposition gefährden kann. Ein Hauptproblem ist das Fehlen eines umfassenden QoS Modells und Techniken zur Messung verschiedener QoS Attribute. Die Verfügbarkeit akkurater QoS Information ermöglicht eine QoS-getriebene und optimierte Komposition durch automatisches Selektieren und Ersetzen von Services. Existierende Ansätze fokussieren hauptsächlich auf die Optimierung von Kompositionen hinsichtlich deren QoS. Es ist aber umso wichtiger die ganzheitliche Sicht und Integration von QoS Eigenschaften über die verschiedenen Ebenen wie Choreographie, Komposition und Ausführung eines Service-orientierten Systems zu betrachten. Dies bietet den Vorteil einer End-to-End Sicht auf QoS und ermöglicht eine bessere Integration und Optimierung über den ganzen Applikationslebenszyklus um die Vision von adaptiven Service-orientierten Systemen zu verwirklichen.

Diese Arbeit leistet einen Beitrag zur Lösung der oben genannten Probleme. Erstens wird ein erweiterbares QoS-Modell sowie ein Ansatz und das dazugehörige Werkzeug vorgestellt, um diese laufzeit-spezifischen QoS-Attribute automatisch messen zu können. Zweitens wird ein Verfahren vorgestellt wie Service Level Agreements (SLAs) in Choreographien eingebunden werden können. Diese werden in der Folge automatisch auf Kompositionen für die verschiedenen Partner abgebildet. Drittens wird ein Verfahren präsentiert, dass die Spezifikation von QoS-getriebenen Kompositionen verbessert. Dazu wird eine domain-spezifische Sprache VCL (Vienna Composition Language) beschrieben, die es erlaubt den gewünschten QoS einer Komposition in Form von harten und weichen Constraints zu spezifizieren und somit eine weitaus flexiblere Form der Spezifikation zu ermöglichen. Aufbauend darauf werden Verfahren und Algorithmen präsentiert um auf Basis von VCL ausführbare und QoS-optimierte Kompositionen zu erstellen die den Constraints in der Spezifikation entsprechen. Um diese Ausführung zu realisieren wird eine neuartige Web Service Laufzeitumgebung, genannt VRESCo, vorgestellt, die typische SOC Konzepte implementiert und damit die Grundlage für die QoS-getriebene Komposition darstellt.

Acknowledgments

A PhD thesis is never the result of a single person's effort and it would not have been possible without the support of many different people. First and foremost, I would like to thank my adviser Prof. Schahram Dustdar for giving me the chance to do a PhD thesis in his Distributed Systems Group (DSG). He created an environment that allowed me to work on new ideas and he kept me on track at the right time to turn them into reality. Additionally, I would also like to thank Prof. M. Brian Blake from the University of Notre Dame for being my second adviser and examiner.

I would also like to thank my colleagues in our group, especially the core members of the VRESCo team, Philipp Leitner and Anton Michlmayr. It was a great time working together with you on various papers, hacks and other things. I also had the honor to work with extraordinary masters students: Predrag Celikovic, Christian Enzi, Andreas Huber, Thomas Laner and Oliver Moser contributed a lot to this thesis while working on joint papers or prototypes with me. I'm also very grateful for the great time I had during my two stays at the IBM T.J. Watson Research Center in New York. The Component Systems Group, especially Francisco Curbera, Matthew Duftler and Rania Khalaf, provided a great environment and were a main source of inspiration for many things we did there. Additionally, I would like to thank our DSG secretaries for keeping all the administrative stuff away from us.

My deepest and sincere thanks goes to my parents Franz and Christine who supported me emotionally and financially throughout my whole life. My sister Verena for her support and the fun weekends we could enjoy together by keeping me away from work. Last but not least, I would like to thank the most important person in my life, my beloved partner Andrea, for her indescribable support during all the years we have been together. All things we have experienced in different places of the world so far will never be unforgotten. This thesis is dedicated to you!

Florian Rosenberg
Vienna, Austria, May 2009

Für Andrea und meine Familie

Publications

Parts of the work presented in this dissertation have been published in the following conference, journal and workshop papers. A full list of papers published by the author of this thesis can be found at the end of this dissertation.

- F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, pages 205–212. IEEE Computer Society, Sept. 2006. `doi:10.1109/ICWS.2006.39`.
- C. Platzer, F. Rosenberg, and S. Dustdar. *Securing Web Services: Practical Usage of Standards and Specifications*, chapter Enhancing Web Service Discovery and Monitoring with Quality of Service Information. Idea Group Inc. (IGI), Nov. 2007.
- A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar. Towards Recovering the Broken SOA Triangle – A Software Engineering Perspective. In *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering (IW-SOSWE'07), Dubrovnik, Croatia*, pages 22–28. ACM Press, 2007. `doi:10.1145/1294928.1294934`.
- F. Rosenberg, C. Enzi, A. Michlmayr, C. Platzer, and S. Dustdar. Integrating Quality of Service Aspects in Top-Down Business Process Development using WS-CDL and WS-BPEL. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC'07), Annapolis, Maryland, USA.*, pages 15–26. IEEE Computer Society, Oct. 2007. `doi:10.1109/EDOC.2007.23`.
- O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Adaptation for WS-BPEL. In *Proceedings of the 17th International International World Wide Web Conference (WWW'08), Beijing, China*, pages 815–824. ACM Press, Apr. 2008. `doi:10.1145/1367497.1367607`.
- O. Moser, F. Rosenberg, and S. Dustdar. VieDAME – Flexible and Robust BPEL Processes through Monitoring and Adaptation (Informal Demo Paper). In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany*, pages 917–918. ACM Press, May 2008. `doi:10.1145/1370175.1370186`.
- F. Rosenberg, P. Leitner, A. Michlmayr, and S. Dustdar. Integrated Metadata Support for Web Service Runtimes. In *Proceedings of the Middleware for Web Services Workshop (MWS'08), co-located with the 12th IEEE International Distributed Object Computing*

Conference (EDOC'08), Munich, Germany. IEEE Computer Society, Sept. 2008. `doi:10.1109/EDOCW.2008.38`.

- F. Rosenberg, A. Michlmayr, and S. Dustdar. Top-Down Business Process Development and Execution using Quality of Service Aspects. *Enterprise Information Systems*, pages 459–475, November 2008. `doi:10.1080/17517570802395626`.

- F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar. Towards Composition as a Service - A Quality of Service Driven Approach. In *Proceedings of the First IEEE Workshop on Information and Software as Services (WISS'09), co-located with the 25th International Conference on Data Engineering (ICDE'09), Shanghai, China*. IEEE Computer Society, Mar. 2009. `doi:10.1109/ICDE.2009.153`.

- F. Rosenberg, P. Celikovic, A. Michlmayr, P. Leitner, and S. Dustdar. An End-to-End Approach for QoS-Aware Service Composition. In *Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference (EDOC'09), Auckland, New Zealand*, 2009.

- A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-End Support for QoS-Aware Service Selection, Invocation and Mediation in VRESCo. Technical Report TUV-184-2009-03, Technical University Vienna, June 2009. Available from: `http://www.infosys.tuwien.ac.at/Staff/rosenberg/papers/TUV-1841-2009-03.pdf`.

Contents

List of Figures

List of Tables

Listings

Abbreviations

AI	Artificial Intelligence
AOP	Aspect-Oriented Programming
CaaS	Composition as a Service
COP	Constraint Optimization Problem
CSP	Constraint Satisfaction Problem
DAG	Directed Acyclic Graph
DAL	Data Access Layer
DAO	Data Access Objects
DSL	Domain-Specific Language
IETF	Internet Engineering Task Force
IP	Integer Programming
ITU	International Telecommunication Union
JSON	JavaScript Object Notation
MCDM	Multiple Criteria Decision Making
MCOP	Multi-Constrained Optimal Path
MDA	Model-Driven Architecture
MIP	Mixed Integer Programming
MMKP	Multidimensional Multi-Choice Knapsack Problem
QoS	Quality of Service
RCSP	Resource Constrained Project Scheduling Problem
RFC	Request for Comments
SLO	Service Level Objective
SMS	Short Message Service
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service-Oriented Computing
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
UDDI	Universal Description Discovery and Integration
VCL	Vienna Composition Language
VMF	VRESCo Mapping Framework
VPN	Virtual Private Network
VQL	VRESCo Query Language
VRESCo	Vienna Runtime Environment for Service-Oriented Computing

WS-BPEL	Web Service Business Process Execution Language
WS-CDL	Web Service Choreography Description Language
WS-QDL	Web Service Quality Definition Language
WSDL	Web Service Description Language
WSLA	Web Service Level Agreements
WSOL	Web Service Offering Language
XAML	eXtensible Application Markup Language
XML	eXtensible Markup Language

Chapter 1

Introduction

It's supposed to be hard! If it wasn't hard, everyone would do it. The hard. . . is what makes it great!" - Jimmy Dugan

Contents

1.1 Motivation

Over the last years, the increasing distribution of software systems has led to an enormous rise in application complexity. This increase in complexity has a multitude of reasons, among them, the enormous need to integrate and connect heterogeneous applications and resources within and across organizational boundaries. However, most legacy systems and applications were not designed to be integrated and adapted to new application scenarios, therefore, requiring new paradigms and approaches to cope with these challenges.

The concept of service-orientation as a design paradigm provides the necessary conceptual foundations to deal with the increasing complexity and integration challenges by promoting the development of autonomous and loosely coupled software entities called services. A service is usually characterized by several distinct properties such as loose coupling, well-defined service contracts as well as the fact that services are based on standards and are independent of any particular implementation technology [103, 166].

In general, Service-Oriented Computing (SOC) is seen as an emerging discipline promoting science, research, and technology related to services [44, 117]. The overall motivation behind SOC is the idea that businesses offer their application functionality as services over the Internet and other companies or users can integrate and compose these business services into their applications. This concept is manifested in an architectural style that is commonly referred to as Service-Oriented Architecture (SOA). In Figure 1.1, the core stakeholders of an SOA are depicted: a service provider publishes a specific service contract which describes a service in a service registry ("Register"). A service requester (the client) can then query a service from the service registry ("Find") and dynamically bind to one of the services that were returned by the search query. This SOA triangle can be implemented, for example, by using existing technologies from the Web services stack [166]. These include SOAP (formerly known as Simple Object Access Protocol) [158] as a transport protocol, WSDL (Web Service Description Language) [157] as a service description language to specify service contracts and UDDI (Universal Description Discovery and Integration) [106] as a registry for storing services and its metadata. It has to be noted that Web services are only one technology for implementing an SOA. However, the concepts presented in this thesis can also be applied to other SOA technologies.

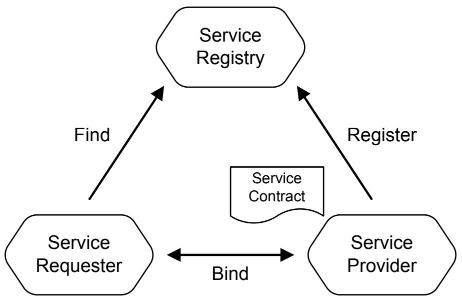


Figure 1.1: SOA Triangle

One of the core principles of service-orientation is the idea of composing these network-available services by discovering and dynamically invoking them rather than building applications from scratch or reusing other applications [44]. The process of building service-oriented applications from existing services is known as *service composition* or *orchestration*, the result of the composition process is called a *composite service* [45]. WS-BPEL (Web Service Business Process Execution Language) or BPEL for short [107] is the de-facto standard orchestration language in the Web service area. In contrast to orchestration, the concept of *choreography* describes the message interchanges between participants in service-oriented systems and provides a global model of all the participants and their message exchanges without requiring a central coordinator [16,172]. An orchestration specifies the executable behavior of each participant in the choreography for example by using BPEL. Despite some critique [15], the Web Service Choreography Description Language (WS-CDL) is one of the first examples for describing the global model of service interactions [161]. Both choreography and orchestration underlie the general area of service-oriented software engineering and thus represent different design choices when implementing service-oriented systems. The concrete scenario determines whether an explicit top-down approach to choreography and derived partner orchestrations is required and useful or a bottom-up approach is sufficient.

Irrespective of the design and modeling approach, a core requirement for service-oriented systems is a certain degree of self-adaptivity [42, 104]. According to Cheng et al. [34], "a self-adaptive system is able to modify its behavior according to changes in its environment". A key enabler for realizing adaptive behavior for service-oriented systems in general, and service compositions in particular, is the availability of Quality of Service (QoS) data. The term QoS has its origin in the networking community where it is defined by Crawley et al. [36] as "a set of service requirements to be met by the network while transporting a flow" (where a flow represents a stream of IP packets from source to destination). In the SOC community, QoS comprises all non-functional attributes of a service, ranging from performance-specific attributes to security and cost-related data. In general, QoS can be grouped into deterministic and non-deterministic attributes [81]. Deterministic QoS attributes, on the one hand, indicate that their value is known before a service is invoked, including price or the supported security protocols. On the other hand, their non-deterministic counterpart includes all attributes that are uncertain at service invocation time, for example the service response time. Therefore, the availability of accurate non-deterministic QoS information plays a crucial role during development and execution of a composite application. Firstly, QoS enables a QoS-aware dynamic binding to concrete services that are available in registries known at runtime. Secondly, QoS enables an optimization of composite services in terms of its overall QoS and adaptation of services whenever QoS changes. We denote a composite service leveraging QoS to enable adaptive behavior as *QoS-aware composite service* and the engineering process as *QoS-aware service composition*.

In general, this thesis addresses different facets within the lifecycle of developing and optimizing QoS-aware composite applications in SOA environments. In the following, we focus specifically on research issues addressed in this thesis and provide a coherent framework to illustrate and interlink these problems and their contributions.

1.2 Problem Definition

When developing QoS-aware service-oriented applications, several distinctive software layers can be used to master the development complexity and provide a logical application structure. A common architectural approach for implementing service-oriented systems is the use of a layered architecture [48] as shown in Figure 1.2.

The *choreography layer* defines an abstraction where all participants in a service-oriented system agree on the publicly observable behavior in terms of messages that are exchanged among partners in a business process. Typically, such a choreography layer is most adequate when using a top-down approach for developing a system because it defines a common agreement among the participants.

The *orchestration layer* focuses on the internal behavior of each participant in the choreography that is required to realize and implement a business process. Following Zdun et al. [173, 174], we distinguish between two different types of processes: a *macroflow* represents higher-level business processes and *microflows* address the process flow within macroflow process activities. This conceptual distinction is important for two reasons. Firstly, it enables the design of business processes at the right level of granularity. Macroflows capture long-running processes whereas microflows implement short running processes more on a technical level (composite services). Secondly, this distinction helps to separate business problems from the technical/application space.

The *service layer* comprises all atomic services that are available for the upper layers to integrate them in compositions or use them as part of the choreography description. These services can be, among others, public services, corporate services or simple wrapper services for legacy systems. These services are managed by the execution layer.

The *execution layer* combines all aspects related to the SOA triangle functionalities such as publish-find-bind, the dynamic invocation of services and the execution of composite services and business processes. Additionally, QoS monitoring techniques are implemented in the execution layer to incorporate up-to-date QoS information of all deployed services.

1.2.1 Key Research Issues

However, following such a layered design approach does not per se imply that these applications provide QoS-awareness and adaptive behavior. An example of such adaptive behavior could be an automatic replacement of services in a composition with low accuracy. Current approaches and runtimes have a lack of flexibility to realize such adaptive behavior across the full services lifecycle. This has a multitude of reasons and requires a number of research issues to be addressed.

QoS Integration and Monitoring. A major concern when implementing flexible service-oriented systems is the availability of QoS information for atomic services. QoS issues have received a lot of attention over the last years (e.g., [87, 88, 176]), however, no coherent and extensible model for addressing and integrating QoS on various layers exists (choreography, orchestration, and service layer).

In terms of modeling service-oriented systems, a number of approaches exist, such as choreography modeling leveraging different languages [40, 41, 171]. However, non of these approaches considers QoS aspects from the beginning of the modeling phase. An integration of QoS in early design phases of the choreography can reduce the burden of an integration at a later stage in the development process.

Besides the ability to specify QoS, it is of utmost importance that QoS attributes can be monitored continuously by using non-intrusive monitoring mechanisms (i.e., no need to know any service internals, the service interface description should be sufficient). In general, the availability of a QoS model, the integration of QoS into modeling approaches and automated monitoring capabilities are crucial aspects to increase adaptivity of service-oriented systems.

QoS-Aware Composition and Execution. On the orchestration layer, several existing workflow or composition languages such as WS-BPEL [107] or the Microsoft Windows Workflow Foundation [96] can be used. However, these approaches are purely static and once the composition is fully specified and deployed, no adaptive behavior is available allowing a composition to change at runtime (e.g., dynamically select another service because the QoS of an existing one is decreasing). These "static" languages do not support a flexible specification and execution of QoS-aware composite services. Most existing works on QoS-aware composition focus purely on the optimization part of the problem which is known to be NP-hard [4, 9, 26, 59, 65, 66, 170, 175, 176]. Moreover, these approaches only allow the specification of hard QoS constraints in a composition (globally or for a specific service) such that, in many cases, solutions cannot be determined that fulfill all QoS requirements. In general, the problem of QoS-aware service composition comprises a much broader range of necessary tasks to solve the problem from an end-to-end perspective.

Integrated Service Runtime Environment. Successfully addressing the aforementioned issues requires a strong runtime support. The main problem with current runtimes is the fact that they do not fully implement the SOA triangle from Figure 1.1. In theory, this triangle is designed to enable adaptivity for applications implementing this model. In practice, however, this model is mostly implemented in a point-to-point way where the requester interacts with the provider without dynamically discovering and binding to a service. This simplified model is mainly used due to a lack of flexible SOA runtimes that provide support for the original SOA triangle as part of the middleware or runtime environment [93]. Therefore, a Web service runtime is required to provide native and integrated support for service-oriented software engineering. This includes issues such as publishing and managing services, dynamic binding, and service discovery. An integrated end-to-end environment is required to enable QoS-aware composition and adaptation.

1.2.2 Research Questions

The aforementioned problems raise the need for a set of methods and a framework to effectively develop QoS-aware service-oriented systems. In particular, this thesis is guided by the following two main research questions including a set of subquestions to further structure the main questions.

Q1: *What is a method and supporting system for managing QoS issues in service-oriented systems that operate across the full services lifecycle (i.e., modeling, development, and runtime)?*

- Which QoS aspects have to be considered to address different lifecycle phases?
- How can QoS be effectively monitored to support the full services lifecycle?
- Which methods facilitate modeling support for QoS in service-oriented systems?

Q2: *Does a QoS-aware language and supporting runtime environment effectively facilitate composition and adaptation of QoS-aware service-oriented systems?*

- Which methods provide a flexible and effective way to specify QoS-aware compositions?
- What kind of runtime support mechanisms are required to address composition and adaptation in QoS-aware systems?

1.3 Contributions

In regard to the aforementioned problems, the contributions of this thesis are summarized by presenting a research and integration architecture to interrelate the contributions to manifest the "big picture" of this thesis. The architecture is depicted in Figure 1.2 and comprises three different layers. We have annotated the architecture to associate the specific contributions to the corresponding layers and use curly braces to define their scope.

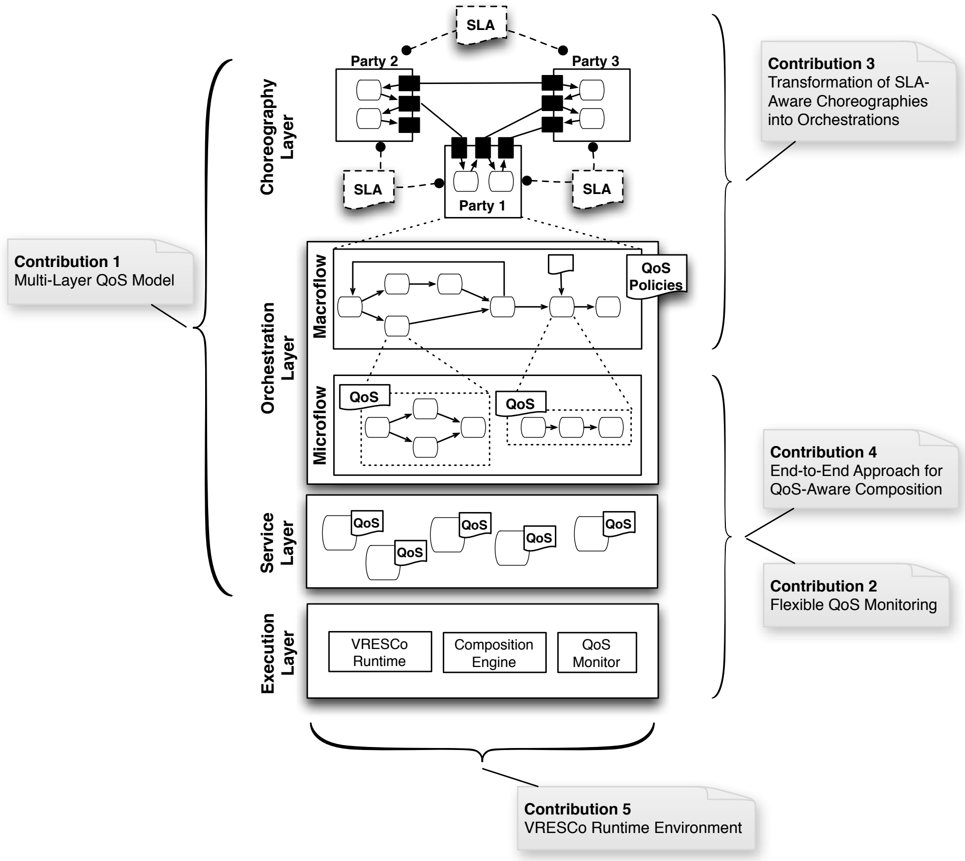


Figure 1.2: The Big Picture

Multi-Layer QoS Model. The first contribution of this thesis is a QoS model comprising multiple layers from the SOC stack. On the service layer, we consider elementary QoS attributes such as the response time of a service or the supported security protocol. On the orchestration layer, QoS policies can be specified to define QoS guarantees for various partner invocations in an orchestration. Additionally, QoS attributes from the service layer are aggregated to calculate the QoS of a composite service. On the choreography layer, QoS is expressed on a higher level in form of Service Level Agreements (SLAs) between two partners. These SLAs are defined by combining multiple fine-grained QoS attributes into Service Level Objectives (SLOs) that have to be guaranteed by the service provider. The presented QoS model has been successfully used for several approaches published in [124, 131, 134, 136].

Flexible QoS Monitoring. Alongside with the aforementioned QoS model, this thesis contributes a novel client-side QoS monitoring approach for Web services called QUATSCH. It allows to monitor performance-specific QoS attributes such as response time, latency or throughput continuously from a client-side perspective without requiring access to service provider internals. The approach combines dynamic invocation of services and aspect-oriented programming (AOP) techniques [72, 73] with low-level TCP packet capturing and analysis to calculate the server-side execution time of a service operation. Since a service is treated as a black-box, the QoS monitor can be used as a suitable third-party tool for QoS monitoring. The QUATSCH approach has been published in [124, 136] and successfully integrated in other systems, such as VRESCO (Vienna Runtime Environment for Service-Oriented Computing), which is described later in this thesis. A different approach for real-time monitoring of QoS attributes within the context of business processes, was designed and implemented as non-intrusive monitoring solution for a WS-BPEL engine by leveraging AOP techniques [99, 100].

Transformation of SLA-Aware Choreographies into Orchestrations. In order to support the development of QoS-aware service-oriented systems, we present a top-down modeling approach by considering non-functional aspects (QoS) from the beginning of the modeling phase as a first-class citizen. In this regard, we use WS-CDL as a choreography description language and leverage SLAs (specified in WSLA [61]) as the main technique for capturing the outcome of the QoS negotiations between different service providers in the global model. This global model is used to automatically generate the necessary orchestration stubs for each partner in the choreography in WS-BPEL notation. These orchestrations are automatically annotated with QoS attributes in form of QoS policies by using WS-Policy [165]. Furthermore, we leverage VieDAME [99, 100] as an execution engine for enacting the orchestrations and enforcing QoS policies. This contribution has been published in [131, 134].

End-to-End Approach for QoS-Aware Composition. Besides modeling the global view on the choreography layer, this thesis puts a strong emphasis on the composition layer. Firstly, we contribute a domain-specific language (DSL) called VCL (Vienna Composition Language) allowing a constraint-based specification of functional and non-functional requirements that each service in the composition has to fulfill. We put a particular focus on the specification of global and local QoS constraints (hard and soft constraints). The former specify constraints for the overall composition whereas the latter specify constraints for a single service in the composition. The proposed QoS constraint specification follows the theory of constraint hierarchies [23] to allow a fine-grained distinction of the importance of a QoS constraint and to reduce the risk of specifying an over-constrained composition (i.e., a composition with existing contradictory constraints in the problem space). Secondly, we present a Composition as a Service (CAAS) approach based on VCL allowing a semi-automated generation, execution and deployment of QoS-aware composite services. The approach combines a set of techniques, such as data flow analysis, a constraint optimization approach, Integer Programming (IP) and the generation of a executable composition. These approaches have been published in [129, 132].

VRESCo Runtime Environment. We contribute VRESCO (Vienna Runtime Environment for Service-Oriented Computing), a novel runtime and programming model based on an extensible service metadata model [133]. It addresses typical software engineering related issues in SOC, such as publishing services, dynamic binding and invocation [93], service mediation, and service discovery by using a type-safe query mechanism. The VRESCO approach comprises many aspects within the SOC stack, however, we can only briefly describe the overall VRESCO system and focus on the service metadata model, the query language and the mediation framework. The QoS-aware composition approach described above, uses the VRESCO runtime, especially the service metadata model as a foundation. Additionally, the above mentioned QoS monitoring approach has been integrated to enable QoS-awareness of services published in VRESCO. The VRESCO work relevant for this thesis has been published in [91, 93, 133].

It is important to note that most proof-of-concept implementations described in this thesis use technologies from the Web services stack as a means to implement an SOA. However, most concepts would also apply to other technologies that can be used to realize service-oriented systems. For example, our composition approach is focused around SOAP-based Web services and uses the Windows Workflow Foundation as an execution platform. The same QoS-aware composition concepts could be applied to RESTful services [120] and the Bite composition environment [130], co-developed as part of two internships at IBM Research, or BPEL for REST [118].

Please also note that all relevant publications co-authored by the author of this thesis are referenced in this chapter and are explicitly listed in the Publications section (page vii). This thesis mainly summarizes these publications and in the remainder of this thesis they are used without being referenced individually.

1.4 Organization of the Thesis

This thesis is organized as follows: Chapter 2 presents the related work classified into the core areas described earlier as part of the contributions. We compare the contributions described in this thesis with existing work in the respective areas. Then, the thesis is split in two major parts:

Part I comprises the contributions related to the integration of QoS into service-oriented systems. Chapter 3 describes the multi-layer QoS model that forms the basis for all further approaches described in this thesis. Chapter 4 describes the QoS monitoring approach which is later used and integrated into the VRESCo environment. Chapter 5 presents the transformation of SLA-aware choreographies into orchestrations using WS-CDL and WS-BPEL.

Part II comprises the contributions related to QoS-aware service composition and execution. Chapter 6 introduces the VRESCo runtime and discusses the service metadata model in detail as it forms the basis for the QoS-aware composition approach. Chapter 7 presents the domain-specific language called VCL, which is used as the main language for specifying QoS-aware compositions. Chapter 8 builds upon VCL and describes the QoS-aware composition approach to achieve "Composition as a Service" (CaaS) as the final contribution.

Finally, Chapter 9 concludes this thesis and outlines some future work in this area.

Chapter 2

Related Work

This section presents the related work according to the main areas aligned with the contributions of this thesis: QoS models, QoS monitoring, choreography modeling and transformation, and service composition in general with a particular focus on QoS-aware service composition.

Contents

2.1 Quality of Service Models

In general, there is no formal definition for QoS, however, several definitions exist in the telecommunications domain where QoS is mainly used to define certain communication level properties of networks (such as throughput or error rate). One of the first definitions appeared in 1995 in the International Telecommunication Union (ITU) standard X.902 [62] where they define QoS as follows: "A set of quality requirements on the collective behavior of one or more objects". Several QoS attributes describe the speed and reliability of data transmission, e.g., error rate, transit delay or throughput.

Numerous other QoS definitions have emerged especially in the networking community, in particular related to ATM (Asynchronous Transfer Mode) networks. These networks are able to provide QoS guarantees on a transport-level such as bit rate, delay or jitter. The Internet Engineering Task Force (IETF) addresses these definitions in various RFCs related to ATM, e.g., RFC 1932 [35] and RFC 1946 [63].

In the context of multimedia, Vogel et al. [155] define QoS with a particular focus on the application-level using real-time communication: "The set of those quantitative and qualitative characteristics of a distributed multimedia system, which are necessary in order to achieve the required functionality of an application".

The importance of QoS in the area of service-oriented systems has been initially discussed in early 2000 by several researchers [38, 83, 87, 125] by leveraging the knowledge from earlier networking-related QoS attributes. Additionally, application- and business related QoS attributes have been addressed to cope with the need for providing quality attributes of loosely coupled distributed services. Up to now, QoS issues in SOC have received a lot of attention, in particular, QoS-aware service selection and composition have been core areas of research.

Ran [125] was one of the first who has proposed a QoS model and a UDDI extension for associating QoS information to specific Web services. The model comprises QoS categories, such as runtime-related QoS, transactional QoS attributes and several other categories. QoS information of a Web service is included when publishing a service in the UDDI registry. Unfortunately, Ran does not describe whether QoS information can be updated once it is published, therefore, leading to static QoS information that quickly becomes obsolete. Additionally, that paper does not specify how runtime-related QoS attributes are calculated or monitored.

Tian et al. [144, 145] have defined a QoS model for Web services representing a combination of XML schema and ontologies. They distinguish between server QoS (e.g., processing time or availability), transport QoS (e.g., delay, jitter, throughput), security and transactions. Extensibility is supported by using an extension element in XML schema. Each QoS attribute in the aforementioned categories has a reference to an ontology where custom metrics can be defined.

In the workflow domain, Cardoso et al. [29] present a QoS model for workflows and Web processes that mainly consists of task cost, task time and task reliability. Besides the QoS model, the authors also present a stochastic workflow reduction algorithm to aggregate atomic QoS attributes of each task in the workflow to calculate the overall workflow QoS.

In 2005, OASIS established a technical committee [105] for defining a *Web Services Quality Model*, thus, highlighting the importance of QoS for Web services in general. The ambition of the committee is to develop a quality model for Web services and in the long run a WS Quality Definition Language (WS-QDL) to describe quality aspects of Web services by using an XML-based language. The current committee draft defines several quality categories. For example, the category *Service Level Measurement Quality* defines attributes such as response time or throughput whereas the category *Security Quality* defines all security-related QoS aspects. Currently, their work is still a draft and it remains unclear how the model can be implemented and used with real Web services.

Ontologies for representing Web service QoS have also been proposed. The approach presented in [116] describes an ontology which captures QoS attributes, their metrics, the relationship to other attributes and several other aspects. Unfortunately, the authors do not present any details whether their QoS ontology is integrated into existing registries or any other technology allowing to retrieve the actual QoS values. The QoS ontology presented in [178] is similar to the one presented in [116]. Additionally, the authors present a matchmaking approach allowing to determine all compatible QoS descriptions for a given one.

Common to all the aforementioned approaches is the fact that QoS is seen only from one perspective, mostly from an atomic service point of view. The model that we propose in this thesis considers multiple viewpoints on QoS. Therefore, we enable the use of QoS on different layers from the SOC stack such as the service, orchestration and choreography layer. Additionally, most approaches do not indicate how QoS attributes are measured and updated to avoid having outdated QoS values associated with certain services. We address these issues by proposing an integrated QoS framework as part of our VRESCo runtime environment and additionally use a QoS monitor to accurately measure performance-specific attributes.

2.2 QoS Monitoring

An accurate QoS monitoring approach for atomic services and service compositions is important for numerous reasons. Firstly, it enables to assess, rank and select service providers based on certain QoS attributes. Secondly, it opens a wide range of new application scenarios and possibilities for improving the stability and adaptability of distributed service-oriented systems.

An overview of QoS monitoring approaches for Web service based systems is presented by Thio et al. [143]. The authors discuss possible techniques such as low-level sniffing or a proxy-based solution to perform the monitoring. The prototype system presented in this paper adopts an approach where the SOAP engine library is modified in the sense that the code is instrumented with logging statements to emit the necessary information for QoS measurement. A major drawback of this approach is the dependency on the modified SOAP library and the resulting maintenance and distribution of the modified library.

Mani and Nagarajan [83] present a set of QoS attributes and discuss possible bottlenecks that might influence the QoS. They also outline a monitoring approach for response time by manually instrumenting the generated Web service proxy to perform the measurement. The drawback with this approach is the manual instrumentation of generated Web service client stubs which makes it impractical for automated monitoring and for systems that use dynamic (and stub-less) Web service invocations [80].

Wickramage and Weerawarana [167] elaborate how SOAP requests are typically processed by modern Web service frameworks such as Apache CXF [6]. They define 15 distinguishable time periods a SOAP request goes through before completing a round trip. Our monitoring approach utilizes this knowledge, however, we do not make use of all 15 time periods because not all of them are of interest to service consumers and can be determined from a client-side perspective.

Song and Lee [139] propose a simulation based Web service performance analysis tool called sPAC. It allows to analyze the performance of Web processes (i.e., a composition) by using simulation code. Their approach invokes a Web service once under low load conditions and then transforms these testing results into a simulation model. Our work focuses also on the performance aspects of Web services whereas we do not use simulation code, we perform our evaluation on real Web services with the advantage that we do not require access to the Web service implementation.

Zeng et al. [177] present a model-driven QoS monitoring approach for observable QoS metrics. An observable QoS metric is one that can be computed based on monitoring operational service events. Their approach is integrated into their SOA environment and provides declarative service QoS monitoring by leveraging ECA (Event-Condition-Action) rules to define for which service what kind of QoS should be monitored. In contrast to our work, Zeng et al. extend a full-blown SOA infrastructure to perform the monitoring, whereas we provide a monitoring tool that can be used to monitor QoS from a third-party perspective. However, besides IT-level QoS attributes, their approach also supports monitoring of business-related QoS.

Barbon et al. [11] describe a monitoring approach for WS-BPEL processes enabling runtime checking of various domain-specific situations of interest (e.g., the number of rejected credit card payments). Additionally, their approach also supports statistical analysis of properties and timing-specific information. Fei et al. [52] present a distributed framework for QoS monitoring based on their QoS management system called Q-Peer. The monitoring is based on policies (using WS-Policy [165]) to specify what metric should be monitored, either applied to atomic services or compositions. The authors use SOAP intermediaries to apply their policies by using dedicated monitoring peers. Contrary to our monitoring approach, these two approaches focus on monitoring of business-relevant QoS, whereas our approach is designed to monitor performance and dependability related QoS attributes.

Jurca et al. [69, 70] present a reputation-based approach for monitoring Web service QoS. The authors argue that most existing monitoring approaches based on message interception, provider-based monitoring or client-side probing have their drawbacks. These include scalability problems, lack of trustworthiness and they are too error-prone. Therefore, their approach uses client feedback as a reputation mechanism. The feedback is collected using a trusted center which then aggregates the feedback to generate QoS reports. In contrast to our approach, the authors have quite high requirements on the infrastructure and the service provider, e.g., the provisioning and negotiation of SLAs with consumers, a UDDI repository, and an SLA repository. In fact, this functionality is hardly available, therefore, we use a client-side approach based on probing without any requirement except a public access to the WSDL file from the monitoring host.

Baresi et al. [12–14] present a general framework called Dynamo for monitoring composite services. They do not solely focus on monitoring QoS, however, they focus on a general assertion-based language, called *WSCoL*, which is based on the WS-Policy framework [165] to define monitoring assertions. It allows to define the functionality and the required QoS as pre- and postconditions on partner invocations in a composition. An example assertion could be a check whether a given return value of a BPEL invoke is within a given range. Compared to Dynamo, our monitoring solution focuses solely on QoS by treating a service as a black-box.

Thus, there is no need to put QoS monitoring assertions on the composite service specification since we have a pre-defined QoS model defining the attributes which are monitored.

Al-Masri and Mahmoud [2] investigated how QoS can improve Web service discovery and ranking to deliver better results and generate a higher user experience. They use a client-side probing technique to determine the QoS of a Web service (e.g., availability, response time, etc.). Based on the probed QoS data, a client can then use a QoS-based discovery to retrieve the best service that matches their criteria. Compared to our monitoring approach, their work does not allow to calculate the service-side execution time, therefore, using response time as a discovery criteria can lead to unwanted results. For example, if the probing entity has a slow network connection, it can distort the results. Additionally, Al-Masri and Mahmoud presented an experimental study investigating the availability of Web services on the World Wide Web [3]. They applied well-known crawling techniques to search for WSDL files in registries and search engines. Moreover, they performed a series of tests such as validation of the WSDL interface, measurement and probing of QoS attributes to determine if the service is usable.

Truong et al. [148] present a QoS classification, measurement and monitoring approach for dependent grid services. Grid services are computational resources requiring different methods for monitoring various grid resources. These include machines, network paths, middleware and applications which have different requirements on the measurement methods. In contrast to our approach, their paper focuses on different grid services and resources, therefore, different monitoring techniques of arbitrary complexity need to be combined (e.g., ping vs. analyzing log files).

2.3 Choreography Modeling and Transformation

Integrating SLA and QoS aspects in top-down development of service-oriented systems, especially choreographies, has not yet received much attention whereas modeling of choreographies is subject to various research activities (e.g., [41,171]). We mainly discuss existing choreography modeling and transformation approaches as well as extensions of current Web service standards to include QoS attributes and SLAs and the integration of policies in BPEL.

Mendling and Hafner [89] define mapping rules for deriving BPEL processes from a WS-CDL choreography description. For each WS-CDL ordering structure and activity the corresponding BPEL construct is determined. These mapping rules define the basis for the mapping rules used throughout the top-down modeling process in Chapter 5. The mapping of WS-CDL to BPEL is referenced in detail, whereas the generation of WSDL interfaces used in the BPEL process is not addressed explicitly. In contrast to the work presented in this thesis, no explicit endpoint projection rules are defined to determine which ordering structures are relevant for the participants in the choreography. Finally, we also define mapping rules for the generation of WSDL descriptions which correspond to the service interface descriptions of the derived BPEL processes.

Dyaz et al. [46] use an intermediary model for the generation of BPEL processes from a WS-CDL choreography description focusing on Web services where time constraints play a critical role. A choreography description is first transformed into a timed automaton model which is verified and validated for correctness using formal model checking techniques. This model is then further used to generate BPEL processes. In contrast to our work, their focus is laid on the generation and verification of the timed automaton model. Detailed mapping rules for the derivation of BPEL processes from this model are not specified. In the context of top-down modeling, it seems more appropriate to perform a direct mapping between WS-CDL and BPEL instead of using an intermediary model.

Pi4soa [122] is a toolset from π4 Technologies and one of the first WS-CDL implementations. They provide a Eclipse-based designer tool which we used for modeling our choreographies, and a possibility to generate Java services from a WS-CDL document. In contrast to pi4soa, our work considers QoS from the beginning of the development. It might be interesting to include the SLA/QoS related aspects into the pi4soa Eclipse plugin to enable a combined modeling of choreographies and SLA between the partners in the choreography.

Decker et al. [40,41] propose a new extension to BPEL, called BPEL4Chor that allows modeling of choreographies within BPEL by leveraging an interconnected interface behavior model, whereas WS-CDL represents an interaction model. As stated in [41], it has not been investigated yet which of these two approaches is more appropriate for human modelers. While we follow a top-down approach by transforming WS-CDL into BPEL, the authors propose a bottom-up approach by introducing a new choreography layer on top of BPEL. However, in contrast to our work, the integration of QoS into choreographies is not addressed.

In the area of SLAs, Dan et al. [39] describe the WSLA (Web Service Level Agreement) framework [55] to formally define electronic contracts between business partners. WSLA provides the language and the necessary foundations for capturing different SLA parameters and their metrics to define guarantees and obligations among the parties in the agreement. We leverage WSLA and associate these SLAs with the corresponding partners on the choreography layer.

Another SLA framework is WS-Agreement [55], a specification from the Open Grid Forum. It can be seen as the evolution of WSLA by defining a language and a protocol for establishing agreements and interfaces to monitor agreement compliance at runtime. One of the advantages of WS-Agreement is its extensibility by allowing the specification of domain-dependent metrics and business values such as cost or penalties. WS-Agreement offers a basic negotiation mechanism by simply allowing an offer to be accepted or rejected.

Unger et al. [149] present a formal model and mechanism to aggregate SLAs in business processes. The authors attach SLAs to WS-BPEL partner links to associate them with the corresponding partner Web services within the process. Then the aggregation algorithm computes an overall SLA for the process. Contrary to our work, Unger et al. attach SLA directly to BPEL processes, whereas, we use SLAs on the choreography layer and decompose them into QoS policies that are then directly attached to the partner links in a WS-BPEL process. This allows the process engine to enforce these policies and trigger necessary actions in case of degrading QoS.

2.4 Service Composition Approaches

A vast number of service composition approaches have been proposed in literature. In this section, we review selected works based on their relevance for our approach. In particular, we focus on QoS-aware composition approaches, DSLs in the context of SOA and composition as well as some selected papers that take a different approach to service composition.

2.4.1 QoS-Aware Composition and Optimization

In this section we present a broad overview of existing work in the area of QoS-aware optimization. Almost all presented approaches share the goal to find an optimized composition with respect to several QoS constraints (e.g., minimizing the overall response time while maximizing the availability). A major difference between the QoS-aware composition approach presented in this thesis and all works highlighted below is the fact that our approach explicitly enables the user to specify hard and soft QoS constraints in form of constraint hierarchies to specify the importance of a constraint. In existing approaches, QoS is typically considered as hard constraint which may lead to over-constrained systems [17]. As a consequence, it is difficult to satisfy dynamic composition requests from users if the service QoS values do not match (e.g., required response time of 1500 msec but a service has 1502 msec).

To the best of our knowledge, Guan et al. [57] are the first who proposed a framework for QoS-guided service composition using constraint hierarchies as a formalism for specifying QoS. Their idea of modeling functional requirements as hard constraints and using constraint hierarchies to model QoS has some commonalities with the work presented in this thesis, however, the authors only support a small set of QoS attributes and their approach has very limited support for well-known composition constructs. They use a branch and bound algorithm that is only capable of solving sequential compositions, whereas, our approach supports various composition constructs (AND, XOR split, loops, sequences, etc). Additionally, the authors do not elaborate on the performance of their approach.

Zeng et al. [175,176] present a QoS-aware composition approach based on state diagrams to model a composition. A composition is split into multiple execution paths, each considered to be a directed acyclic graph (DAG). For local optimization they use Multiple Criteria Decision Making (MCDM) to choose a service which fulfills all requirements and has the highest score. Global optimization is achieved by using a naive global planning approach (high runtime complexity) and an Integer Programming (IP) solution. Additionally, the authors describe an approach to re-plan and re-optimize a composition based on the fact that QoS can change over time. Therefore, a composition is split into regions according to the state of the tasks that allow a re-planning by adding constraints of what has already been accomplished to optimize services that still have to be executed. In our work, we do not require a user to fully specify a composition as a state diagram, however, we provide VCL to specify the functional and QoS constraints, and the business protocol. In line with their approach, we also use IP to solve the optimization problem, however, we use a different approach to aggregate QoS (since we support different control-flow composition constructs) and to model the IP problem due to the fact that we use a constraint hierarchy based approach.

Canfora et al. [26] propose an approach to solve the QoS-aware composition problem by applying genetic algorithms. The genome represents the composition problem by using an integer array where the number of items equals the number of distinct abstract services. Each item, in turn, contains an index to the array of the concrete services matching that abstract service. The crossover operator is a standard two-point crossover, while the mutation operator randomly selects an abstract service (position in the genome) and randomly replaces the corresponding concrete service with another one from the pool of available concrete services. The selection problem is modeled as a dynamic fitness function with the goal to optimize the QoS attributes. Additionally, the fitness function must penalize individuals that do not meet the QoS constraints. The approach is evaluated by comparing it to well-known integer programming techniques. The authors also describe an approach that allows re-planning of existing service compositions based on slicing [27]. In contrast to their work, we model the problem as constraint optimization problem as well as an IP problem with support for QoS constraints hierarchies (hard and soft constraints).

Ardagna and Pernici [8, 9] propose a QoS-aware optimization approach using dynamic service selection that is based on the MAIS architecture [121]. The composition leverages WS-BPEL where each service in the process can be subject to global and local constraints which are fulfilled at runtime through adaptive re-optimization. The authors apply loop peeling techniques to optimize loop iterations and negotiation techniques to find a feasible solution to the optimization problem. The authors claim that they solve the optimization problem, in particular the fulfillment of global constraints, under more stringent conditions. Similar to their approach, we also allow to specify local and global constraints that are optimized, however, we propose a simple DSL to specify these constraints. This gives the developer a greater flexibility in specifying the dynamic QoS-aware composite service.

An efficient global optimization approach for QoS-aware service composition supporting global constraints on a composition level is proposed by Alrifai and Risse [4]. In contrast to other existing approaches, the authors decompose global QoS constraints into local constraints with conservative upper and lower bounds. These local constraints are resolved by using an efficient distributed local selection strategy. The crucial aspect is the mapping of global constraints to so-called quality levels which determine a benefit value for using a quality level as local constraint. This is solved by using MIP (Mixed Integer Programming) to find an assignment of local constraints which are used for the local selection. Although this approach is very efficient compared to existing work supporting only hard constraints, it does not allow to specify global soft constraints and can be used only in scenarios where solutions are likely to be found. Moreover, it is very limited in terms of the supported composition constructs because it only supports sequential compositions.

Jaeger et al. [65–67] present an approach for calculating the QoS of a composite service by using an aggregation approach that is based on the well-known workflow patterns by Wil van der Aalst et al. [150]. The authors analyze all workflow patterns for their suitability and applicability to composition and then derive a set of seven abstractions that are well-suited for compositions, so-called *composition patterns*. Additionally, the authors define a simple QoS model consisting of execution time, cost, encryption, throughput, and uptime probability including QoS aggregation formulas for each pattern. The computation of the overall QoS of a composition is then realized by performing a stepwise graph transformation. It identifies a pattern in a graph, calculates the QoS according to pre-defined aggregation functions and replaces the calculated pattern with a single node in the graph. The process is repeated until the graph is completely processed and only one single node remains. For optimizing a composition, the authors analyze two classes of algorithms, namely the 0/1-Knapsack problem and the Resource Constrained Project Scheduling Problem (RCSP). For both algorithms, a number of heuristics are defined to solve the problems more efficiently. Their approach was influential for our approach as we leverage their aggregation concept, however, we use a different algorithm to calculate the aggregated QoS.

Yu et al. [170] discuss algorithms for Web service selection with end-to-end QoS constraints. Their approach is based on several composition patterns similar to [67] and they group their algorithms according to flows that have a sequential structure and others that solve the composition problem for general flows (i.e., flows with splits, loops etc). Based on this distinction, two models are devised to solve the service selection problem: a combinatorial model that defines the problem as Multidimensional Multi-Choice Knapsack Problem (MMKP) and the graph model that defines the problem as a Multi-Constrained Optimal Path (MCOP) problem. These models allow the specification of user-defined utility functions to optimize some application-specific parameters and to enable the specification of multiple QoS criteria taking global QoS into account. In the case of the combinatorial model, the authors use a MMKP algorithm that is known to be NP-complete, therefore, heuristics are applied to solve the problem in polynomial time. For the general flow structure, the authors use an IP approach (also NP complete), thus they again apply different heuristics to reduce the time complexity. Compared to our work, the proposed algorithms and heuristics only deal with hard constraints, soft-constraints are not supported. Currently, we do not apply any heuristics to our algorithms, however, some of their heuristics may also be applicable to constraint hierarchies. However, our IP solution presented in this work is currently fast enough without any heuristics.

Blake and Cummings [22] elaborate on the use of SLAs for effective workflow composition. They propose an approach to annotate Web services with SLA information such as cost, response time, uptime, etc. At composition time, these SLA measures need to be aggregated to ensure that they meet the specified user thresholds. The proposed algorithm constructs the workflow by selecting services that match the user constraints and have the most effective SLA measures. Their work can be seen complementary, however, our QoS-aware composition approach focuses on user QoS constraints for a composite and determines an optimal solution within the boundaries given by the user QoS constraints.

Brandic et al. [24] describe Amadeus, a QoS-aware Grid workflow system that supports a comprehensive set of QoS requirements. In addition to performance and economical aspects, also legal and security aspects are considered. The Amadeus environment comprises: (1) a

Visualization and Specification component; (2) a Planning, Negotiation and Execution component called QoS-aware Grid Workflow Engine (QWE); and (3) a set of Grid resources. A user may specify the workflow with Teuta (a UML-based workflow editor) by composing predefined workflow elements. Different QoS properties such as execution time, price and location affinity may be specified to indicate the user's QoS requirements. For the static planning approach mixed-integer linear programming is applied. Similar to our approach, the authors use an IP to model and solve the problem, however, their approach optimizes a workflow at design time, runtime optimization and adaptive behavior is not supported. Additionally, their approach requires a fully-specified workflow using a graphical language, whereas we use a textual DSL to specify a QoS-aware composition.

Mukhija et al. [101] present the Dino framework that is targeted for the use in open dynamic environments. Their main argument is that no global view on a service composition is available in dynamic environments, therefore, each service specifies which other services it requires for its own execution. The service composition is formed at runtime by the infrastructure. A key aspect is the fact that service requirements can change dynamically (triggered for example by changes in the application context). Dino also supports QoS-aware service composition by describing it formally using an ontology. QoS is computed by using the actual and estimated QoS values that are monitored by the Dino broker. QoS computation is modeled by using a continuous-time Markov chain that enables the association of probability values to express the confidence of the QoS specification. Contrary to our work, their approach focuses on adhoc service composition without having a global view on the composition problem using a broker-based approach by leveraging an efficient local selection of services.

2.4.2 DSLs for Service Composition

Domain-specific languages (DSLs) have a long history, not only in software engineering, also in other areas such as document generation. A well-known example of a DSL is the document typesetting tool TEX. van Deursen et al. [153] define a DSL as follows: "A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

Recently, DSLs have become very popular within the context of Model-Driven Architecture (MDA) and in the field of SOC. However, applying DSLs for QoS-aware composition of service-oriented systems, as we propose in this thesis, is a relatively new idea.

Oberortner et al. [113] discuss the use of DSLs for SOAs in general, with a special focus on model-driven development and process-driven SOAs [173]. The authors differentiate between DSLs for domain experts (high-level DSLs) and DSLs for technical experts (low-level DSL). According to their classification, VCL can be classified somewhere in the middle. On the one hand, it abstracts from low-level semantic and syntactic issues such as constructs in WS-BPEL or any other composition language and provides a high-level approach to specify QoS requirements. On the other hand, it still requires some technical understanding to be practically usable (e.g., how data is passed to the composition).

In [114], the aforementioned authors present a DSL for specifying QoS and SLA requirements, especially for their use in model-driven software development (MDSD). Their DSL is tailored to multiple stakeholders, i.e., non-technical experts versus technical experts, therefore providing a high-level and low-level DSL. Their approach is to capture the SLA requirements and corresponding actions in the high-level DSL and then use the low-level DSL to define how QoS is measured. They use the low-level DSL to generate interceptors for Apache CXF to perform the QoS measurement. In contrast to their approach, VCL is a DSL for the purpose of specifying QoS-aware compositions. To this end, it allows composite service developers to specify their functional and non-functional requirements in form of constraints.

In [154], the author describes the WebDSL approach, a domain-specific language for dynamic Web applications. It allows the specification of domain models, presentation logic, page flows and access control [56]. Similar to our approach, WebDSL abstracts from the complexity of the underlying execution languages and runtimes (JSF, Hibernate and Seam in their approach). Besides abstracting from the underlying runtime, in our approach we additionally introduce an optimization layer in between the language specification and the generation of the executable composition to optimize the local and global QoS constraints that have been specified by the user.

2.4.3 Other Service Composition Approaches

Casati et al. [30] present *eFlow*, a dynamic and adaptive environment for defining, monitoring and enacting composite e-services (modeled through business processes). The main goal is a dynamic service selection based on selection policies and dynamic process adaptation. Adaptation is supported on process instance and process definition level. Contrary to our composition and adaptation approach, eFlow only supports adaptation of tasks in the process based on functional criteria, non-functional aspects are not supported. Our approach supports both, functional and non-functional adaptation of compositions.

JOpera [119], developed at the ETH Zürich, provides a visual composition language which focuses on an interactive environment allowing users to visually specify, design and test their compositions. This approach does not focus on pure SOAP-based services, it also handles arbitrary Java or Enterprise JavaBeans and RESTful services. In contrast to their approach, we go into a different direction by providing a semi-automated approach using a textual DSL that gives developers a simple language to rapidly develop and deploy a composition without requiring any composition infrastructure.

In [76], the authors discuss an approach for interleaving planning and execution of service compositions by means of a special language called XSRL (XML Service Request Language). It enables users to specify goals and constraints for a (pre-compiled) composite service where the services are dynamically bound in the composition, e.g., by using UDDI (Universal Description Discovery and Integration). Their approach is based on AI (Artificial Intelligence) planning and constraint satisfaction techniques to fulfill a service request. In contrast, we focus on providing a language and runtime to specify and deploy a composite service with various constraints in terms of functionality and QoS. Once a composite service is deployed our approach is still able to re-bind to another service once the QoS changes since the initial deployment of the composition.

Charfi and Mezini [32,33] present AO4BPEL, an aspect-oriented workflow language for Web service composition with the goal to increase flexibility and adaptability of BPEL processes at runtime. As one application of AO4BPEL, they propose a process container framework to provide middleware support for WS-BPEL processes. Non-functional properties such as security, reliable messaging or transactions for BPEL activities can be declaratively specified using a deployment descriptor. Compared to their approach, our approach increases flexibility and adaptability of composite applications by providing native support for dynamic invocation, QoS-aware rebinding, and automated invocation-level mediation by using a novel programming model within the VRESCo environment.

Part I

QoS Integration in Service-Oriented Systems

Chapter 3

A Multi-Layer QoS Model for Service-Oriented Systems

This section introduces a multi-layer QoS model for service-oriented systems. It is the foundation for the SLA-aware choreographies presented in Chapter 5 and the QoS-aware composition approach presented in the second part of this thesis.

Contents

3.1 Motivation

Besides considering functional aspects of software systems, non-functional attributes such as performance, dependability, security, safety or usability play a crucial role in the lifecycle and success of a software system. These non-functional attributes are often referred to as "quality attributes" or "quality goals" in literature about software engineering or software architecture [18, 141]. In the SOC community, these quality attributes are manifested by the term "Quality of Service" or QoS for short [87].

The importance of QoS for service-oriented systems is based on the fact that most service-oriented applications use highly-distributed and loosely coupled software services available over the network. These services are invoked by thousands of users concurrently that are often unknown at design or deployment time. Therefore, it is of utmost importance, that services provide, besides their functional interface description (e.g., a WSDL document), a description of non-functional aspects. The QoS attributes of a service can either be classified as deterministic or non-deterministic [81]. The former indicates that their value is known before a service is invoked, including price or penalty rate whereas the latter includes all attributes that are uncertain at service invocation time, for example service availability. Dealing with non-deterministic attributes is more complex since it requires to perform calculations based on data gathered during runtime monitoring. In this thesis, we mainly focus on non-deterministic attributes, since they are essential to realize adaptive behavior in service-oriented systems. However, we also introduce and handle deterministic attributes.

Typically, the design of service-oriented systems comprises multiple layers as seen in Figure 1.2. In this chapter we propose a QoS model encompassing multiple layers from the SOC stack. We do not claim that this model is complete, however, it is designed in an extensible way by allowing to add new QoS attributes without much effort.

Firstly, we describe the lowest layer of the QoS model, the *Service Layer* because it defines all the atomic QoS attributes needed for the other layers. Secondly, we describe the top layer in our model, the *Choreography Layer*, followed by the *Orchestration Layer*.

Service Layer. The lowest layer of the SOC stack represents all atomic services within a service-oriented system. On this layer, various QoS attributes are defined, such as response time or accuracy. Each QoS attribute defined on this layer can be used in the upper layers (such as orchestration or choreography) in an aggregated form to define polices and/or SLAs (Service Level Agreements).

Choreography Layer. A *choreography* represents the global model and viewpoint of all participants and its policies among multiple partners participating in the business process. It describes multi-party peer-to-peer interactions with a strong focus on the data exchanged between multiple partners in the process [151]. A choreography model, such as expressed by WS-CDL [161] is a non-executable description. QoS in its atomic form as used on the service layer is not enough to represent complex quality guarantees between two partners. Therefore, we leverage SLAs [61] on this layer to capture guarantees and obligations between two partners.

Orchestration (or Composition) Layer. An *orchestration* or *composition* represents an executable assembly of services from the service layer by specifying a set of services including their control and/or data flow [84,115]. Popular languages for specifying an orchestration are WS-BPEL [107] or the Windows Workflow Foundation [96]. As discussed in the introduction, we distinguish between macroflows and microflows. The former requires the integration of QoS policies as a means to specify what QoS is expected from a macroflow activity (derived from the SLA associated with the choreography) whereas the microflow requires an aggregation of QoS attributes to represent the overall QoS of all the individual services in the composition. Therefore, we provide a set of aggregation rules for the QoS attributes from the service layer.

3.2 Service Layer

On the service layer, QoS attributes are classified into various QoS classes. The core taxonomy introduces four basic classes as visualized in Figure 3.1.

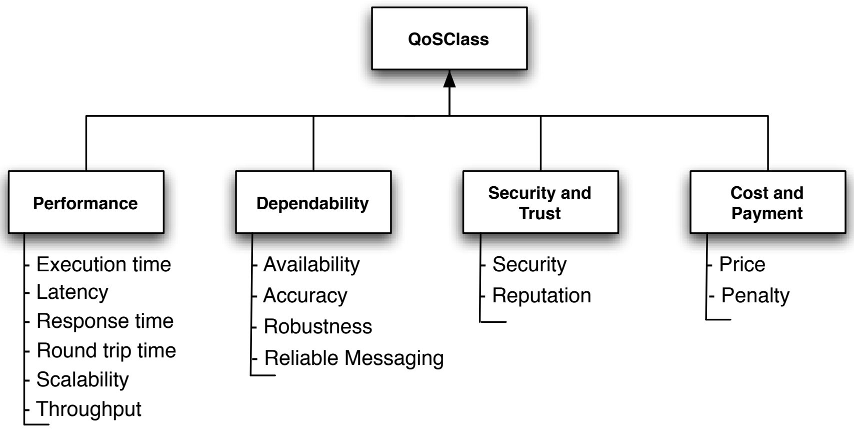


Figure 3.1: Service Layer QoS Taxonomy

In the following, we depict the pre-defined QoS classes and describe each related attribute in detail. For all measurable QoS attributes (the classes *Performance* and *Dependability*), we present a novel and automated monitoring technique in Chapter 4.

3.2.1 Performance

Performance-related QoS comprises a group of attributes related to observable and measurable runtime performance of a service, such as response time or throughput. It is important to distinguish between user-specific attributes such as response time and latency because they depend on both, the users location regarding the distance to the service and the network connection (i.e., modem vs T1). This important distinction is considered later in Chapter 4 when introducing our monitoring approach.

In Figure 3.2, we depict a typical service invocation of a client c, a service s and an operation o by splitting it into its elementary time frames. These time frames show a subset of performance-related QoS attributes that are of interest for a given service. The values $\{t_{ci}|0 \leq i \leq 3\}$ and $\{t_{pi}|0 \leq i \leq 3\}$ represent client-side (c) and provider-side (p) timestamps that can be gathered through instrumentation of the underlying runtime system.

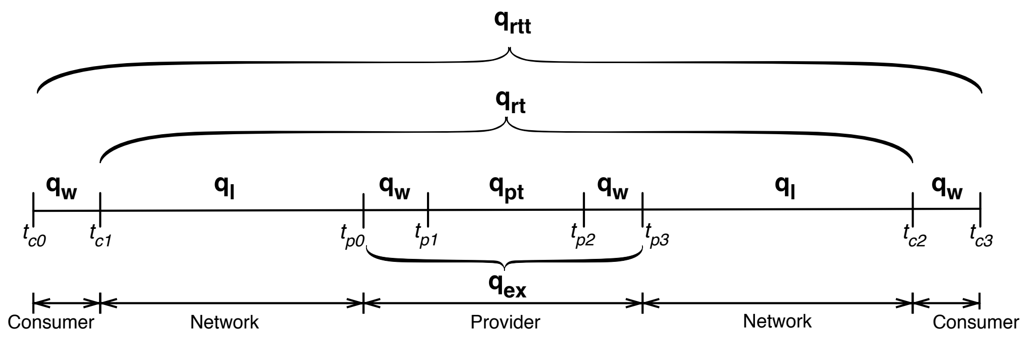


Figure 3.2: Service Invocation Timeline

In the following, we describe each attribute in detail. For each timing-related attribute, we present the formula to illustrate how one single QoS attribute value is calculated. However, when monitoring QoS attributes in real-world environments, we usually take the average of n measurements to get a better approximation of the attribute. Each formula is either consumer- or provider-specific, therefore, we denote a formula representing a consumer-specific QoS attribute with a parameter c in the definition.

Processing Time: Given a service s and an operation o, the processing time on the server is defined as follows:

$$q_{pt}(s, o) = t_{p2} - t_{p1} \tag{3.1}$$

It denotes the time needed to execute an operation for a specific service request. The value is calculated by using the timestamps t_{p1} and t_{p2} taken before and after the processing phase on the server (see Figure 3.2 for details). The processing time does not include any network communication and is, therefore, an atomic attribute with the smallest granularity.

Wrapping Time: The wrapping time of a given service s and an operation o is the time needed to wrap and unwrap an XML message on both, the client- and server-side. We do not consider the wrapping time as a QoS attribute on its own, however, it is needed to calculate other QoS attributes. On the server-side, the wrapping time q_{wp} is defined as follows:

$$q_{wp}(s, o) = t_{p1} - t_{p0} + t_{p3} - t_{p2} \tag{3.2}$$

On the client-side, the wrapping time q_{wc} is defined as follows:

$$q_{wc}(c, s, o) = t_{c1} - t_{c0} + t_{c3} - t_{c2} \tag{3.3}$$

The actual wrapping time value is heavily influenced by the Web service framework (more specifically the XML parser) and even the operating system itself. In [167], the authors even split this time into three sub-values where receiving, (re-)construction and sending of a message are distinguished. For our purpose it does not matter if the delay is caused by the XML parser or the socket connection for reading and writing the message. If non-XML based Web service technologies are used, such as RESTful services [120, 126] based on JSON (JavaScript Object Notation), the wrapping time does not reflect XML messages. Nevertheless, wrapping still occurs, however, in a different context because JSON requests also have to be wrapped to implementation specific objects (such as Java or Ruby objects).

Execution Time: The execution time of a service s and an operation o is defined as the timespan that the service provider needs to process a request. It starts with unwrapping the XML structure, processing the result and wrapping the answer into an XML response that is sent back to the requester. It is calculated as follows:

$$q_{ex}(s, o) = q_{pt} + q_{wp} \tag{3.4}$$

Latency: The latency (or network latency) of a service s and an operation o represents the timespan a service request (the XML message) from a client c needs to reach its destination. During the transmission of a service request, we typically have two latency values, one for the outgoing service request (viewed from a client-side perspective) and one for the incoming service request. We use the average of both latency values:

$$q_l(c, s, o) = \frac{t_{p0} - t_{c1} + t_{c2} - t_{p3}}{2} \tag{3.5}$$

The latency is influenced by the network connection type, routing, network utilization and request size.

Response Time: The response time of a service s and an operation o is the time needed for sending a message from a given client c to s until the response message returns back to the client. The response time is consumer-specific, therefore, it is not possible to specify a globally valid value for each client. The response time is calculated by using the following formula:

$$q_{rt}(c, s, o) = q_{ex}(s, o) + 2 * q_l(s, o) \tag{3.6}$$

Round Trip Time: The round trip time of a service s and an operation o represents the overall time that is consumed from the moment a request is issued at the client c to the moment the answer is received and successfully processed (i.e., the user or application can see the response). It comprises all values on both, consumer and provider-side. The round trip time is calculated as follows:

$$q_{rtt}(c, s, o) = q_{wc}(s, o) + q_{rt}(s, o) \tag{3.7}$$

Throughput: The throughput of a service s denotes the number of Web service requests r for an operation o that can be successfully processed by s and returned to client c within a given time interval $[t_0, t_1]$. It is calculated using the following formula:

$$q_{tp}(c, s, o) = \frac{r}{t_1 - t_0} \tag{3.8}$$

The throughput depends mainly on the hardware power (CPU, memory, IO subsystem) and the network bandwidth of the service provider. It is measured by sending several requests in parallel (e.g., 10000) for a given period of time (e.g., 10 seconds) and then count how many valid requests come back to the requester to calculate a throughput value for a time interval of one second (e.g., $\frac{10000}{10} = 1000$ operations per second).

Scalability: The term scalability is frequently used in the context of system performance, however, as stated in [43], it is poorly defined and poorly understood. One good definition that fits into our context is from Smith and Williams [138]: "Scalability is the ability of a system to continue to meet its response time or throughput objectives as the demand for the software functions increases." Following their definition, we calculate the scalability specific to a client c of a service s and an operation o using the following formula:

$$q_{sc}(c, s, o, m) = \frac{\frac{1}{n} \sum_{i=1}^{n} q_{rtt_i}(s, o)}{\frac{1}{m} \sum_{i=1}^{m} q_{rtt(Throughput)_i}(s, o)} \tag{3.9}$$

Basically, the scalability expresses the ratio between the average round trip time (over n measurements) in the numerator and the average round trip time measured during a throughput test of m parallel requests in the denominator. The expression $q_{rtt(Throughput)}$ refers to the round-trip time during a throughput measurement. If the service s is not scalable and gets overloaded by the high number of parallel request m, the round trip time during the throughput tests increases (denominator), thus, leading to a decrease in the overall scalability.

3.2.2 Dependability

Avižienis et al. [10] define dependability of a system as "the ability to avoid service failures that are more frequent and more severe than is acceptable". It evolved to an integrated concept for distributed systems encompassing attributes such as availability, reliability or integrity. In our approach, we only consider those attributes for Web services which are relevant from a consumer's perspective, such as availability, accuracy, etc. In contrast to performance-specific QoS attributes, dependability-related QoS attributes are measured on the service level not on the operation level.

Availability: The availability q_{av} defines the probability that a service s is up and running and produces correct results. The availability is calculated using the following formula:

$$q_{av}(s) = 1 - \frac{t_d}{t_1 - t_0} \tag{3.10}$$

The variables t_0 and t_1 represent timestamps for the overall uptime of the service, whereas t_d denotes the downtime of a service in seconds.

Accuracy: The accuracy q_{ac} of a service s is defined as the success rate produced by s. It is calculated by recording all invocations within a given time interval and relate it to all failed requests during that interval. The following formula expresses this relationship:

$$q_{ac}(s) = 1 - \frac{r_f}{r_t} \tag{3.11}$$

The variable r_f denotes the number of failed requests, whereas r_t expresses the number of total requests within the observation period.

Robustness: The robustness q_{ro} of a service s defines the probability that a system can react properly to invalid, incomplete or conflicting input messages. It can be measured by tracking all incorrect input messages and put them in relation with all valid responses from a given point in time:

$$q_{ro}(s) = \frac{1}{r_t}\sum_{i=1}^{n} f(resp_i(s)) \tag{3.12}$$

We use a helper function $resp_i(s)$ to denotes the i-th response produced by service s where n is the total number of requests issued to s. The utility function $f(m)$, where m denotes a response message, is used to evaluate whether the response was correct for a given input. It is calculated using the following formula:

$$f(m) = \begin{cases} 1, & isValid(m) \\ 0, & \neg isValid(m) \end{cases} \tag{3.13}$$

The function $isValid(m)$ checks whether a response is correct in terms of syntax and semantic. The validity checks do not consider any service-specific semantic or exceptions. The robustness property only requires that a service handles malformed input in a way that it does not crash or continue to return wrong messages and is able to react appropriately and resume using its expected behavior.

Reliable Messaging: Guaranteed message delivery between requesters and providers is the core idea behind reliable messaging (RM). The Web service stack for example defines the WS-Reliability standard [109] to define how reliable messaging can be achieved in an application agnostic way (i.e., the application does not notice that RM is available). We define reliable messaging simple by using the following function:

$$q_{rm}(s) = \begin{cases} true, & \text{RM available for } s \\ false, & \text{RM not available for } s \end{cases} \tag{3.14}$$

3.2.3 Security and Trust

Security is an important issue in SOC [47], however, it does not receive much attention in current research. Many services consume and produce plain text messages, even for important and confidential data and no service-level security mechanisms are used. Currently, most security solutions in SOC consist of establishing a VPN (Virtual Private Network) among the providers and consumers of certain services. Another commonly used method is SSL (Secure Socket Layer) as transport-level security. The same is true for trust, currently no assembled solution exists to define trust among services, expect the OASIS WS-Trust specification [111], however, it is not widely adopted. Reflecting security and trust related QoS for services is important to enable discovery of trusted and secure services, otherwise, no one would use a credit card service in a real-world application without proper encryption and transport-level security.

Security: The security attribute $q_{sec}(s)$ defines the security mechanism that is supported by service s. These values can be `None` (for no security support), `UsernamePassword`, `X.509`, `SAML` or `Kerberos`. The security mechanisms that can be used are basically aligned with the security tokens that can be specified as part of the WS-Security framework [112].

Reputation: The reputation $q_{rep}(s)$ of a service s is a measure for the trustworthiness of a service as rated by the service consumers [81]. Typically, consumers have a different experience and trust when using a service, thus, simple reputation mechanisms such as user-feedback allowing to express this experience using different metrics. Assuming numerical reputation values such as $[1, 5]$ (1 is the worst, 5 the best), we can calculate the average reputation of a service s over n values r_i as follows:

$$q_{rep}(s) = \frac{1}{n}\sum_{i=1}^{n} r_i \tag{3.15}$$

3.2.4 Cost and Payment

Cost and payment represent QoS attributes that are related to monetary costs when invoking and using services. Since cost is a main business value for optimizing service-oriented systems, in particular service compositions, we address the specification of related QoS attributes in this category.

Cost: The cost $q_c(s, o)$ of a service s represents the monetary value that is associated with a specific service operation o when invoking it. Typically, cost is calculated per invocation, however, more advanced pricing strategies can be implemented and defined in SLAs, such as usage contingents (e.g., 100 EUR for the first 1000 invocations, then 80 EUR for the next 1000). As complex pricing strategies are usually modeled on a higher level, we simply assume that a cost value can be assigned to a service to achieve optimization of composite services without the awareness of all details specified in an SLA.

Penalty: The penalty $q_{pl}(s, o)$ of a service s and an operation o is the monetary value that the provider has to pay to the consumer in case a QoS value, as negotiated in an SLA, is violated.

Similar to cost, a penalty is usually negotiated on an SLA level, however, we define it as a QoS attribute on the service layer to ensure that a QoS-aware optimization strategy can leverage this attribute to minimize the penalty if desired.

Attribute	Symbol	Dimension	Unit	Deterministic
		Performance		
Processing Time	q_{pt}	desc	μs	no
Execution Time	q_{ex}	desc	μs	no
Latency	q_{l}	desc	μs	no
Response Time	q_{rt}	desc	μs	no
Round Trip Time	q_{rtt}	desc	μs	no
Throughput	q_{tp}	asc	operation/sec	no
Scalability	q_{sc}	asc	percent	no
		Dependability		
Availability	q_{av}	asc	percent	no
Accuracy	q_{ac}	asc	percent	no
Robustness	q_{ro}	asc	percent	no
Reliable Messaging	q_{rm}	exact	{true, false}	yes
		Security and Trust		
Security	q_{sec}	exact	{None, UsernamePassword, X.509, Kerberos, . . . }	yes
Reputation	q_{rep}	user-defined	user-defined	no
		Cost and Payment		
Cost	q_{c}	desc	monetary value	yes
Penalty	q_{pl}	desc	monetary value	yes

Table 3.1: Summary of Service Layer QoS Attributes

3.2.5 Summary of QoS Attributes

Table 3.1 summarizes the available QoS attributes and also denotes their dimension, the unit and whether an attribute is deterministic. The dimension column indicates the QoS attribute dimension expressing whether a ascending, descending or exact QoS value is generally better. For example, a lower response time is generally better, therefore the response time has *descending* dimension (abbreviated as *desc*). The throughput, for example, has a *ascending* dimension (abbreviated as *asc*) whereas reliable messaging has an *exact* dimension meaning that either it is supported or not. These dimensions are especially important when using QoS attributes in a composition because they indirectly influence the QoS-aware optimization (e.g., while the response time of a composition should generally be minimized, the throughput should be maximized). The unit column denotes the unit of measurement for each QoS attribute (e.g., μs or msec for performance related attributes). The final column indicates whether a QoS attribute is deterministic or non-deterministic [81].

3.3 Choreography Layer

The top layer of the QoS model aligns QoS in general with choreographies. The main goal is to express guarantees and obligations in terms of QoS between contractual parties on a high level of abstraction. The contractual agreements between two partners are commonly referred to as SLA [71]. Contrary to choreographies, which define a global viewpoint among multiple partners, SLAs are mutually defined between two partners. As a consequence, multiple SLA are needed between partners in a choreography (cf. Figure 1.2). Our multi-layer QoS model does not enforce any particular SLA dialect (e.g., WSLA [61], WS-Agreement [55], SLang [137] or the Web Service Offering Language (WSOL) proposed by Tosic et al. [147]). For our prototype implementation of integrating SLAs into choreographies, we use WSLA as an SLA language, therefore, we briefly introduce SLA-specific concepts within the scope of WSLA.

SLAs can describe different arbitrary information about the agreement itself, however, there is a common understanding in literature what an SLA, specifically related to SOC, should contain [71]. WSLA accommodates this common structure by defining three major sections:

- *Parties*: This section identifies and defines all contractual parties including their identification information and all related technical properties such as the interface descriptions or service endpoints.
- *Service Descriptions*: This section defines all service characteristics (e.g., operation names) and the observable parameters (referred to as SLA parameters) as well as the metrics that are used to monitor a service.
- *Obligations*: The third section defines constraints for various guarantees on SLA parameters defined in the previous section. Such obligations are expressed by *Service Level Objectives (SLOs)* using a combination of various QoS parameters.

We briefly introduce two simple examples to illustrate the definition of an SLA parameter and an SLO. An example definition of an SLA parameter `ExecutionTime` (lines 1–3) and `Throughput` (lines 5–7) is given in the following Listing 3.1.

Based on the above SLA parameter definition, these parameters can now be used to define a specific SLO on a given service. In Listing 3.2, an objective called `ServicePerformance` is defined to express, as the name implies, the performance of a *HardwareSupplier* service (line 2)

```xml
<SLAParameter name="ExecutionTime" type="ExecutionTime" unit="msec">
  <Metric>ExecutionTimeMetric</Metric>
</SLAParameter>

<SLAParameter name="Throughput" type="Throughput" unit="ops">
  <Metric>OperationsPerSecondMetric</Metric>
</SLAParameter>
```

Listing 3.1: SLA Parameter Definitions

as a combination of throughput and execution time. More specifically, it states that the execution time has to be less than 1500 msec (lines 9–14) and the throughput has to be greater or equal to 130 operation per second (lines 15–20). The validity of the SLO can be expressed using the `Validity` element (lines 3–6), thus enabling the definition of different SLOs for different time slots. The optional `EvaluationEvent` in line 23 defines that the SLO should be evaluated, in this case whenever a new value is detected (element text `NewValue`). WSLA provides a set of operators, functions and predicates to define SLOs. Whenever an SLO is violated by the service provider, numerous *action guarantees* can be defined (not shown in the listing) to react appropriately to these violations, e.g., by charging a pre-negotiated penalty rate to the provider. However, WSLA does not define how these violations as well as pricing and penalty models can be handled. One approach would be a rule-based implementation, e.g., using distributed business rules [102, 135] among the involved parties.

```xml
1 <ServiceLevelObjective name="ServicePerformance">
2   <Obliged>HardwareSupplier</Obliged>
3   <Validity>
4     <Start>2009-01-01T00:00:00.000+01:00</Start>
5     <End>2009-12-31T00:00:00.000+01:00</End>
6   </Validity>
7   <Expression>
8     <And>
9       <Expression>
10        <Predicate xsi:type="wsla:Less">
11          <SLAParameter>ExecutionTime</SLAParameter>
12          <Value>1500</Value>
13        </Predicate>
14      </Expression>
15      <Expression>
16        <Predicate xsi:type="wsla:GreaterEqual">
17          <SLAParameter>Throughput</SLAParameter>
18          <Value>130</Value>
19        </Predicate>
20      </Expression>
21    </And>
22  </Expression>
23  <EvaluationEvent>NewValue</EvaluationEvent>
24 </ServiceLevelObjective>
```

Listing 3.2: Service Level Objective Example

In general, the information specified as part of the SLA can be seen as a contract between the service provider and the consumer. Therefore, SLOs in the contract need to be monitored and enforced by the runtime environment. However, from an SLA description it is hard to identify and enforce which SLOs affect which parts of the business process (especially the interactions with the partners). Therefore, we have identified a mapping between SLAs on this layer, to concrete QoS policies on the orchestration layer. The policy language is based on common Web service standards and is described in the next section. As already mentioned, our approach does not enforce any particular SLA dialect, however, an agreement on a common set of QoS attributes is required. In our approach, we use the set of attributes defined on the service layer. Unfortunately, a service provider will only use a subset of our service layer QoS attributes in an SLA, as some values depend on non-controllable elements, such as the network connection from the service consumer to the provider. Therefore, a real-world SLA will exclude all QoS attributes that include network communication (such as latency, response time or round trip time).

3.4 Orchestration Layer

In our quality model, the orchestration layer comprises the business processes (macroflows) and all its compositions (microflows) as shown in Figure 1.2. The main goal of the QoS integration on this layer is twofold:

- SLAs specified between partners on the choreography layer (see Section 3.3) need to be associated with the corresponding parts of the business process to enable SLA enforcement. To achieve this integration we use *QoS policies*, specified using WS-Policy [165], to achieve a loosely coupled integration into business processes.
- In order to enable validation of SLA values, accurate QoS information for each service is needed. These values are obtained through service monitoring, however, they need to be aggregated to reflect the QoS of a composition. Aggregation is performed based on well-defined composition patterns [67, 150].

3.4.1 Integration of QoS Policies

Long-running business processes (or macroflows) are typically implemented using a workflow system such as YAWL [152] or more recently using a WS-BPEL engine such as ActiveBPEL [1]. WS-BPEL has become the de-facto standard language for process orchestration within the Web services stack. Commonly, these systems do not provide native support for specifying and enforcing SLAs. Typically, a process invokes services from multiple partners, therefore, we assume that SLAs are defined among the partners. However, at runtime the process engine invokes a service from a specific provider as part of the process definition but is not aware of the negotiated SLOs that need to be checked against the observable QoS value. Therefore, the relevant SLA specific information from the choreography layer is mapped to

QoS policies that can then be attached to business processes. We present this mapping in detail in Chapter 5.

3.4.1.1 WS-Policy

The Web services stack defines an extensible framework, called WS-Policy with the goal to define "a framework and model for expressing policies that refer to domain-specific capabilities, requirements, and general characteristics of entities in a Web-services-based system" [165]. Generally, a policy in WS-Policy terminology is a collection of policy alternatives which itself is a collection of policy assertions. Policy assertions define requirements, capabilities, or other properties of a policy subject (e.g., an endpoint, message, operation, etc). Policy assertions express domain-specific semantics (e.g., transactions or security) and are defined in separate specifications. A policy expression is an XML Infoset representation of a policy whereby four major elements are used to define a policy expression: `Policy`, `All`, `ExactlyOne`, `PolicyReference`. The first three elements are referred to as policy operators and are used to define policies (`Policy` element) or combine policy assertions (`All` and `ExactlyOne`) whereas the fourth is used to reference a policy expression, either as standalone policy or within another policy expression. Additionally, an XML attribute `wsp:Optional` can be used to specify optional policy assertions.

3.4.1.2 WS-QoSPolicy

Currently, the WS-Policy specification focuses on security using WS-Security Policy [110] and reliable messaging using WS-RM Policy [108], whereas other QoS related policies are currently not available. In order to achieve an automated mapping from SLAs to QoS policies, we need to extend the WS-Policy framework to enable support for the definition of QoS policies.

The WS-QoSPolicy defines an assertion model for the service layer QoS attributes which are not influenced by non-controllable behavior for the service provider such as the network communication. The normative outline of the new policy assertions is shown in Listing 3.3.

```
<qosp:[QoS]Assertion unit="xs:string" predicate="tns:PredicateType"
                     value="xs:integer|xs:double|xs:string"/>
```

Listing 3.3: WS-QoSPolicy Assertions

The value [QoS] acts as a placeholder for concrete assertions, such as `ExecutionTime` or `Throughput`. The `unit` attribute defines the measurement unit for each attribute. We simply use a string to denote the unit because we have a small and finite set of QoS attributes and the units for these attributes are clearly defined. The predicate defines the logical expression that is used for the QoS value comparison and is bound to the following values: `Greater`, `Less`, `Equal`, `GreaterEqual`, `LessEqual` (defined by the `PredicateType` schema element). The policy assertion value can either be a string, a double or an integer. In Listing 3.4, a small excerpt of the WS-QoSPolicy is depicted. It shows the definition of the `ExecutionTime-Assertion` (lines 11–17) and the `ThroughputAssertion` (lines 18–24) and the available predicates (lines 28–36).

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <xs:schema xmlns:tns="http://vresco.vitalab.tuwien.ac.at/wsqospolicy"
 3            xmlns:xs="http://www.w3.org/2001/XMLSchema"
 4            xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
 5            targetNamespace="http://vresco.vitalab.tuwien.ac.at/wsqospolicy"
 6            elementFormDefault="qualified"
 7            attributeFormDefault="unqualified" version="1.0">
 8    <xs:import namespace="http://schemas.xmlsoap.org/ws/2004/09/policy"
 9               schemaLocation="ws-policy.xsd"/>
10
11    <xs:element name="ExecutionTimeAssertion">
12       <xs:complexType>
13          <xs:attribute name="value" type="xs:integer" use="required"/>
14          <xs:attribute name="unit" type="xs:string" use="required"/>
15          <xs:attribute name="predicate" type="tns:PredicateType" use="required"/>
16       </xs:complexType>
17    </xs:element>
18    <xs:element name="ThroughputAssertion">
19       <xs:complexType>
20          <xs:attribute name="value" type="xs:double" use="required"/>
21          <xs:attribute name="unit" type="xs:string" use="required"/>
22          <xs:attribute name="predicate" type="tns:PredicateType" use="required"/>
23       </xs:complexType>
24    </xs:element>
25
26    <!-- other elements cut for readability -->
27
28    <xs:simpleType name="PredicateType">
29       <xs:restriction base="xs:string">
30          <xs:enumeration value="Greater"/>
31          <xs:enumeration value="Less"/>
32          <xs:enumeration value="Equal"/>
33          <xs:enumeration value="GreaterEqual"/>
34          <xs:enumeration value="LessEqual"/>
35       </xs:restriction>
36    </xs:simpleType>
37 </xs:schema>
```

Listing 3.4: WS-QoSPolicy Schema Excerpt

A concrete example for two such policy assertions is illustrated in Listing 3.5. This example basically reassembles the SLO from Listing 3.2.

Based on these policy assertions, QoS requirements for services can be integrated in business processes using the WS-QoSPolicy. It is then up to the process engine to handle these assertions properly, e.g., by emitting events to the SLA enforcement service to notify that a specific value is violated. This information can then be correlated with the information in the SLA to define appropriate actions. To this end, this approach can form the basis for effective SLA enforcement, however, this thesis does not address enforcement on its own, it rather focuses on the integration of SLA and QoS in the modeling process and on the optimization and

```
<wsp:Policy>
  <wsp:All>
    <qosp:ExecutionTimeAssertion unit="msec" predicate="Less" value="1500"/>
    <qosp:ThroughputAssertion unit="ops" predicate="GreaterEqual" value="130"/>
  </wsp:All>
</wsp:Policy>
```

Listing 3.5: Assertion Example

adaptation of QoS-aware compositions on lower levels.

3.4.2 Aggregation of Service Layer QoS

In order to calculate the QoS of a composite service on the microflow level, a set of well-defined composition patterns needs to be identified within the overall composition. For each of these patterns a QoS aggregation rule can be applied to calculate the QoS of a pattern and, by applying it recursively, the overall QoS of a composition. In [150], van der Aalst et al. have defined a catalog of workflow patterns which typically occur in modern workflow systems. These patterns are not directly applicable to typical composition scenarios because they contain many patterns that are not relevant for QoS aggregation. Additionally, workflow patterns are elementary units such as splits or joins, however, in order to perform QoS aggregation a structured pattern is needed that combines for example a split and a join in one pattern.

3.4.2.1 Composition Patterns

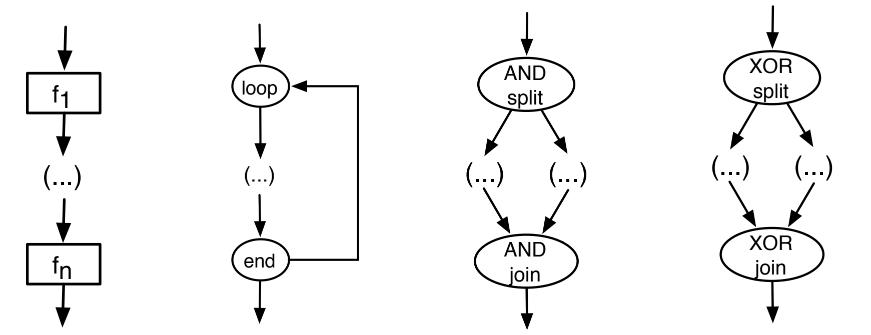
Jaeger et al. [67, 68] have identified a set of theoretically implementable composition patterns from van der Aalst's pattern catalog. Initially, they have identified seven patterns [67,68], later extending it to nine possible patterns [64]. These composition patterns range from simple sequences of activities and loops to parallel or conditional executions with different semantics. The following subset of composition patterns is used in our approach and visualized in Figure 3.3 for two reasons: Firstly, the domain-specific language VCL, for specifying QoS-aware compositions, that we contribute in this thesis does not support complex composition patterns. Additionally, the underlying composition language that we use to actually execute a VCL composition does not support them either. Secondly, these four composition patterns used in our approach are usually sufficient to express common compositional logic.

Pattern 1: Sequence. Defines a sequential execution of n services in a composition. The length of a sequence has to be at least greater than one. The pattern is shown in Figure 3.3a.

Pattern 2: Loop. A loop defines a repetitive execution of its body a given number of n times. The pattern is shown in Figure 3.3b.

Pattern 3: AND split/AND join. An AND split specifies a parallel execution of n different branches. The AND join represents a barrier that waits for all n parallel executions to complete before continuing with the subsequent activities in the composition. The pattern is shown in Figure 3.3c.

Pattern 4: XOR split/XOR join. An XOR split chooses one out of n branches (based on a condition) that is executed. The final XOR join waits for exactly one branch to finish before continuing with the execution of subsequent activities in the composition. It is typically used to model conditional execution (`if-then`) as known from traditional programming languages. The pattern is shown in Figure 3.3d.



(a) Sequence (b) Loop (c) AND split/AND join (d) XOR split/XOR join

Figure 3.3: Composition Patterns

3.4.2.2 Aggregation Rules

Based on the above mentioned composition patterns, a unique aggregation formula is needed to calculate the QoS attribute value of a pattern that is then used recursively to calculate the overall QoS of the composition. The atomic aggregation formulas for each composition pattern are presented in Table 3.2. These patterns are then used by the aggregation algorithms for our QoS-aware Composition as a Service (CaaS) approach presented in Chapter 8.

For defining the aggregation rules, we use the following notation. A pattern in a composition consists of a set of n activities (or features as it is called in VRESCo – see Chapter 6) $F = \{f_1, f_2, \dots f_n\}$. We assume that loops and conditionals (XOR-XOR pattern) are annotated at design time to reflect the loop count and the execution probability. For the loop count, we use the variable c in the aggregation rules. In the XOR split, we re-use f_i to represent all possible branches, each with a probability p_i where $\sum_i^n p_i = 1$. Additionally, we use the QoS attribute as a function on a feature $f_i \in F$ to get its QoS value. For example $q_{tp}(f_1)$ retrieves the throughput value of a feature f_1 in a composition. It is important to note that the annotations for loops and the XOR-XOR pattern are used at design time in our QoS-aware composition approach to initially aggregate the QoS and generate and optimized composition w.r.t. the

Attribute	Sequence	Loop	XOR-XOR	AND-AND
		Performance		
Response Time[a] (q_{rt})	$\sum_{i=1}^{n} q_{rt}(f_i)$	$q_{rt}(f) * c$	$\sum_{i=1}^{n} p_i * q_{rt}(f_i)$	$\max\{q_{rt}(f_1), .., q_{rt}(f_n)\}$
Throughput (q_{tp})	$\min\{q_{tp}(f_1), .., q_{tp}(f_n)\}$	$q_{tp}(f)$	$\sum_{i=1}^{n} p_i * q_{tp}(f_i)$	$\min\{q_{tp}(f_1), .., q_{tp}(f_n)\}$
Scalability (q_{sc})	$\min\{q_{sc}(f_1), .., q_{sc}(f_n)\}$	$q_{sc}(f)^c$	$\sum_{i=1}^{n} p_i * q_{sc}(f_i)$	$\min\{q_{sc}(f_1), .., q_{sc}(f_n)\}$
		Dependability		
Availability[b] (q_{av})	$\prod_{i=1}^{n} q_{av}(f_i)$	$q_{av}(f)^c$	$\sum_{i=1}^{n} p_i * q_{av}(f_i)$	$\prod_{i=1}^{n} q_{av}(f_i)$
Reliable Messaging (q_{rm})	$q'_{rm} = \begin{cases} true & \forall_{1<i\leq n} q_{rm}(f_i) = true \\ false & \exists_{f_i \in F} q_{rm}(f_i) = false \end{cases}$			
		Security and Trust		
Security (q_{sec})	$q'_{sec} = \begin{cases} X.509 & \forall_{1<i\leq n} q_{sec}(f_i) = X.509 \\ None & otherwise \end{cases}$			
Reputation (q_{rep})	$\frac{1}{n}\sum_{i=1}^{n} q_{rep}(f_i)$	$q_{rep}(f)$	$\sum_{i=1}^{n} p_i * q_{rep}(f_i)$	$\frac{1}{n}\sum_{i=1}^{n} q_{rep}(f_i)$
		Cost and Payment		
Price (q_p)	$\sum_{i=1}^{n} q_p(f_i)$	$q_p(s) * c$	$\sum_{i=1}^{n} p_i * q_p(f_i)$	$\sum_{i=1}^{n} q_p(f_i)$
Penalty (q_{pl})	$\sum_{i=1}^{n} q_{pl}(f_i)$	$q_{pl}(s) * c$	$\sum_{i=1}^{n} p_i * q_{pl}(f_i)$	$\sum_{i=1}^{n} q_{pl}(f_i)$

[a]The same aggregation rule is used for execution time, latency, processing time and round trip time.
[b]The same aggregation rule is used for accuracy and robustness.

Table 3.2: QoS Attributes and Aggregation Formulas

structure of the composition and the annotations of the QoS values. Once a composition is deployed and running, these annotated values will be adapted based on execution monitoring of the composition to accurately reflect historical loop counts and XOR branching decisions.

In case of security, we simply assume that if one service in the composition requires a specific security protocol (e.g., X.509), then all services in the composition have to support the same security protocol. In the formula in Table 3.2, we only formally define it for the case of X.509, however, it is the same for all other possible security attributes (e.g., such as `UsernamePassword`). In case of reliable messaging, the same restriction applies. A more sophisticated handling of security in Web service compositions is left for future work.

3.5 Summary

In this chapter we have introduced the multi-layer QoS model that forms the basis for the further chapters presented in this thesis. The multi-layer model combines some of the basic modeling concepts in SOC, namely choreography, orchestration and services as the core entities. Each layer has different requirements and goals, especially w.r.t. their specification of non-functional properties of the overall system. Our model takes these different requirements into account and presents an extensible model for the Web services stack. Firstly, we presented a detailed set of QoS attributes and their definitions grouped into four basic categories. Based on the service layer QoS, we have outlined the specification of SLAs on the choreography layer as the most general way of modeling non-functional guarantees in SOA-based systems. The definition of SLAs during the choreography modeling enables a mapping of these SLAs to concrete QoS policies on the orchestration layer. An orchestration engine can then enforce these QoS policies and define appropriate actions for dealing with violated policies. The QoS policies are specified using WS-QoSPolicy, a custom extension to the WS-Policy framework. Orthogonal to the concept of defining QoS policies is the pattern-based aggregation of QoS attributes on the orchestration layer. Such an aggregation is necessary to calculate the QoS of the overall composition or for specific parts of the composition. This is especially helpful when optimizing a composition in terms of QoS as done in the second part of this thesis where certain user-defined QoS constraints have to be enforced.

Chapter 4

Monitoring and Measuring Web Service QoS Attributes

This chapter introduces a client-side QoS monitoring approach that is used to bootstrap and evaluate performance and dependability related QoS attributes of Web services in a black-box manner.

Contents

4.1 Motivation

Efficient and automated monitoring of Web service QoS attributes is a crucial aspect enabling a large number of application scenarios such as QoS-aware Web service selection, composition and optimization. Currently, most Web services do not provide QoS information as part of their interface contract, simply because there is no "external" entity to bootstrap and assess

QoS information and attach it to a service. However, in case they do, it cannot be guaranteed that this information is accurate and can be trusted.

In this chapter, we present a client-side monitoring approach and a tool called QUATSCH. This approach treats a Web service as a black-box and invokes it just as a regular Web service client would do. A novelty of this approach is the fact that the server-side execution time (cf. with Figure 3.2 in Chapter 3) can be estimated by using low-level TCP sniffing and analysis without requiring access to any service internals. This makes the tool a reasonable candidate to validate the service provider processing time that is often guaranteed as part of an SLA.

An alternative approach for monitoring Web service QoS within compositions is implemented as part of the VieDAME system. This non-intrusive approach monitors performance-specific QoS attributes within a WS-BPEL process execution and allows to trigger adaptations based on the measured QoS. It leverages aspect-oriented programming (AOP) to retrieve the monitoring data from the process engine without instrumenting the business process code nor changing the code of the process engine. For a brief description of VieDAME, we refer to Chapter 5 and for a detailed description to [99].

4.2 Overview of Monitoring Approaches

A number of different approaches for QoS monitoring exist, each of them has its advantages and drawbacks [143]. In this section, we give a brief conceptual overview of available approaches for monitoring atomic Web services (not compositions) which are applicable in service-oriented systems.

A general design consideration when implementing a QoS monitoring approach is the decision whether to monitor on the *service-side* or from the *client-side*. Server-side monitoring is easier and less cumbersome as the Web service source code is typically available and can be directly instrumented with performance measurement code. Additionally, the measurement accuracy is typically higher and several QoS attributes can be calculated more precisely (e.g., throughput). However, by applying a server-side approach it is not possible to measure the network latency and therefore the "real" QoS experience delivered to a service consumer such as the response time (not just the execution time on the server). An alternative is a pure client-side approach, where only the WSDL interface of the Web service is available. Therefore, such an approach requires other ways to measure the execution time q_{ex} or the processing time q_{pt} since these attributes cannot be directly measured or calculated without analyzing the TCP traffic.

4.2.1 Provider-Side Instrumentation

Instrumenting the service implementation at the provider is a simple approach to measure various dependability-specific QoS attributes. In general, one has to distinguish between instrumenting the service code directly to calculate the QoS attributes (invasive instrumentation) or use mechanisms and tools which allow a non-invasive instrumentation, e.g., by using

AOP or operation system specific mechanisms such as Windows performance counters [98]. The advantage of this approach is the monitoring accuracy because all measurements are based on real-time invocations and high-load during peek times are immediately reflected by an increase or decrease of the corresponding QoS attribute. The main drawback of this approach is the lack of monitoring capabilities for all the client-specific attributes that include network latency (cf. all non-deterministic client-side attributes from Chapter 3). Additionally, the service consumer has to trust the performance values specified by the provider.

4.2.2 SOAP Intermediaries

SOAP intermediaries are applications that can accept, process, and forward SOAP messages routed from the origin to its final destination. Such an intermediary party can be used to measure and handle QoS related data thus act as a "trusted" external QoS monitor between the client and the server. Advantage of an intermediary is the fact that it is loosely coupled and, therefore, can be provided by the client or server, or even by a "trusted" third party. However, this third party is effectively a proxy and all requests need to be routed through this proxy to facilitate a seamless and continuous monitoring. This can lead to bottlenecks at intermediary nodes due to a high runtime overhead which also limits scalability. Additionally, both client and server have to agree on a dedicated intermediary by adding the corresponding information to each incoming and outgoing SOAP message.

4.2.3 Probing

Probing is a technique where a probing entity between the service consumer and the provider issues probes to the service provider on a regular basis. Contrary to SOAP intermediaries, this approach does not intercept the traffic between the consumer and the provider, however, it sends probing requests to the service provider. The main advantage of this approach is the flexibility regarding its location. It can either be situated on the client-side, service-side or in between as an independent third-party monitoring tool. Typically such a probing approach is used to check the availability of a certain network service and measure its uptime. In order to be useful for measuring the QoS, a probing toolkit needs to be configurable and extensible to enable an automated invocation of various Web services with different payloads (to simulate different message payloads).

4.2.4 Sniffing

An alternative approach for client-side monitoring is the analysis of low-level TCP packet streams during the invocation of a service to accurately calculate the response time, latency and other values. Different from all above mentioned approaches is the fact that these values are consumer-specific, which is often a desired goal when monitoring QoS attributes in real-world applications. This approach usually has a higher runtime overhead because a set of technologies are needed to facilitate sniffing, aggregation and correlation of low-level data. However, sniffing, correlation and aggregation of measurement data is typically executed asynchronously from the real business application, therefore, reducing possible interferences and providing higher stability.

4.3 Client-Side Monitoring Approach

In this section, we present a novel QoS bootstrapping and evaluation approach called QUATSCH for monitoring performance and dependability related QoS attributes as described in Chapter 3. The approach implements a *client-side monitoring technique* for Web services which works completely service and provider independent. It provides a black-box view on a service just as a regular Web service client. The approach combines sniffing and probing to enable a client-side view on QoS attributes and allows to calculate the server-side execution time by using low-level TCP traffic sniffing.

Several processing steps are necessary to successfully bootstrap and evaluate QoS attributes for arbitrary Web services, therefore, we have split the evaluation approach into three different phases.

1. **Preprocessing**: In this phase a Web service which should be monitored is preprocessed in the sense that stubs are being generated and an evaluation configuration has to be defined (e.g., which operation to invoke and how often). Since Web services based on SOAP come in different flavors, such as RPC-style or document-literal style, we use a different framework and preprocessing strategy for each style. The stub code together with the rest of the configuration data is stored in the database.

2. **Evaluation**: In this phase the actual QoS monitoring is performed. Based on the previously generated evaluation configuration, the generated stubs are dynamically instantiated to invoke the service in order to calculate a set of QoS attributes. A scheduler component handles the continuous service monitoring based on the interval specified in the evaluation configuration. The results of the scheduling are stored in the database.

3. **Result Analysis**: The result analysis uses the raw QoS evaluation data from the previous phase to generate QoS statistics. This step is supported by using a Web-based user interface (UI) where the user can generate performance charts and inspect raw QoS data. Additionally, the UI supports the aforementioned preprocessing steps by adding, removing or changing existing services and their evaluation schedules.

4.3.1 QUATSCH Toolkit

The QUATSCH toolkit is implemented as a three tier approach by using Hibernate [58] to implement a data access layer for achieving an Object Relational Mapping (ORM) as the core abstraction from the relational database storage. The middle tier is divided into the three phases as described above. The top tier implements a Web-based user interface and provides a Web service API to allow external systems to send preprocessing requests to QUATSCH. The overall system architecture is depicted in Figure 4.1. In the following, we discuss the middle tier according to the three main phases and its components in detail.

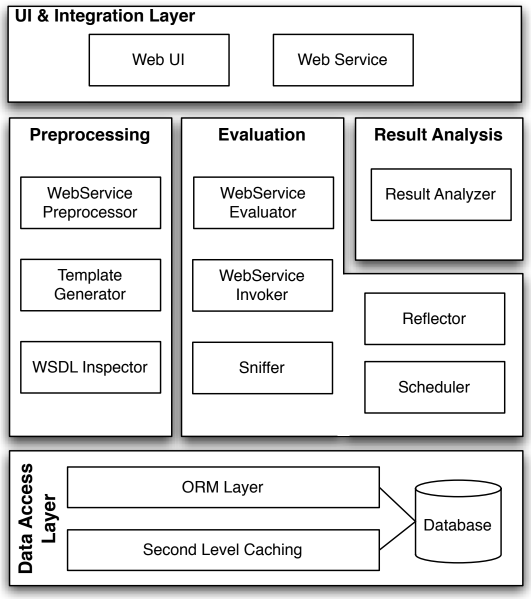


Figure 4.1: System Architecture

4.3.1.1 Preprocessing Phase

The preprocessing of Web services can be triggered either by using the Web-based UI or the Web service as well as the command-line tool. The preprocessing phase is encapsulated in the `WebServicePreprocessor` component. It reads the WSDL file and extracts all necessary information required for this step (by using the `WSDL Inspector` component). It parses the WSDL file to determine the SOAP binding name (we only support SOAP bindings). From the binding name, we can retrieve the corresponding `portType` element, thus, getting all operations which we have to evaluate. Furthermore, all XSD data types defined within the `types` element are parsed to retrieve all available types needed for invoking the service operations.

Evaluation Configurations. The user can configure the evaluation process by defining one or more evaluation configurations. An evaluation configuration defines the following items:

- a time frame when a service has to be monitored (beginning and end date);
- a monitoring interval as a Unix cron expression, e.g. `"0 0/10 * * * ?"` for monitoring every ten minutes;

- a name of the operation to evaluate or all operations (default);
- a service invocation template to define a set of valid parameters that are used when invoking a service, otherwise the default values for each data type are used;
- whether to use a standard evaluation or a throughput evaluation strategy (discussed below).

One service can have multiple evaluation configurations because different operations may require different evaluation schedules and individual invocation templates.

Generating Service Stubs. The most important step in the preprocessing phase is the generation of the Web service stubs to be able to invoke the services and monitor their performance. Most existing Web services mainly implement an RPC-style communication usually referred to as RPC-encoded style in SOAP. Unfortunately, this style is not in line with the document-centric idea of Web services, therefore, the document-literal style is now the de-facto standard style for provisioning SOAP-based Web services. The QUATSCH system has to handle both styles properly, however, existing Web service frameworks in the Java environment do not properly support both. Therefore, we use Apache Axis [7] for handling all RPC-style Web services and Apache CXF [6] for dealing with all document-style Web services. Using these two frameworks requires specific QoS measurement approaches. In the case of Axis, we leverage aspect-oriented programming (AOP) as a means to decouple the QoS measurement code from the Web service stub as produced by Axis WSDL2Java tool. The code weaving is done during the preprocessing. In case of CXF, we use a set of interceptors that can be plugged into the CXF framework to handle the QoS measurement. Irrespective of the framework in use, the stubs are dynamically compiled and bundled with all other related artifacts to build a JAR file that is stored in the database as BLOB (Binary Large Object). These concrete service evaluation mechanisms are explained in Section 4.3.2 and 4.3.3, respectively. All these steps are fully automated by the `WebServicePreprocessor` component.

Template Generation. Besides generating and compiling the service stubs, an invocation template can be generated to define template values to be used during the invocation. A template, in form of a sample SOAP request, is generated based on the XSD type definition in the WSDL file. For example, consider the following listing defining a simple XSD type for a conversion from Celsius to Fahrenheit.

```xml
<xs:complexType name="CelsiusToFahrenheit">
  <xs:sequence>
    <xs:element name="celsius" type="xs:double"/>
  </xs:sequence>
</xs:complexType>
```

A template for a service operation that has the above `CelsiusToFahrenheit` type as an input would look like this:

```
<CelsiusToFahrenheit>
  <!--type: double-->
  <celsius>36.3</celsius>
</CelsiusToFahrenheit>
```

At evaluation time, the `Reflector` component will use this template to set the corresponding values in the request message that is sent to the service. This template mechanism is useful especially for services that require certain data elements to be set with specific values. For example think of an application ID that is issued by the service provider and which has to be included in every message. There is simply no way to automatically detect the value of a certain field, therefore, user input is required to provide that specific information. Without templates, the `Reflector` can only use default or dummy data for each data element which may result is meaningless requests. This may lead to exceptions at the service provider and as a consequence it may falsify the measured QoS data (e.g., the response time might be much shorter than for a "real" request).

4.3.1.2 Evaluation Phase

In this phase, the basic task is to evaluate a Web service by invoking it and use the sniffer to capture the TCP trace that is generated by the invocation. The `WebServiceEvaluator` component encapsulates the evaluation logic and implements two evaluation strategies. Figure 4.2 shows the dependencies between the invoker and evaluation strategies in the QUATSCH architecture.

- **Default Evaluation Strategy**: It is used to evaluate a service based to its evaluation configuration by sending one evaluation request. Depending on the style (document-literal or RPC) supported by the provider, the `WebServiceEvaluator` creates the corresponding `WebServiceInvoker` to invoke the service. The `Sniffing` component, based on the Java packet capture library called Jpcap [53], is automatically triggered due to the fact that the corresponding code was previously added to the Web service stubs in the preprocessing phase. This way the response time, latency and other QoS attributes can be measured. Details on the sniffing approach are provided in Section 4.3.4.
- **Throughput Evaluation Strategy**: Basically, it has the same functionality as the default strategy with the major difference that it is used to perform a throughput evaluation based on the number of requests configured in the evaluation configuration. It uses a thread pool to send all requests in parallel and uses a synchronization barrier to wait for all responses.

In order to dynamically invoke a Web service, the `WebServiceInvoker` component uses reflection to instantiate the corresponding Web service stub. The operation to invoke a Web service depends on the evaluation configuration. Due to the fact that we use two different frameworks to invoke a service, we also need two different invocation strategies because the Web service stubs are totally different. Thanks to the flexible architecture, it is

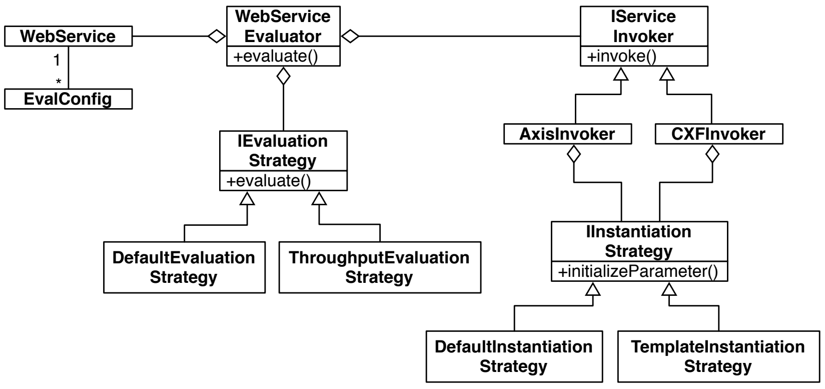


Figure 4.2: QUATSCH Class Diagram

very easy to plug in a concrete invoker. In QUATSCH, two different invokers are available, the `AxisServiceInvoker` and the `CXFServiceInvoker` both implementing the common `IServiceInvoker` interface (we commonly refer to this set of classes as `WebService-Invoker`).

Each `WebServiceInvoker` implements two different strategies that define how the parameter of a service operation are created. Firstly, the `DefaultInstantiationStrategy` uses the `Reflector` to instantiate each parameter of a service operation. These parameters are either simple types such as `string` or `integer` or complex types represented by classes that were generated by WSDL2Java tools of the corresponding Web service framework (Axis and CXF) as part of the preprocessing phase. Since these complex classes adhere to the Java-Bean standard they can be instantiated and assigned default values using the setter methods (such as 0 for `int`). Secondly, the `TemplateInstantiationStrategy` is used to instantiate the parameter of an operation by using the values defined in the previously generated invocation template. These values are extracted at runtime by using XPath. The algorithm for dynamically invoking a Web service operation with the `AxisInvoker` is shown in Listing 4.1.

The algorithm uses the Web service stubs and the template which have been generated during the preprocessing phase and tries to find a matching value for each parameter in the template file. If no parameters can be found in the template or the template is not available, the `Reflector` tries to instantiate the required parameter type with a default value. The invocation algorithm first has to check whether the service implements a given operation name (lines 3–6). Then, the algorithm iterates over all WSDL message parts (lines 14–25) and finds a corresponding type (either a simple or complex type) that is initialized by using one of the available `IInstantiationStrategy` implementations (line 24). At the end, the service method is invoked and the response is returned to the caller. Exception handling is omitted to enhance readability.

```java
public Object invoke(EvaluationConfig ec, ClassLoader cl) throws ServiceInvocationException {
  // find the given operation in the service
  Operation operation = _service.getOperation(ec.getOperationName());
  if (operation == null) {
    throw new ServiceInvocationException(String.format("Operation '%s' does not exist!", ec.getOperationName()));
  }
  initClasses(cl); // initialize all stub classes and helpers
  Message inputMsg = operation.getInput().getMessage();
  Map parts = inputMsg.getParts();

  Class[] parameters = new Class[parts.size()];
  Object[] paramInstances = new Object[parts.size()];

  List<Part> orderedParts = inputMsg.getOrderedParts(null);
  for (int i = 0; i < orderedParts.size(); i++) {
    Part p = orderedParts.get(i);
    QName type = p.getTypeName();
    Class<?> param = WSDLInspector.convertXSDTypeToJavaType(type.getLocalPart());
    if (param == null) { // it is not a simple type
      String javaName = WSDLUtils.convertQNameToPackageName(type);
      param = Class.forName(javaName, false, cl);
    }
    parameters[i] = param;
    paramInstances[i] = _strategy.initalizeParameter(ec, p.getName(), param);
  }
  Method serviceMethod = _stubClass.getMethod(Utils.xmlNameToJava(ec.getOperationName()), parameters);
  return serviceMethod.invoke(_stub, paramInstances);
}
```

Listing 4.1: Dynamic Invocation Code

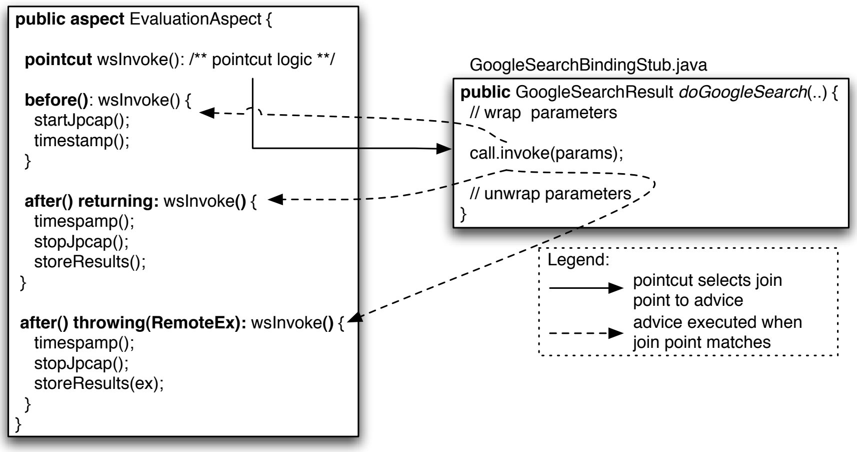
Service Scheduling. Finally, the remaining part is the periodic execution of the service evaluation to ensure continuous monitoring. The `Scheduler` component implements a cron scheduler which invokes a service according to its settings specified as part of the evaluation configurations that are associated with a service. It periodically examines all service entries in the database to detect new services added by the client. If a new service is added, the evaluation configuration is inspected, a service trigger is constructed and a job is added to the scheduler. The scheduler then autonomously reacts to firing triggers by executing the associated job. We use the popular Quartz framework [31] to implement our scheduling component. Additionally, there is a command-line tool to evaluate a service just once which is intended to be used for testing.

4.3.1.3 Result Analysis Phase

The final phase is the result analysis which is totally decoupled from the evaluation itself. The results generated from the `WebServiceEvaluator` are stored in the relational database by using the ORM layer. The `ResultAnalyzer` component is implemented as a scheduling job and it iterates over the collected results and generates the necessary statistics for each QoS attribute (e.g., calculating the average response time per hour, day, etc.).

4.3.2 AOP-based Evaluation

In order to decouple QoS measurement logic from the concrete Web service stub code, an AOP approach is used for RPC-style services based on the Axis framework. AOP represents an ideal technique for modeling cross-cutting concerns. QoS measurement is such a cross-cutting concern as it has to be applied to each RPC-style service which has to be invoked during the evaluation. In order to weave the measurement code into the service stubs generated by the Axis WSDL2Java, we have to identify the corresponding join points in the stub code. The basic synopsis of our approach is described in Figure 4.3. During the preprocessing phase, we generate the stubs for the service which should be evaluated by using the WSDL2Java tool. For the Google Web service, as one example, the main stub class that is generated is called `GoogleSearchBindingStub`. Each stub method looks similar, first is the wrapping phase, where the input parameters are encoded in XML. Secondly, the actual invocation is carried out by using the `invoke(..)` method of the `Call` class from the Axis library. At last, the response from the service is unwrapped and encoded as Java arguments and returned to the caller.



```
public aspect EvaluationAspect {

  pointcut wsInvoke(): /** pointcut logic **/

  before(): wsInvoke() {
    startJpcap();
    timestamp();
  }

  after() returning: wsInvoke() {
    timespamp();
    stopJpcap();
    storeResults();
  }

  after() throwing(RemoteEx): wsInvoke() {
    timespamp();
    stopJpcap();
    storeResults(ex);
  }
}
```

```
GoogleSearchBindingStub.java
public GoogleSearchResult doGoogleSearch(..) {
  // wrap  parameters

  call.invoke(params);

  // unwrap parameters
}
```

Legend:
pointcut selects join point to advice
advice executed when join point matches

Figure 4.3: Aspect for Service Invocations (simplified)

The join point for measuring the response time q_{rt} is the `invoke(..)` method of the Axis `Call` class. Therefore, we define the following pointcut in the `EvaluationAspect` as depicted in Listing 4.2 that has to match that join point at runtime.

Whenever a service operation is invoked by using the `WebServiceEvaluator` component, the `wsInvoke()` pointcut defined for this join point is matched. Before the actual service invocation, the `before` advise is executed. This is where the actual evaluation has to be carried out. It mainly consists of a timestamp and the generation of an `EvaluationResult`,

```
pointcut wsInvoke(): target(org.apache.axis.client.Call) &&
                     ( call( Object invoke(..)) || call( void invokeOneWay(..)) );
```

Listing 4.2: Response Time Pointcut for Axis

as well as starting the packet sniffer using our `Sniffing` library to trace the TCP traffic caused by this request.

After the `wsInvoke()` pointcut the execution of the corresponding `after` advise is triggered. In case of a successful execution of the service invocation, the `after() returning` advise is selected otherwise the `after() throwing` advise is invoked. At this point, the packet sniffing is stopped and the TCP trace data is collected. The timestamps taken before and after the invocation can directly be used to calculate the response time. For the calculation of the latency and execution time, the TCP trace needs to be analyzed which is described in detail in Section 4.3.4.

4.3.3 Interceptor-based Evaluation

Apache CXF provides a framework for provisioning and consuming document-centric Web services as well as REST services. When invoking a Web service, a number of interceptors are used to perform specific tasks such as writing the XML request to the binary stream or reading the binary response from the stream. The CXF framework uses the well-known interceptor pattern to provide these functionalities. These interceptors are managed by different interceptor chains, such as an incoming or outgoing chain. These chains are divided into a number of phases such as *RECEIVE* to handle the transport-level processing on the incoming interceptor chain. Our approach leverages these incoming and outgoing interceptor chains to add custom interceptors, for example to set the corresponding timestamps and run the TCP sniffing process. In the outgoing chain, the following interceptors are added to the corresponding phases:

- `SETUP` phase: In this first phase we add a `QoSInitInterceptor` to set up all the data structures required for the monitoring and set timestamps for the round trip time.
- `PREPARE_SEND` phase: This phase opens the connection to the endpoint of the service thus we add a `QoSSendInterceptor` to initialize and start the `Sniffing`.
- `PRE_STREAM` phase: This phase is used for preparing the stream level processing thus we add a `RequestMessageInterceptor` to retrieve the raw XML message that is sent to the server for storing it with the evaluation data.

After these phases are processed and the request is finally sent to the service provider, the `Sniffing` component is listening and capturing all packets between the sending and receiving host. When the response is received, an incoming chain is created by the Apache CXF framework where we have to add some additional interceptors to set the corresponding timestamps, stop the sniffing component and store the results. In the incoming phase, we add the following interceptors:

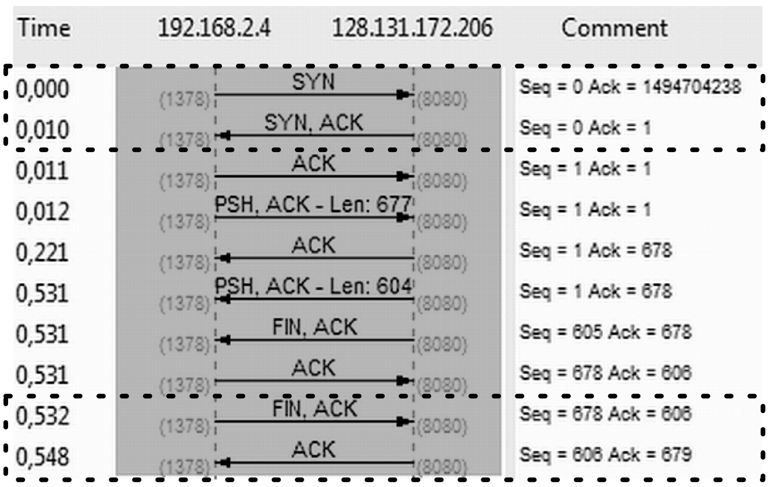
- `RECEIVE` phase: This phase is executed as the first one right after the response was received thus we add a `QoSReceiveInterceptor` to record a timestamp for calculating the response time and retrieve the data from the session data.
- `POST_STREAM` phase: This phase is used for post stream-level processing thus we add a `ResponseMessageInterceptor` to retrieve the raw XML message that was received from the server for storing it with the evaluation data.
- `PRE_INVOKE` phase: Before the local service method is invoked with the response, the `QoSResultInterceptor` is invoked to stop the `Sniffing` component and store the results in the database.

4.3.4 TCP Sniffing and Reassembly

The core element of this approach is the combination of automated and client-side QoS monitoring with low-level TCP sniffing and traffic analysis. This enables the approximation of the server-side response time and the calculation of the network latency as two important QoS attributes. A typical Web service invocation on the API-level is a simple invocation of the corresponding stub method, however, on a TCP level we can distinguish three different phases. The first phase is the connection establishment following the three-way handshake to set up a TCP connection [140]. The second phase handles the data transmission itself (i.e., the payload from the client to the server and the response). The final phase is the TCP connection termination.

The three-way handshake is a simple way of establishing a connection between two hosts. In this case, the client sends a SYN packet to the server which responds with a SYN and an ACK package to acknowledge reception. Finally, the client sends an ACK to the server to acknowledge the SYN and the connection is established. The connection termination is very similar but actually not three-way, instead it is a pair of two-way handshakes. The end of the connection is triggered by one of the hosts by sending a FIN and the recipient sends back an ACK. The other party also sends a FIN that is ACKed by the other side. Each of those handshake packets does not have a payload thus they are a very good indicator of the network latency. They only need the time to reach its destination plus some negligible timespan the operation system requires to create an ACK packet and send it back. Therefore, we can extract at least two meaningful network latency values. A full TCP trace of a Web service invocation in the local network is shown in Figure 4.4. The trace was generated by the packet capturing tool Wireshark [169]. The dashed boxes indicate those parts that are used to calculate the latency values.

Unfortunately, calculating the execution time from a TCP trace is not as easy as it appears from the trace in Figure 4.4. Consider a simple Web service request where a SOAP message



Time	192.168.2.4		128.131.172.206	Comment
0,000	(1378)	SYN →	(8080)	Seq = 0 Ack = 1494704238
0,010	(1378)	← SYN, ACK	(8080)	Seq = 0 Ack = 1
0,011	(1378)	ACK →	(8080)	Seq = 1 Ack = 1
0,012	(1378)	PSH, ACK - Len: 677 →	(8080)	Seq = 1 Ack = 1
0,221	(1378)	← ACK	(8080)	Seq = 1 Ack = 678
0,531	(1378)	← PSH, ACK - Len: 604	(8080)	Seq = 1 Ack = 678
0,531	(1378)	← FIN, ACK	(8080)	Seq = 605 Ack = 678
0,531	(1378)	ACK →	(8080)	Seq = 678 Ack = 606
0,532	(1378)	FIN, ACK →	(8080)	Seq = 678 Ack = 606
0,548	(1378)	← ACK	(8080)	Seq = 606 Ack = 679

Figure 4.4: TCP Message Flow

is sent over HTTP to the service provider. In the simplest case, there is one message on the TCP level where the message is transmitted to the server. However, in practice multiple cases exist that have to be considered when calculating the latency and approximating the execution time of a service operation on the server. Firstly, the request or the response may have a large message size thus exceeding the maximum TCP frame length. This will result in a multi-frame transmission or an update in the fragment size. Due to the fact that multiple messages are transferred over the wire, it is hard to determine when the service started the execution of a service operation that is need to determine the execution time. Secondly, TCP packets can get lost or the connection gets interrupted. In these cases, TCP automatically retransmits lost packets. The obsolete packets must not be used to calculate network latency.

In order to address these issues, we have designed a simple TCP Trace Analysis algorithm as depicted in Algorithm 1. This algorithm separates the packets in the TCP trace T into two separate maps. The src and $dest$ maps store the client and server packets indexed by the sequence number of the package (client) and acknowledgment number (server), respectively. By using map data structures with indizes to associate sequence and acknowledgment numbers to TCP packets, multi-frame transmissions, retransmissions and window size updates are transparently handled because they are mapped to the same index in the corresponding map. A loop processes all the packets and stores them either in the src or $dest$ map (lines 4–14).

After splitting the TCP trace in two different maps, they can be analyzed as shown in lines 16–23. For each source packet in the src map, it tries to find a packet in the destination map that matches the acknowledgment number (by using the sequence number of the source packet + 1). For each match, we can calculate a timespan value between two messages exchanged between the client and server. This value is added to the $timestamps$ list. As an example consider the first and the second message in Figure 4.4. The timestamp on the client says 0,010 seconds, thus it is a pretty good measure for the latency (in fact it is twice the latency because a packet was sent from the client to the server and back). The timespan after sending the payload to the service and receiving the response payload is assumed to be the execution time q_{ex} on the server. The function $calcExecutionTime$ is used to calculate the execution time by using the maximum value in the list (variable $maxExecTime$) and subtracting the arithmetic mean of all latencies to approximate the execution time of a service (line 25).

Algorithm 1 TCP Trace Analysis

```
1:  function ANALYSETRACE(TCP trace $T$)
2:      $src \leftarrow \emptyset$                                  ▷$src$ and $dest$ are map structures
3:      $dest \leftarrow \emptyset$
4:      for all $p_i \in T$ do
5:          if $p_i$ is outgoing then
6:              $seqNr \leftarrow p_i.seqNr$
7:              if $seqNr \notin src$ OR $src[seqNr]$ has no payload OR $p_i$ has no payload then
8:                  $src[seqNr] := p_i$
9:              end if
10:         else[$p_i$ is incoming]
11:             $ackNr \leftarrow p_i.ackNr$
12:             $dest[ackNr] := p_i$
13:         end if
14:     end for
15:     $timestamps \leftarrow \emptyset$                ▷a list of timespans for the TCP packet transmissions
16:     for all $sp_i \in src$ do
17:         $seqNr \leftarrow sp_i.key$
18:         $key \leftarrow seqNr + sp_i.payloadLength$
19:         if $dest[key]$ != null AND $sp_i.payloadLength > 0$ then
20:             $execTime \leftarrow (dest[key].timestamp - sp_i.timestamp)/2$
21:             $timestamps \leftarrow timestamps \cup execTime$
22:         end if
23:     end for
24:     $maxExecTime \leftarrow \max(timestamps)$                     ▷get max element from list
25:     return calcExecutionTime($maxExecTime, timestamps$)
26: end function
```

TCP Packet Correlation. When using the aforementioned TCP trace analysis approach, we have to ensure that all captured TCP packets belong to the same trace (i.e., the source and destination host have to be the same). Jpcap, the base library for our `Sniffing` component, supports filters to restrict the packet capturing device to a certain host and port. However, these filters are not enough since there can be multiple service invocations to the same host at the same time, e.g., during throughput testing. In order to distinguish multiple parallel invocations, we use the local port assigned by the operating system (OS) as a means to distinguish between multiple traces to the same destination host and port. The locally assigned port is an ideal correlation identifier as it cannot be influenced and the OS ensures that it is unique on the client machine.

Besides distinguishing multiple packets routed to the same service, we need to ensure that we have a unique correlation of a TCP trace on the packet capturing level with the corresponding service invocation on the application level (the `WebServiceInvoker` component). Whenever a service is evaluated, a dynamic filter (specifying host and port) needs to be added to the `Sniffing` component, however, the locally assigned port that is used as a correlation identifier is not known at that time (since no TCP packets have been transferred so far). Therefore, we use one semaphore for each unique host/port combination to lock the critical section until the first packet was captured and the correlation identifier can be extracted. This semaphore ensures that only one service evaluation call can add a filter for the same host/port at the same time thus ensuring proper correlation.

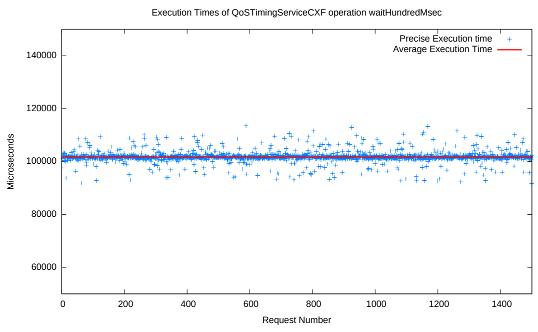
4.3.5 Implementation Aspects

The QUATSCH approach has been implemented on top of the Java 1.6 platform using AspectJ to implement the evaluation aspects for Axis-based stubs. The data access layer has been implemented using the Hibernate framework, a popular ORM framework for the Java platform. The sniffing part leverages Jpcap, which provides a Java wrapper for libpcap, a packet capturing library available on most platforms (Linux, OS X and also as a Windows port called Winpcap [168]). The UI has been implemented using Grails 1.1 [127], a lightweight Web framework based on the Groovy programming language. The reason for choosing Grails was its seamless integration with the Java platform, thus enabling an easy access and reuse of the QUATSCH core components. Grails also acts as the base library for implementing the QUATSCH Web service to allow a remote management and integration with other tools. QUATSCH is the core QoS monitor for the VRESCO SOA runtime described in Chapter 6.

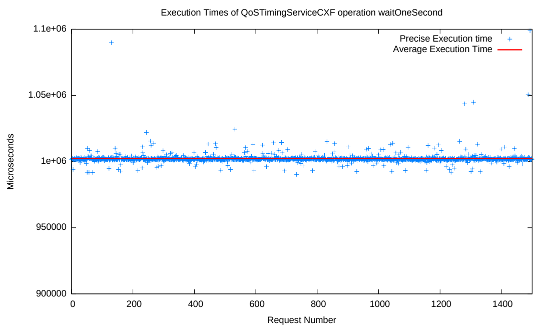
4.4 Evaluation

In order to evaluate our approach, especially the approximation of the Web service-specific execution time, we have developed two distinct Web services. The `QoSTimingServiceRPC` implements an RPC-encoded service using the Axis toolkit, whereas the `QoSTimingService-CXF` implements a document-literal style service using Apache CXF. Each Web service has several timing-specific operations, such as `waitHundredMsec` which waits exactly 100 msec before sending a response. By using such special-purpose Web services, we get a better idea of the accuracy of our server-side execution time estimation. All Web services used during the experiments were hosted on a dedicated server machine (Dell Blade 1955 with a Dual Core Xeon CPU 3.2 GHz and 10 GB RAM). All Web services have been deployed on the respective Web service container running on a Tomcat 6 [146] without any special configuration or tuning. The QUATSCH toolkit is installed in a VMware Virtual Machine with an Ubuntu Linux operating system with 1 GB RAM also hosted on a Dell Blade with the same configuration as the one mentioned above. For measuring the QoS of the `QoSTimingServiceCXF` and the `QoSTimingServiceRPC` service, we have chosen two operations, `waitHundredMsec` and `waitOneSecond` and created an evaluation configuration for each operation to continuously measure it every 5 minutes. Over a longer period of time, we collected approximately 10000 test runs for each operation.

In Figure 4.5, the server-side execution times of the `QoSTimingServiceCXF` service are depicted. In all plots, we have chosen a sample of 1500 requests on the x-axis and the execution time on the y-axis (in μs). Figure 4.5a shows the execution time for `waitHundredMsec` operation. The red line marks the average execution time of 101715 μs over all measured values, whereas the blue points show each measured execution time value. The standard deviation is 2338 μs. Figure 4.5b shows the same data for the `waitOneSecond` operation. In this case the average execution time is 1002182 μs and the standard deviation is 4498 μs. These measured values are very close to the known execution times of 100000 μs and 1000000 μs, respectively. In fact, the exact execution time of our testing services are slightly over the proposed value, since we use a `Thread.sleep()` to simulate the execution time which has some additional minor overhead. Overall, we have an approximation error which is less than three percent in both cases, therefore, our approach is accurate in measuring the execution time from a black-box view.



(a) Operation `waitHundredMsec`

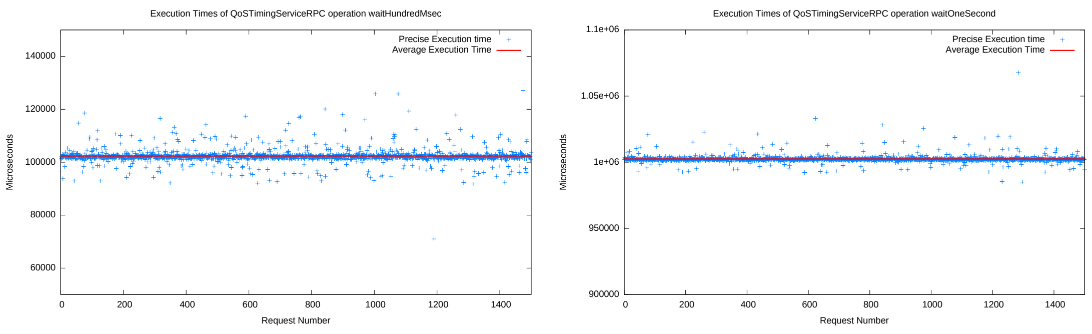


(b) Operation `waitOneSecond`

Figure 4.5: Execution Time Approximation for the `QoSTimingServiceCXF` Service

Besides the document-literal based service as shown above, we have executed the same set of tests for an RPC-encoded Web service. It is important to run the same tests for both types because we use different service frameworks (Axis and CXF) and thus also different techniques (AOP vs. interceptors) to evaluate the services. In Figure 4.6, the results for both operations are depicted. The results are very similar as for the document-literal services. Figure 4.6a shows the values for the `waitHundredMsec` operation with an average execution time of 102171 μs and a standard deviation of 2261 μs. Figure 4.6b shows the values for the `waitOneSecond` operation with an average execution time of 1002530 μs and a standard deviation of 6501 μs.

According to our definition of response time in Chapter 3, it includes the network-specific



(a) Operation `waitHundredMsec`   (b) Operation `waitOneSecond`

Figure 4.6: Execution Time Approximation for the `QoSTimingServiceRPC` Service

time. As shown before, the approximation of the service-side execution time is very accurate, however, we now use a real-world service to measure the execution- and response time where the execution time is not predictable. To do so, we have monitored an `ISBNCheck` Web service[1] to show the differences between execution and response time. The invocation leverages an invocation template with real ISBN values. The data is shown in Figure 4.7, displaying the monitoring dates (grouped per day) on the x-axis and the measured time in μs on the y-axis (average per day). We can see that the execution times of both operations (`IsValidISBN10` and `IsValidISBN13`) are pretty stable. The difference between the execution and response time is basically the network latency and the wrapping time (which we ignore as we cannot explicitly measure it on the server-side). Based on the observation that the response time varies during the observation period, we can argue that this due to the changing network latency. However, it can also be observed that the execution time is pretty stable. Based on the above mentioned results for the accuracy of our execution time estimation, we can assume that the real execution time does not differ much from our calculations.

Throughput Measurements. Measuring the throughput from the client-side perspective can only provide a snapshot of the possible throughput that a Web service may have. A main drawback is that it cannot simply be measured constantly, since this could overload the target Web service and influence normal operations for other customers. However, an approximation of the throughput value can help to make decisions about the operations that can be processed by a service and its scalability. In Figure 4.8, the number of throughput values for the operations `waitHundredMsec` and `waitFiftyMsec` of the `QoSTimingServiceRPC` service are depicted. All experiments were executed on the same hardware as the other tests above.

In particular, the x-axis depicts the number of requests ranging from 250 to 2000 (in steps of 250), whereas the y-axis depicts the operations per second (OPS) for each service. The

[1]http://webservices.daehosting.com/services/isbnservice.wso?WSDL

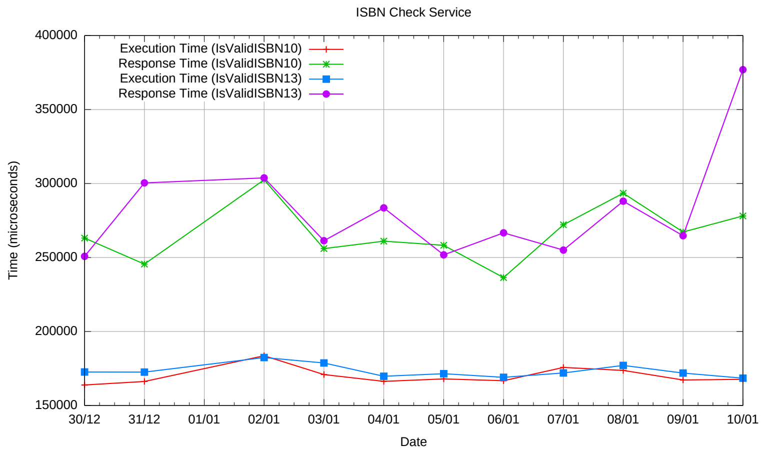


Figure 4.7: Execution vs. Response Time of the `ISBNCheck` Service

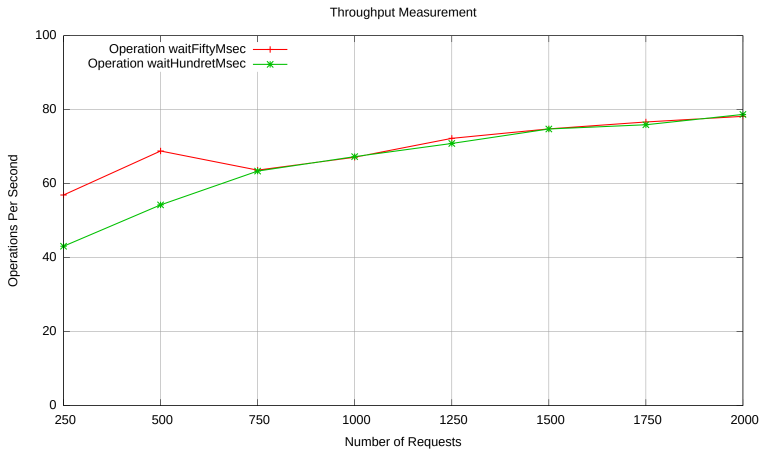


Figure 4.8: Throughput Evaluation of `QoSTimingServiceRPC`

graphs depict the two different operations. When analyzing the results, one can see that the `waitHundredMsec` scales up to 2000 parallel requests. In contrast, the `waitFiftyMsec` starts at a higher throughput up to 500 parallel requests but then it has a drop-off to almost the same throughput as the `waitHundredMsec`. The most obvious reason is the internal queuing and processing capacities of the Tomcat server. Up to 750 parallel requests they are processed in parallel resulting in higher throughput values for the `waitFiftyMsec` operation because it executes in half of the time as the other operation. From 2000 parallel request and up, we observed that the throughput was going down and the success rate dropped (not shown in Figure 4.8).

Tool Support. In order to enable a simpler use of the QUATSCH system, we have build a Web-based UI to facilitate common tasks such as adding a service, removing a service, managing evaluation configurations, etc. An important feature is a dynamic chart generator which allows users to dynamically build a chart containing various QoS attributes of a given Web service. Two screenshots illustrating the UI are shown in Appendix A.

4.5 Discussion and Limitations

Although the results of the proposed monitoring approach are satisfying, there are a number of issues and limitations that need to be considered when using and implementing a client-side evaluation strategy. First and foremost are scalability issues. Whenever a service needs to be monitored, a decision has to be made how often a service needs to be monitored and whether throughput testing needs to be performed. The number of monitoring requests can cause a serious network overhead when monitoring numerous services at a constant and short interval (e.g., every minute). Additionally, the monitoring results can be deformed when the monitoring interval is very low because these continuous monitoring requests can cause a higher load on the service. Whenever such a high-number of service monitoring requests is required, it might be better to propose an integrated monitoring approach without the need to continuously send monitoring data over the network.

Another important aspect in the Web service area are pricing issues. Whenever a public service should be monitored, it is important to ensure that the number of monitoring requests is not very high or an agreement can be made that monitoring requests are not billed like regular requests. However, general approaches to distinguish ordinary requests from a monitoring requests are currently not available, though it would be possible to leverage SOAP headers to contain the relevant information. Another approach would be to combine a client-side and server-side monitoring to reduce the number of monitoring requests. The server-side monitoring component can provide monitoring data that can then be combined with data from the client-side approach (e.g., for measuring the latency). This way it would also be possible to verify monitoring data advertised by the service-provider. Finally, a major issue and limitation is the use of throughput measurement. Especially for public services, throughput measurement as carried out by QUATSCH cannot be used because this would be identified as denial-of-service attack by the provider when sending 1000 parallel requests at the same time. As a consequence, a client might be permanently blocked thus being unable to consume the service any longer.

Chapter 5

Transformation of SLA-Aware Choreographies into Orchestrations

This chapter introduces an approach to engineer service-oriented systems, in particular business processes, in a top-down manner by defining a choreography and automatically generate the orchestrations for each partner. Additionally, SLAs can be associated with choreographies to facilitate a transformation to enforceable QoS policies on the orchestration layer.

Contents

5.1 Motivation

The integration of QoS concerns and service level guarantees among various partners are an important aspect in early stages during the development of adaptive service-oriented systems. It enables a top-down modeling by associating higher-level SLA requirements and guarantees with choreographies and provide a semi-automated transformation to orchestrations stubs and WSDL artifacts for each partner. These SLA requirements in the choreography are then automatically transformed and mapped to enforceable QoS policies on the orchestration layer (macroflow) and attached to the BPEL process of the corresponding partner. A policy-aware middleware can then enforce these policies and react appropriately if the actual QoS does not comply with the specified policy. A main reason for mapping SLAs to enforceable QoS policies is based on the fact that SLAs are not directly monitorable at the process execution level. Moreover, a monitoring component for SLA achievement would have to be provided separately for each SLA dialect and BPEL engine. A transformation to enforceable QoS polices in a standardized language (such as WS-Policy) reduces the burden of implementing a custom component for each BPEL engine. Most existing engine provide support for WS-Policy, thus requiring only a custom policy enforcement module.

The proposed approach leverages the Web Service Choreography Description Language (WS-CDL) [161] as one of the first publicly available choreography languages targeted to Web services. It provides a XML-based language to describe the cross-organizational message exchanges from a global viewpoint. It promotes a top-down design approach for efficient development of cross-organizational business processes similar to the idea of model-driven software development (MDSD) [156]. WS-BPEL [107] is used as the orchestration language since it is widely accepted as the de-facto standard language for describing Web service based orchestrations (macroflows). Both languages are briefly introduced in Section 5.3. Furthermore, different policy-aware BPEL runtimes are available such as ActiveBPEL [1].

5.2 Illustrative Example

In order to illustrate the top-down modeling concepts and the proposed QoS integration, we use a simplified *Build-to-Order* (BTO) scenario from the supply chain domain as illustrated in the sequence diagram in Figure 5.1. The use case consists of a customer, a manufacturer, and suppliers for CPUs, main boards and hard disks. The manufacturer offers assembled IT hardware equipment to its customers. For this purpose, the manufacturer has implemented a BTO business model. It holds a certain number of individual hardware components in stock and orders missing components if necessary. In the implemented BTO scenario, the customer sends a quote request with details about the required hardware equipment to the manufacturer. The latter sends a quote response back to the customer. As long as the customer and the manufacturer do not agree on the quote, this process will be repeated. If a mutual agreement was achieved the customer sends a purchase order to the manufacturer. Depending on its hardware stock, the manufacturer has to order the required hardware components from its suppliers. If the manufacturer needs to obtain hardware components to fulfill the purchase order, an appropriate hardware order is sent to the respective supplier. In turn the supplier sends a hardware order response to the manufacturer. Finally, the manufacturer sends a purchase order response back to the customer.

Most parts of the choreography from Figure 5.1 can be fully specified by using WS-CDL,

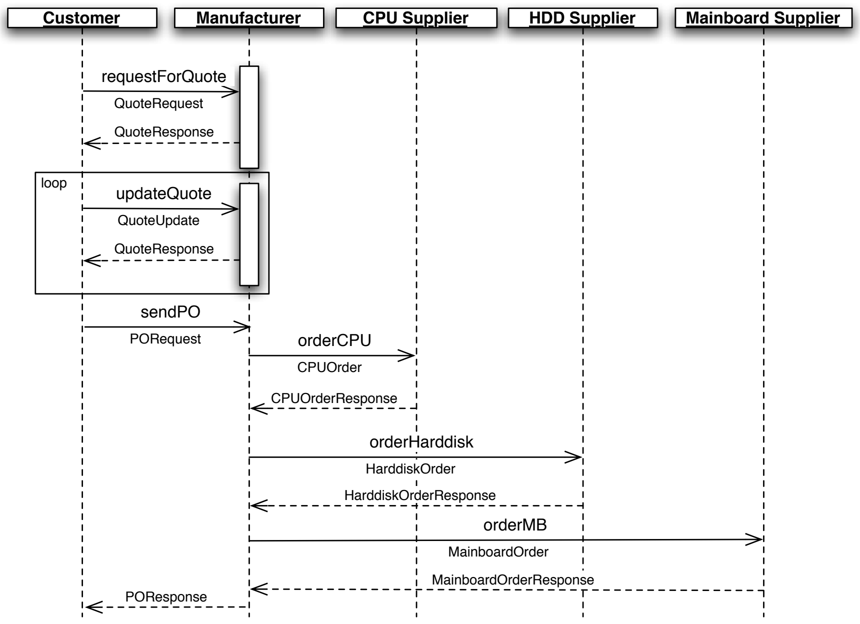


Figure 5.1: BTO Case Study

however, some non-observable behavior that is internal to a specific partner cannot be modeled. Therefore, these parts are declared as silent actions in WS-CDL to express that some internal logic has to be implemented in the orchestration layer (the BPEL code). For example, the internal quote processing at the manufacturer has to be specified as silent action because it does not represent globally observable behavior, however, it needs a refinement in the BPEL process to specify whether a quote is accepted or rejected.

The definition of SLAs and QoS plays a crucial role in cross-organizational business processes. Each participant offers services to other partners over the Internet which the latter need to run their businesses. Therefore, a certain degree of reliability concerning response time, throughput, uptime, etc. is desired and has to be specified and explicitly expressed from the beginning of the modeling phase. In our scenario, we distinguish four different relationships between the choreography participants. The customer interacts with the manufacturer, the manufacturer interacts with different suppliers. For each relationship, an SLA is defined between the partners to regulate the non-functional and contractual issues for their interactions.

5.3 Background and Basic Concepts

In this section, we briefly introduce the two basic technologies that we use in our approach including some illustrating examples.

5.3.1 An Overview of WS-CDL

WS-CDL represents a non-executable XML-based specification language allowing each involved party to describe its part in the message exchange by specifying details on collaborations, information handling and activities. In the following paragraphs, we introduce the basic concepts by using the case study from Figure 5.1 to illustrate some WS-CDL examples.

5.3.1.1 Collaborations

The collaborations of a choreography are specified by defining participant types, role types, relationship types and channel types. These four types are used to define the collaborating participants and their coupling. A participant type (element `participantType`) declares an entity playing a particular set of roles in the choreography. Thus a participant type definition contains one or more role type definitions. A role type (element `roleType`) defines a role that enumerates the observable behavior a participant can exhibit in order to interact throughout a message exchange. A role type definition declares a behavior interface which identifies a WSDL interface type. The relations between roles are defined through relationship type definitions (element `relationshipType`). A relationship type always contains exactly two role types, restricting the relationship type definition to 1:1 relations. A channel type definition (element `channelType`) specifies where and how information between participants is exchanged by defining a reference to a role type which is the target of an information exchange (either the receiver of a message request or the sender of a message reply). This role type reference indicates the behavior interface which is used throughout the information exchange.

5.3.1.2 Information Handling

The definition and handling of information within a choreography is performed by declaring information types and variables. On the one hand, information used within a choreography is specified by defining an `informationTypes` element which does not directly reference data types but rather reference type definitions. Such a reference type definition can be either a WSDL 1.1 Message type, an XML Schema type, a WSDL 2.0 Schema element or an XML Schema element. On the other hand, variables capture information about objects in a choreography such as the information exchanged or the observable information of the role types involved and are either bound to information type or channel type definitions.

5.3.1.3 Activities

A choreography comprises three different types of activities, namely ordering structures, work-units, and basic activities.

Ordering structures are block structured, enclosing a number of activities or ordering structures which can be used recursively. Such activities include `sequence` for handling activities in sequential order, `parallel` for a parallel execution of activities, and `choice` for handling data or event-driven conditions.

Workunits prescribe the conditional execution of an activity. This conditional execution can either be repetitive (attribute `repeat` is set to true), competitive (multiple workunit activities are defined inside a choice activity) or blocking (attribute `block` is set to true). The conditional statement is defined by the attribute `guard` which specifies a boolean conditional expression according to the XPath 1.0 lexical rules. In Listing 5.1, an example with competitive guard conditions from our case study is depicted. If there are no CPUs in stock, they are ordered from the supplier, otherwise available CPUs are selected.

```
<choice>
  <workunit name="Choice_CPUNotInStock" guard="cdl:getVariable('CPUInStock','','')&
      gt;0">
    <!-- select available CPUs -->
  </workunit>
  <workunit name="Choice_CPUInStock" guard="cdl:getVariable('CPUInStock','','')=0">
    <!-- order CPUs from supplier -->
  </workunit>
</choice>
```

Listing 5.1: Workunit Example

Basic activities define interactions, actions or variable assignments of the choreography flow. An *interaction* activity defines the information to be exchanged and by what means this information exchange will be performed. The attribute `channelVariable` binds the interaction to a `channelType` and, therefore, to a specific WSDL interface. The attribute `operation` corresponds to a SOAP operation which is defined throughout this WSDL interface description. The element `participate` defines the requesting and receiving part of the interaction. Finally the element `exchange` defines whether the interaction is a request or response and which variables will be used throughout the message exchange. Listing 5.2 illustrates an interaction activity defining a message request from our case study. Throughout the message request, the operation `requestForQuote` will be invoked at the corresponding WSDL interface of the `ManRoleType` to request a quote from the manufacturer. The message request is stored in the variable `QuoteRequest`. The response from the `ManRoleType` has to be modeled as another `interaction` (not shown in Listing 5.2).

```
<interaction channelVariable="tns:QuoteChannelInstance"
  name="RequestForQuote" operation="requestForQuote">
  <participate fromRoleTypeRef="tns:CustRoleType"
    relationshipType="tns:CustMan" toRoleTypeRef="tns:ManRoleType"/>
  <exchange action="request" name="request"
    informationType="tns:QuoteRequest" >
    <send variable="cdl:getVariable('QuoteRequest','','')"/>
    <receive variable="cdl:getVariable('QuoteRequest','','')"/>
  </exchange>
</interaction>
```

Listing 5.2: Interaction Activity

The other basic activities include `assign`, `silentAction` and `noAction`. The `assign` activity enables the creation and manipulation of variables within the choreography. The `silentAction` defines a non-observable behavior which is either performed by one or all participants in the choreography. A `silentAction` has to be further defined in the orchestration layer e.g., in the BPEL process of the corresponding participant.

A WS-CDL tool suite from Pi4soa [122] is available to facilitate choreography modeling without the need to write the XML representation directly. We also used it to model our case study.

5.3.2 An Overview of WS-BPEL

WS-BPEL (or BPEL for short) defines a model and grammar for describing the behavior of a business process based on interactions between the process and its partners. A BPEL process defines how multiple service interactions with partners are coordinated from the perspective of one partner [107]. It is important to understand that there is no global view on the messages exchanged between the partners.

Each partner interacting with a BPEL process is defined using a `partnerLink`. Two different roles (`myRole` and `partnerRole`) exist for a partner link to define the sending and receiving side of the process. The basic element in a BPEL process is an activity which comes in two flavors, *basic* and *structured* activities. Basic activities mainly define communication primitives for interacting with partners. For example, `invoke` to invoke a partner service, `receive` to receive a Web service invocation in a synchronized scenario. The `reply` activity is used to send a response message to a previously received Web service invocation message. Other basic activities include `onMessage`, `assign` and `empty`.

Additionally, *structured activities* are similar to control-flow constructs in imperative programming languages. In BPEL, a `sequence` activity is used to execute a given set of activities within a sequence. Parallelism can be achieved by using the `flow` activity. The `while` and `switch` activities are used to represent loops and conditional branches respectively.

The execution of a BPEL process is achieved using an orchestration engine, such as ActiveBPEL [1] and the Microsoft Windows Workflow Foundation [96].

5.4 Transformation and QoS Integration Approach

Based on the discussion of the preliminary techniques, we present a detailed description of the transformation and SLA/QoS integration approach for choreographies to orchestration stubs for each partner. Additionally, we focus on the generation of WSDL artifacts from the choreography.

5.4.1 Overview

The language constructs of WS-CDL can be mapped to BPEL allowing a choreography description to be automatically transformed into separate BPEL processes, one for each partner in the choreography, including the corresponding WSDL artifacts. It is important to understand that these generated artifacts are stubs that act as a starting point for the individual implementation of each partner and can then be executed directly after finalizing the implementation.

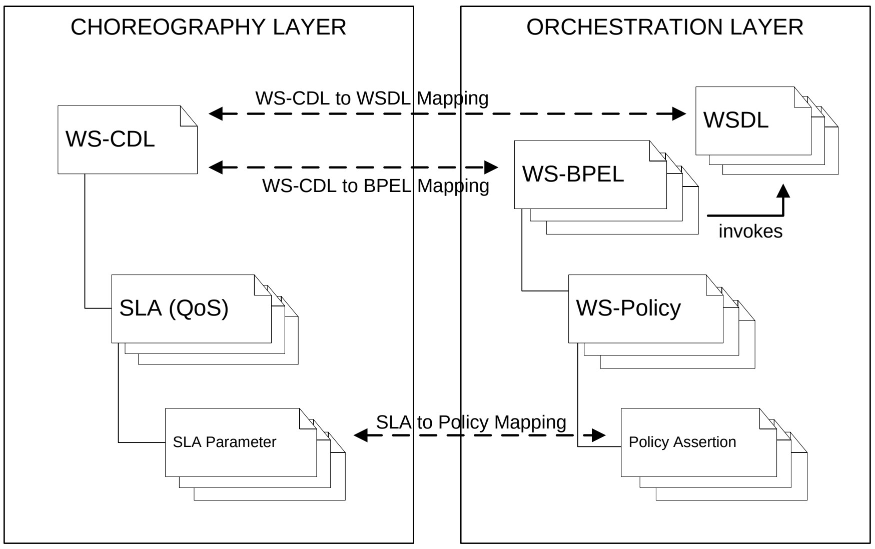


Figure 5.2: Modeling and Transformation Approach

Figure 5.2 shows a general overview of our mapping and transformation approach by depicting the basic models and artifacts. The choreography layer (on the left-hand side) shows the WS-CDL choreography description and the associated SLAs for each partner. Each SLA can define a number of SLA parameters specifying guarantees and obligations. The orchestration layer (on the right-hand side) shows the WS-BPEL orchestrations and the referenced WSDL artifacts that can be generated from the WS-CDL for each participant on the orchestration layer. Additionally, the WS-Policy documents and their assertions represent the QoS policies that are attached to the BPEL process to be able to enforce the QoS specified as part of the SLA.

The gap between these two layers is bridged by transforming the corresponding models from the choreography layer to executable models in the orchestration layer as indicated using the dashed lines in Figure 5.2. As outlined earlier in this thesis, the importance of QoS in cross-organizational business processes makes it necessary to consider these aspects from the beginning of the development process. Similarly, SLAs are transformed to WS-QoSPolicy statements – our domain-specific extension to WS-Policy – that are directly attached to the corresponding partner links in BPEL allowing an enforcement by a BPEL engine.

In the following paragraphs, we present each of these transformation steps and the SLA and QoS integration in detail.

5.4.2 Mapping WS-CDL to BPEL

The main goal of transforming WS-CDL to BPEL is to allow the participants a rapid modeling and development process and generate relevant BPEL and WSDL artifacts which can then be used as a basis to implement the private (non-visible) business logic. The projection of such a global description to endpoint processes whose interactions precisely realize the global description is called *endpoint projection* [28].

```
<package>
  <choreography>
    <sequence>
      <!-- ... -->
      <sequence>
        <!-- ... -->
        <sequence>
          <interaction operation="sendPO" ...>
            <participate fromRoleTypeRef="Customer" toRoleTypeRef="Manufacturer"
                .../>
            <exchange action="request" ...>
              <!-- ... -->
            </exchange>
          </interaction>
          <interaction operation="sendPO" ...>
            <participate fromRoleTypeRef="Customer" toRoleTypeRef="Manufacturer"
                .../>
            <exchange action="response" ...>
              <!-- ... -->
            </exchange>
          </interaction>
        </sequence>
      </sequence>
    </sequence>
  </choreography>
</package>
```

Listing 5.3: Choreography Example

Mendling and Hafner [89] define the basic mapping rules from WS-CDL to BPEL. They use an recursive XSLT-based approach to generate the BPEL processes by iterating through each role type to check the relevance of the node. The authors consider a node as relevant if it contains activities with the attribute `cdl:toRoleTypeRef` and `cdl:fromRoleTypeRef`. However, this approach does not correspond with the endpoint projection definition given above, because more structured BPEL elements are generated than necessary. This is due to the fact that all parent nodes are considered during the mapping process even if they are not directly relevant (it can be considered as a simple 1:1 mapping). Listing 5.3 depicts an example of this problem by using three nested `sequence` elements.

WS-CDL	**BPEL**	**Semantics**
		Collaborations
relationshipType	partnerLinkType	Definition of the bilateral interaction
	role (ref. by a roleType)	Referenced BPEL roleType generated from a CDL roleType.
	portType (ref. by a roleType)	Referenced BPEL roleType generated from a CDL roleType.
participantType	roleType	
roleType	roleType	Role in an interaction
channelType	correlationSet	Message correlation pattern
		Information Handling
variableDefinitions	variable	Message variables
token	property	Message and process instance correlation
tokenLocator	propertyAlias	Message and process instance correlation
		Activities
workunit (nested in choice)	case	repeat and block attributes always false in this case
workunit (block = true)	(receive)	Concept of blocking condition not defined in BPEL [15]
workunit (all other cases)	while	repeat = true and block = false
sequence	sequence	Sequential execution of activity units
parallel	flow	Parallel execution of activities
choice	switch	If inspected `roleType` is referenced in the guard condition of the inner workunit
	onMessage (nested in) pick	If inspected `roleType` is not referenced in the guard condition of the inner workunit but referenced in an `interaction` activity
interaction		
action = request	invoke	`fromRoleType` attribute corresponds to inspected role type
action = request	receive	`toRoleType` attribute corresponds to inspected role type. If `cdl:interaction` inside `cdl:workunit` which is defined inside a `cdl:choice` generate a BPEL `onMessage`
action = response	reply	toRoleType attribute corresponds to inspected role type
action = response	receive	`receive` only in the asynchronous case. For synchronous interaction append `outputVariable` to corresponding BPEL `invoke` which is defined in case 1
timeout	pick, onAlarm (or) onMessage	
perform	*no mapping*	Separately defined choreography is performed
assign	assign (for party in roleType)	Variable assignment
silentAction	sequence with nested empty	To be refined in the BPEL process
noAction	empty (for party in roleType)	Do nothing
finalize	compensationHandler	Finalizing activities after completion

Table 5.1: WS-CDL to BPEL Mapping

Therefore, we have used and adapted the rules from [89] and propose an extended endpoint projection mechanism based on a so-called *relevance mapping*. The basic idea is to map only those WS-CDL elements which are relevant in the BPEL process. To map the different ordering structures, we need to distinguish between *child* and *descendant* relevance. The former describes that a relevant basic activity occurs as an immediate child of the respective ordering structure in the XML tree whereas the latter describes that a relevant basic activity is nested at an arbitrary level. The relevance of a WS-CDL basic activity is determined by the occurrence of a `cdl:interaction`, `cdl:assign` or `cdl:silentAction` where the `roleType` attribute is matching the `roleType` of the corresponding BPEL process.

If a node represents a relevant activity as described above, it is mapped to a BPEL activity according to Table 5.1, otherwise no mapping is generated. The basic algorithm for the relevance mapping is depicted in Algorithm 2.

Algorithm 2 Relevance Mapping Algorithm

```
1:  procedure TRANSFORM(WS-CDL document $cdl$)
2:      for all $role \in cdl.roleTypes$ do
3:          for all $activity \in cdl.activities$ do
4:              if $activity$ is an ordering structure OR $activity$ is a workunit then
5:                  RELEVANCEMAPPING($activity$, $role$)
6:              end if
7:          end for
8:      end for
9:  end procedure

10: procedure RELEVANCEMAPPING(Node $n$, RoleType $role$)
11:     if $n$ is descendant relevant then
12:         if relevant child count of $n > 1$ then
13:             CREATEBPELMAPPING(n)
14:         end if
15:         for all $child \in n.childNodes$ do
16:             if $child$ is child relevant then
17:                 CREATEBPELMAPPING($child$)
18:             else
19:                 RELEVANCEMAPPING($child$,$role$)
20:             end if
21:         end for
22:     end if
23: end procedure
```

From lines 2–8, we generate a BPEL process for each role type. The algorithm inspects the `cdl:choreography` tag of the WS-CDL document by iterating each activity. If the activity type is an ordering structure or a workunit, a relevance mapping has to be performed. If the currently inspected activity is *descendant relevant*, i.e., it contains relevant descendant basic activities or workunits that need to be mapped, we have to consider all child nodes of this activity (line 11). If an activity is *child relevant* (line 16), i.e., the immediate child contains a relevant basic activity that has to be mapped, we have to generate the corresponding BPEL mapping according to Table 5.1 (line 17). Otherwise, we recursively visit all child nodes (line 19). For our BPEL mapping we implemented an additional optimization concerning the ordering structures. If a `cdl:parallel` or `cdl:sequence` ordering structure contains only one basic child activity, this ordering structure is ignored in the BPEL mapping (lines 12–14). For instance considering the example from Listing 5.3, only one BPEL sequence activity will be generated.

In Table 5.1 we have depicted a detailed overview of the WS-CDL to BPEL mapping rules. These rules are based on the mappings proposed by Mendling and Hafner [89] and adapted where necessary. These adaptations mainly include `cdl:interaction` and `cdl:choice`. For the `cdl:interaction` activity and `cdl:choice` ordering structure, we also have to consider the role types to determine the sending and receiving party. Additionally, we address both the synchronous and asynchronous message exchange patterns properly in the `cdl:interaction` activity.

5.4.3 Generating WSDL Descriptions

The WSDL descriptions define a static structure which can be extracted from the choreography without analyzing the choreography flow in detail. The necessary element mapping from WS-CDL to WSDL is shown in Table 5.2. On the left side the structure of a WSDL file is used to show the corresponding elements of WS-CDL on the right side. The knowledge from this mapping is then used to implement the WSDL generation algorithm as shown in Algorithm 3.

WSDL		WS-CDL	
Element	*Attribute*	*Element*	*Attribute*
definitions	xmlns:tns	package	xmlns:tns
	targetNS		targetNS
	name	behavior	name
message	name	exchange	informationType
portType	name	behavior	interface
operation	name	interaction	operation
[input || output]	name	exchange	action
	message		informationType
binding	name	behavior	name + "Binding"
	type		"tns:"+interface+"Binding"
operation	name	interaction	operation
soap:operation	soapAction	behavior	interface namespace
input		interaction	operation
soap:body	namespace	behavior	interface namespace
output			
soap:body	namespace	behavior	interfaces namespace
service	name	behavior	interface+"Service"
port	name	behavior	interface+"Port"
	binding		"tns:"+name+"Binding"

Table 5.2: WS-CDL to WSDL Mapping

The WSDL generation works as follows: From lines 2–6, we generate a new WSDL document for each `roleType` of the choreography if the service interface is invoked somewhere in the choreography flow (line 3). The main idea is to check if the `roleType` is referenced within a `channelType` and a `variable` for this `channelType` exists that is used in an `interaction` with another partner. If this is the case, the `roleType` is in use and a WSDL needs to be generated. The WSDL document itself is created in the `CreateWSDL()` method (lines 8–28). The methods `CreateNode()` and `AppendNode()` are used to build the WSDL document. For readability we omitted the generation of the XML attributes (which can be seen in Table 5.2).

Algorithm 3 WSDL Generation Algorithm

```
1: procedure GENERATEWSDLFILES(WS-CDL document cdl)
2:     for all $role \in cdl.roleTypes$ do
3:         if $role$ is referenced in $cdl$ then
4:             CREATEWSDL(cdl, $role$)
5:         end if
6:     end for
7: end procedure

8: procedure CREATEWSDL(WS-CDL cdl, RoleType $role$)
9:     CREATEFILE(role.behaviorInterface + ".wsdl"))        ▷create a wsdl file
10:    CREATENODE("wsdl:definitions")        ▷creates a "definitions" element in the WSDL file
11:    for all $i \in cdl.interactions$ do
12:        if $role$ == i.getAttribute("toRoleTypeDef") then
13:            CREATENODE("wsdl:message")
14:        end if
15:    end for
16:    for all $b \in role.behavior$ do
17:        $ptNode \leftarrow$ CREATENODE("wsdl:portType")
18:        $bdNode \leftarrow$ CREATENODE("wsdl:binding")
19:        for all $i \in cdl.interactions$ do        ▷for each interaction check if role is referenced
20:            if $role$ == i.getAttribute("toRoleTypeDef") then
21:                APPENDNODE($ptNode$,"wsdl:operation")
22:                APPENDNODE($bdNode$,"wsdl:operation")
23:            end if
24:        end for
25:        $sNode \leftarrow$ CREATENODE("wsdl:service")        ▷create the "service" element
26:        APPENDNODE($sNode$, "wsdl:port")
27:    end for
28: end procedure
```

5.4.4 SLA/QoS Integration

The integration of QoS attributes in Web service based business process development raises the need for appropriate techniques to consider QoS at the choreography and orchestration layer. At the choreography layer this integration can be achieved by using SLAs which focus (among others) on performance and dependability aspects of the underlying QoS model. In contrast, the integration of QoS at the orchestration layer can be attained by using QoS policies. This section describes how to leverage and extend WS-CDL and BPEL to support SLA-aware choreographies and provide an automated transformation to BPEL that includes enforceable QoS policies that conform to these SLAs.

5.4.4.1 SLA Specification and Integration

As described in Chapter 3, our approach leverages SLAs (by using WSLA) to integrate QoS guarantees and obligations at the choreography layer. For the actual integration, we extended WS-CDL with a construct to reference SLAs. WS-CDL provides a simple extension mechanism by deriving all elements from the `cdl:tExtensibleElements` type. This type adds an optional `description` element of type `cdl:tDescriptionType` (Listing 5.4) to every element derived from the aforementioned type.

```
<simpleType name="tDescriptionType">
   <restriction base="string">
      <enumeration value="documentation" />
      <enumeration value="reference" />
      <enumeration value="semantics" />
   </restriction>
</simpleType>
```

Listing 5.4: WS-CDL Description Type

The `documentation` attribute value specifies any kind of documentation about an element in any non-encoded text form. The `reference` value may contain a URI to a document that further describes the element. The `semantics` value is originally used to contain any machine processable definitions in languages such as RDF [160] or OWL-S [159].

Although the purpose of the `description` element is to provide further details and documentation about an element, we leverage it in our approach because it provides a language-integrated mechanism to extend an element. In that sense, an SLA further describes a specific role type behavior by specifying additional description about the negotiated quality guarantees and obligations. In particular, we leverage the `semantics` attribute value in WS-CDL's optional `description` element as shown in Listing 5.5.

```
1 <roleType name="ManRoleType">
2   <behavior interface="b2o:manInterface" name="ManBehavior"/>
3     <description type="semantics">
4       <qosp:slaReference name="SLA1" uri="ManufacturerCustomerSLA.xml"
            serviceconsumer="CustRoleType"
5       </qosp:slaReference>
6     </description>
7   </behavior>
8 </roleType>
```

Listing 5.5: SLA Integration in WS-CDL

From lines 3–6, the newly added `description` element is shown. The attribute `type` indicates that it is of type semantics. The reference to the SLA is straightforward. We use an `slaReference` element to reference the target WSLA file using an `uri` attribute. Additionally, the role type of the service consumer, i.e., the target party of the SLA, has to be specified.

The other party in the SLA is given by the role type the semantic annotation is applied to, the manufacturer role type `ManRoleType` in this case.

5.4.4.2 SLA to QoS Policy Mapping

In order to define an automated mapping between SLOs and QoS policies, we require at least a set of commonly agreed QoS attributes that are used to define the SLOs. In this approach, we use a subset of service layer QoS attributes introduced in Chapter 3. This subset consists of all QoS attributes that do not depend on network-specific attributes such as latency, because a service provider who is negotiating an SLA typically cannot guarantee any QoS that cannot be enforced internally (e.g., by scaling up to multiple servers).

By imposing this restriction to have a common set of QoS attributes (defined in form of SLA parameters), each SLA can be directly mapped to WS-QoSPolicy assertions. This is achieved by implementing the following steps: Firstly, each SLA is mapped to a WS-QoSPolicy and secondly, each SLA parameter is mapped to a policy assertion. As mentioned earlier, each SLA may consist of one or more SLOs, and each SLO can use a set of SLA parameters and a set of logical expression to define an objective. We have identified three different patterns how SLA parameters are used in the definition of SLOs.

Pattern 1: One SLO is defined for each SLA parameter.

Pattern 2: One SLO consists of multiple SLA parameters.

Pattern 3: SLA parameters are defined in multiple SLOs.

Each of these patterns can be mapped to an equivalent policy which is discussed in detail in the following paragraphs. However, we do not provide the concrete mapping algorithms because they are a straightforward implementation of the concepts below. The main idea can be grasped from the examples provided for the patterns.

Pattern 1. In this pattern, a single SLO in the SLA references exactly one SLA parameter. An example SLA is given in Listing 5.6. We only show the relevant obligations element of the whole SLA to enhance readability.

The mapping of this pattern is fairly straightforward. It uses the `All` operator that contains all policy assertions. For each SLO, exactly one policy assertion will be generated. The resulting WS-QoSPolicy statement is depicted in Listing 5.7.

Pattern 2. In the second pattern, SLA parameters are grouped in an SLO by using the logical operators `And`, `Or`, `Not`, `Implies`. These operator need to be mapped to corresponding WS-Policy operators as shown in Table 5.3.

In Listing 5.8, an example SLO representing this pattern is shown that defines an SLO, called `SLOServicePerformance`, by combining `Throughput` and `ExecutionTime`. The resulting mapping is shown in Listing 5.9. The SLO maps to an `All` operator in WS-Policy that is followed by an `ExactlyOne` element to model the logical OR from the WSLA document.

```
<Objectives>
  <ServiceLevelObjective name="SLOExecutionTime">
    <!-- ... -->
    <Expression>
      <Predicate xsi:type="wsla:Less">
        <SLAParameter>ExecutionTime</SLAParameter>
        <Value>1500</Value>
      </Predicate>
    </Expression>
    <!-- ... -->
  </ServiceLevelObjective>
  <ServiceLevelObjective name="SLOThroughput">
    <!-- ... -->
    <Expression>
      <Predicate xsi:type="wsla:GreaterEqual">
        <SLAParameter>Throughput</SLAParameter>
        <Value>130</Value>
      </Predicate>
    </Expression>
    <!-- ... -->
  </ServiceLevelObjective>
</Objectives>
```

Listing 5.6: WSLA Example for Pattern 1

```
<wsp:Policy>
  <wsp:All>
    <qosp:ExecutionTimeAssertion unit="msec" predicate="Less" value="1500"/>
    <qosp:ThroughputAssertion unit="ops" predicate="GreaterEqual" value="130"/>
  </wsp:All>
</wsp:Policy>
```

Listing 5.7: Mapping Result for Pattern 1

WSLA operator		**WS-QoSPolicy operator**
`And`	→	`All`
`Or`	→	`ExactlyOne`
`Not`	→	Reverse predicate
`Implies`	→	`ExactlyOne` and reverse predicate

Table 5.3: SLA Operator Mapping

Pattern 3. In the third case, a time period has to be specified for each SLO. Therefore, it is possible to define multiple SLOs for different time periods. For instance, during peak hours the execution time of a service has to be less than in non-peak hours. Unfortunately, such a time period based enforcement is currently not supported in WS-Policy and is not considered further in this work. A naive approach to support this case would be the integration of time periods in WS-QoSPolicy. The custom policy handler can then use these time period attributes to enable and disable policy enforcement during the respective periods.

```
<Objectives>
  <ServiceLevelObjective name="SLOExecutionTime">
    <!-- ... -->
    <Expression>
      <Or>
        <Expression>
          <And>
            <Expression>
              <Predicate xsi:type="wsla:Less">
                <SLAParameter>ExecutionTime</SLAParameter>
                <Value>1500</Value>
              </Predicate>
            </Expression>
            <Expression>
              <Predicate xsi:type="wsla:GreaterEqual">
                <SLAParameter>Throughput</SLAParameter>
                <Value>130</Value>
              </Predicate>
            </Expression>
          </And>
        </Expression>
        <Expression>
          <!-- an alternative logical AND combination of the above SLA parameters
              -->
        </Expression>
      </Or>
    </Expression>
    <!-- ... -->
  </ServiceLevelObjective>
</Objectives>
```

Listing 5.8: WSLA Example for Pattern 2

```
<wsp:Policy>
  <wsp:All>
    <wsp:ExactlyOne>
      <wsp:All>
        <qosp:ExecutionTimeAssertion unit="msec" predicate="Less" value="1500"/>
        <qosp:ThroughputAssertion unit="ops" predicate="GreaterEqual" value="130"/>
      </wsp:All>
      <wsp:All>
        <-- other alternative -->
        <qosp:ExecutionTimeAssertion unit="msec" predicate="Less" value="..."/>
        <qosp:ThroughputAssertion unit="ops" predicate="GreaterEqual" value="..."/>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:All>
</wsp:Policy>
```

Listing 5.9: Mapping Result for Pattern 2

5.4.4.3 WS-QoS Policy Integration

Yet, the question remains how to integrate the generated QoS policies in the orchestration layer. Regarding the top-down modeling approach of Web services, two integration approaches can be differentiated: Policies can either be attached to service descriptions (WSDL) or be integrated in BPEL processes.

Attaching policies to WSDL descriptions following the WS-PolicyAttachment [164] specification has two main drawbacks. Firstly, service invocations are always subject to a policy, even if the service consumer has no corresponding SLA. Secondly, the service provider cannot differentiate between multiple policies for the same service since policies do not contain information about participating parties. Therefore, we follow the second approach by integrating policies in BPEL processes.

Extensibility in BPEL is achieved by allowing elements from other namespaces to be specified. The BPEL `partnerLink` element is the right location to integrate the policy. For this integration, both synchronous (request-reply) and asynchronous (callback) message exchange patterns have to be considered. In contrast to the asynchronous case, the service provider has no additional information about the service consumer in the synchronous case, because the `partnerLink` has no service consumer specific details. Therefore, the policy has to be integrated at the service consumer side as illustrated in Listing 5.10.

```
<process>
  <partnerLinks>
    <partnerLink name="POService" partnerLinkType="ns1:POServiceLT" partnerRole="
        POServiceRole">
      <wsp:Policy xmlns:qosp="..." xmlns:wsu="..." wsu:Id="xs:QName" qosp:operation
          ="...">
        <!-- the generated policy goes here -->
      </wsp:Policy>
    </partnerLink>
    <!-- ... -->
  <partnerLinks>
  <!-- ... -->
</process>
```

Listing 5.10: Policy Integration in BPEL

The namespace prefix `xmlns:qosp` refers to the WS-QoSPolicy schema. The attribute `wsu:Id` refers to the `id` attribute of the WS-SecurityUtility schema. In our mapping it is used to refer to the name of the SLA from which the policy was derived. The `qosp:operation` attribute specifies the name of the service operation from the SLA. These attributes are used to identify the origins (from which SLA it was derived) and are used to correlate policy violation information with violation actions as specified in the SLA. Besides integrating QoS policies directly, it is also possible to leverage WS-PolicyAttachment [164] to achieve the integration. However, we do not present it here because it is straightforward to adapt the current approach to WS-PolicyAttachment.

5.5 Architecture and Execution Environment

The concepts and algorithms described in this chapter have been implemented in Java using a simple Swing-based graphical user interface. The architecture of this system consists of four parts which are depicted in Figure 5.3. We distinguish between modeling and execution phase. The modeling phase consists of editing choreographies and SLAs, transforming WS-CDL into BPEL and SLAs into policy assertions, and finally generating the WSDL artifacts. After implementing the private business logic in the BPEL artifacts and the corresponding services, they can be deployed in the VieDAME runtime as part of the execution phase. Please note that we do not describe the whole VieDAME system, however, we give a comprehensive overview of the whole system below and show how it can be used to achieve the BPEL process execution and the QoS enforcement by leveraging a dynamic service adaptation approach. For a detailed overview and evaluation of VieDAME, we refer the reader to [99] and [100].

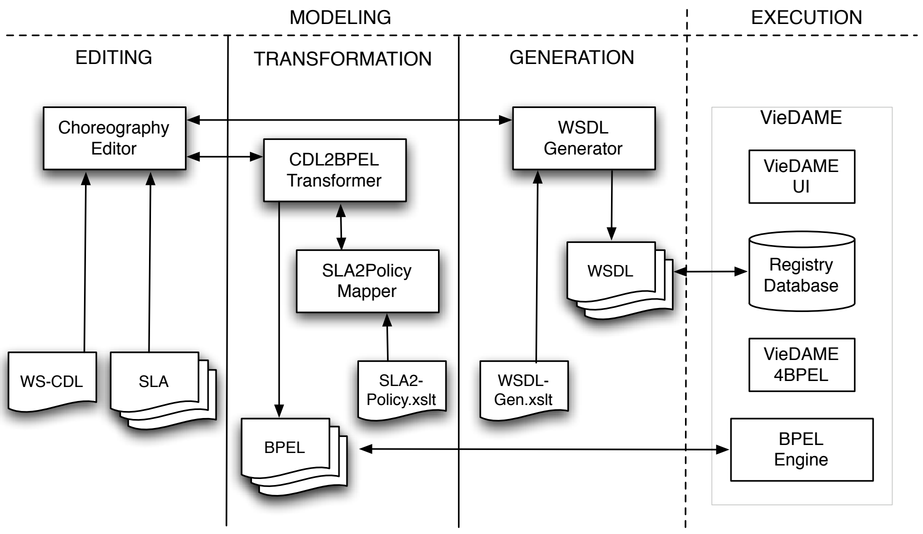


Figure 5.3: System Architecture

5.5.1 Modeling Phase

Editing choreographies in our approach is done in two steps: Firstly, the choreography is modeled using the Pi4soa Eclipse plugin [122]. This plugin provides a graphical tool for creating complex choreographies. Secondly, our simple Swing-based SLA annotation tool is used to add SLA references to specific role types. Furthermore, the editor initiates the generation of WSDL and BPEL artifacts as described above.

The *Transformation* component implements the algorithms for transforming WS-CDL to BPEL, and SLA to policies. The WS-CDL to BPEL transformation is implemented using the DOM4J API whereas the SLA transformation is implemented using XSLT. During the transformation step one BPEL document is generated for each partner including the policy references which conform to the SLAs in the choreography layer.

The *Generation* component is responsible for generating the WSDL files from a choreography according to the algorithm described earlier in this chapter. This component is also implemented using XSLT stylesheets.

5.5.2 Execution Phase

In order to execute the BPEL processes, the artifacts generated during the modeling phase need to be implemented and then deployed for enactment. Therefore, an adaptive BPEL engine with WS-Policy support is required to enable the policy enforcement whenever the specified QoS attribute values are below the guarantees specified by the provider. In this case service adaptation is required to select a service that meets to negotiated QoS guarantees.

5.5.2.1 VieDAME Approach

The VieDAME environment is an ideal approach for executing the generated BPEL processes and enforcing their QoS policies. It provides a non-intrusive monitoring and service adaptation approach based on AOP. The main idea is to achieve *non-intrusive behavior* with regard to dynamic service adaptation, which enables the runtime exchange of partner links within a BPEL process, without any changes to the BPEL process or the involved partner services. This dynamic service adaptation is a key feature to realize and enforce the QoS policies that have been generated and transformed from the SLA as described above.

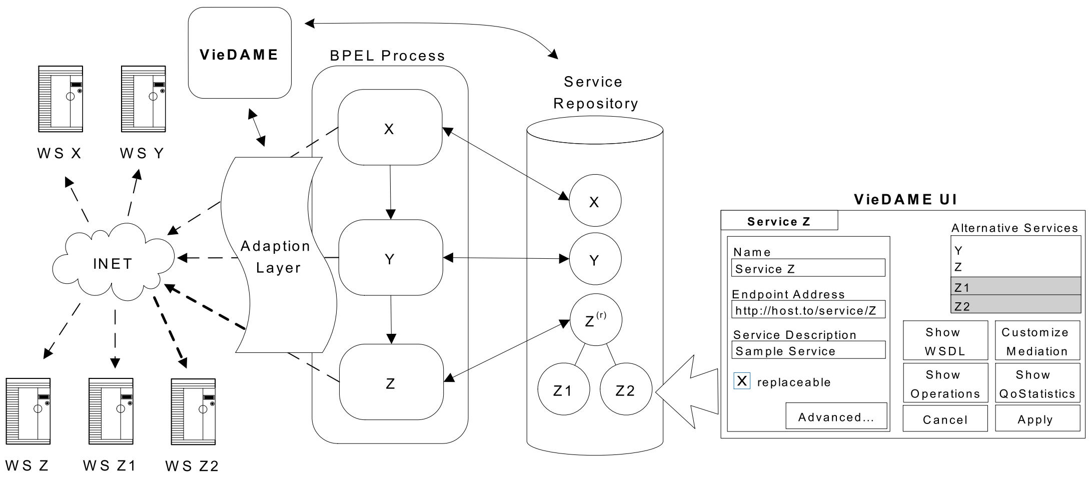


Figure 5.4: VieDAME enhanced BPEL environment

In Figure 5.4, we have depicted the VieDAME approach that we use for the dynamic service adaptation. The core part is the BPEL process which has a certain control flow and invokes a number of partner services. The partner services are generally hosted on different machines distributed over the Web. In VieDAME each service in a BPEL process can be marked as *replaceable* to indicate that alternative services can be configured and invoked instead of the original service that is defined in the process. An alternative service can either be *syntactically* or *semantically* equivalent. The former indicates that the interfaces of the original and the alternative services match. This is, for example, the case, when multiple instances of the same service are hosted on different machines to provide increased reliability. The latter indicates that the services only have the same functionality but expose it using different interfaces, resulting in different representations of the same message payload. This mainly occurs when services need to be exchanged that come from completely different providers on the Web. In our WS-CDL to BPEL transformation approach, this does typically not happen, since the service provider is responsible for providing a number of alternative services that have the same interface, but different QoS (in order to meet it's QoS guarantees).

Each service and all of its alternative services' endpoints are stored in the VieDAME service repository. If a service should be dynamically replaced with an alternative service during process execution, the original partner service captured by the VieDAME's adaptation layer has to be marked replaceable in the VieDAME UI (right side of Figure 5.4). Alternative services that can replace the original service defined by the BPEL process may be added at any time by providing their interface description in the VieDAME UI. Once they are linked to the original service, a replacement policy can be selected to control which of the available alternatives will be used. Additionally, the VieDAME UI can be used to define mediation rules that allow alternative services to be used where the interfaces do not match the original interface of the partner service.

5.5.2.2 Architecture

The VieDAME system is split into the VieDAME core and the VieDAME engine adapters. The VieDAME core ties together the monitoring, service selection and message transformation facilities as well as provides services such as data store access, scheduling and configuration data, whereas the engine adapters represent the connector to the BPEL engine. Thus, to support new BPEL engines, it is (only) necessary to implement an engine adapter specifically to the desired engine implementation. The VieDAME environment currently supports ActiveBPEL 3.0 [1].

Figure 5.5 depicts the architectural approach taken as well as the system dependencies. Firstly, the flow of events in a standard BPEL environment is described, without any interaction with the VieDAME system. Secondly, the additional steps performed in a full-featured VieDAME environment are explained. It includes service monitoring, alternative service selection and message transformation.

System Overview. After deployment of a process definition (1), the BPEL processor (2) is ready to create new process instances. A new BPEL process instance (2a) is created when one

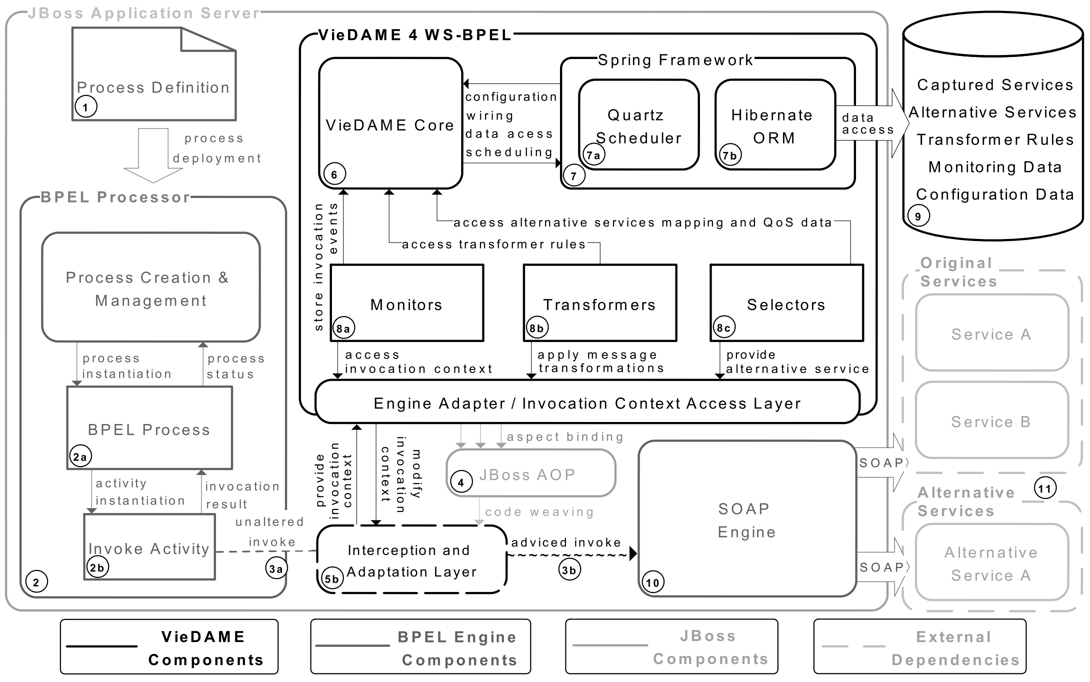


Figure 5.5: VieDAME Overall System Architecture

of its start activities is triggered, e.g., by an incoming message. Interaction with a partner link is initiated by invoke activities (2b) that create SOAP calls (3a). These SOAP calls are executed by a SOAP engine (10) that returns the result of the invocation of an arbitrary partner service (11) upon completion of the request. The invoke activity reports the result to the process instance which in turn proceeds to the next activity.

When the VieDAME system is enabled, an additional level of processing is introduced, manifested in the *Interception and Adaptation Layer* (5b), hereinafter referred to as the *IAL*. Basically, the IAL is created by *aspects* that are bound to specific *join points* in the BPEL engine's code by the definition of *pointcuts*. The *advice* code is then *woven* into the original method invocations by the AOP framework (4) at load time. The IAL provides a bidirectional interface for the *engine adapter* (5a) to tap the communication between the invoke activity (2b) and the SOAP engine (10). The engine adapter in turn provides read-write access to the *invocation context*, enabling other VieDAME components – such as `Monitor` (8a) or `Selector` (8c) – to access and modify invocation parameters and other runtime data.

The first VieDAME component that is called after interception of a partner link invocation by the IAL is the `Monitor` component. It examines the invocation context to find the service name, endpoint address and operation name in order to load a previously persisted service reference or to persist a new service reference for future requests. The `Monitor` leverages the VieDAME core (6) and the ORM framework (7b) respectively to persist objects to a data store (10). Furthermore, the `Monitor` activates a timer to measure the time elapsed during the actual SOAP call and stores this information together with a reference to the involved service and success/failed flag. A scheduling framework (7a) is used to bulk-insert invocation events in order to optimize data store access. Based on the data stored by the `Monitor` component, real-time QoS statistics can be calculated that correspond to a subset attributes described in Chapter 3.

If the service reference loaded by (8a) is marked as replaceable, the next VieDAME component takes control. The `Selector` component (8c) determines an alternative partner service by applying some selection algorithm to a list of configured alternative services (9). If an alternative service is found, the invocation context is updated with the alternative's endpoint parameters. Like the `Monitor` component, `Selector`s access the data store by using (6) and (7b). The same applies to the last VieDAME component that can be called, the `Transformer` component. A `Transformer` (8b) is responsible for compensating the interface mismatch between the original service and the alternative. The `Transformer` uses transformation rules (e.g., XSLT stylesheets) stored in (9) to perform the required transformations.

After all required modifications are applied to the invocation context, the SOAP call is finally proceeded, probably invoking an alternative partner service instead of the original service. The difference between the unaltered invoke (3a) and the advised invoke (3b) is called the *Invocation Context Delta*, or *ICD*. A big ICD indicates many differences between the original service interface and the alternative service interface, whereas a small ICD indicates a replica of the original service (i.e., the original partner service and the alternative partner service only differ in their endpoint address). A zero ICD indicates that neither a service replacement nor message transformation was applied. The ICD measured value can be used as an indicator for determining the degree of adaptation the VieDAME system has performed and whether the environment running VieDAME uses the adaptation facilities at all.

Selectors. Selectors are the key components in VieDAME to realize adaptive behavior that can be leveraged to enforce QoS policies as used in our approach. A selector in the VieDAME system context is the implementation of a particular selection algorithm that determines which of the available alternative services match one or more selection criteria best. VieDAME provides a variety of selection algorithms, ranging from simple randomized and round-robin selectors that can be used for load balancing, to more sophisticated selectors that combine several QoS attributes such as performance and dependability for selection criterion. Moreover, the VieDAME system offers fault-compensating selectors that retry failed service invocations, either with the original service or with an appropriate replacement. To meet further requirements, the VieDAME system can easily be extended with additional `Selector` implementations. This extension is used to implement a custom adaptive selector that leverages the information specified in a QoS policy to select a service that meets these policy requirements. An enforcement of the QoS policies would also require the capabilities of dynamically deploying new service instances if none is currently available that meets the QoS demands, however, this is currently not supported in this approach.

5.6 Discussion

During the implementation of our case study we encountered several aspects which have to be considered when using such a top-down modeling approach. Some of these issues seem inherent to the domain of model-driven development in general. On the one hand, our approach is based on choreographies representing a global viewpoint of the business processes which raises the need for precise modeling of the global behavior. To be more concrete, the business partners have to precisely agree on the message format used for their interaction. On the other hand, after the choreography was initially defined, the underlying business model may evolve and lead to significant changes. Such changes clearly affect the partner processes which causes the generation of new BPEL processes and corresponding WSDL files. However, this regeneration is not applicable in most production systems, therefore, leading to deviations from the choreography.

There have also been some considerable debates as to the relationship between choreography and orchestration. Some people argue that there is no need for choreography and all business interactions can, and in fact, should be modeled in BPEL. Others advocate the use of modeling by using WS-CDL but then lament the lack of execution abilities. In fact, both modeling approaches are feasible and have their strengths and weaknesses. However, BPEL is intended for modeling business processes without knowledge of global viewpoint. In contrast to this, we decided to stay close to the vision of cross-organizational choreography descriptions by using WS-CDL.

The prime motivation for the contribution in this chapter is today's lack of modeling support for SLA-aware business processes. In particular, the need for SLA-aware processes is apparent in inter-organizational business processes. The novelty of our approach lies within the fact that we consider SLAs as first class entities while modeling service choreographies. Our approach enables an automatic generation of executable BPEL orchestrations and WSDL files for each partner in the choreography. A novel contribution is the mapping of QoS information specified in SLAs to QoS policies (by using WS-QoSPolicy) which are attached to the BPEL process. As a consequence, a policy-aware middleware can verify and enforce SLAs, e.g., VieDAME.

Part II

QoS-Aware Service Composition and Execution

Chapter 6

VRESCo – A Runtime for Adaptive Service-Oriented Systems

This chapter describes the Vienna Runtime Environment for Service-Oriented Computing, called VRESCo, which was first introduced in [93]. It includes a detailed description of the service metadata model and the service model, as well as a brief description of the core runtime services. Then we focus on selected aspects of VRESCo, namely querying, dynamic binding and invocation including mediation as a basis for the composition approach in Chapter 7 and 8.

Contents

6.1 Motivation and Overview

The VRESCo (Vienna Runtime Environment for Service-Oriented Computing) project aims at addressing some of the current challenges in Service-Oriented Computing research [117] and practice. This includes topics related to service discovery and metadata, dynamic binding and invocation, service versioning and QoS-aware service composition. Besides this, another goal is to facilitate engineering of service-oriented applications by reconciling some of these topics and abstracting from protocol-related issues. Its main focus is the seamless support of enterprise-level SOA development by providing a feature-driven programming model for adaptive service-oriented applications. This programming model reconciles service metadata and concrete services including their mapping definition and mediation combined with dynamic invocation and QoS support to enable adaptive applications. Adaptiveness is achieved by using dynamic binding and rebinding of services at runtime based on different criteria such as degrading QoS or intermittent or even permanent service failures. As a consequence, this leads to a better availability and reliability in terms of application provisioning and provides more flexibility regarding changes in the architecture because concrete service details are abstracted by features as part of the metadata model.

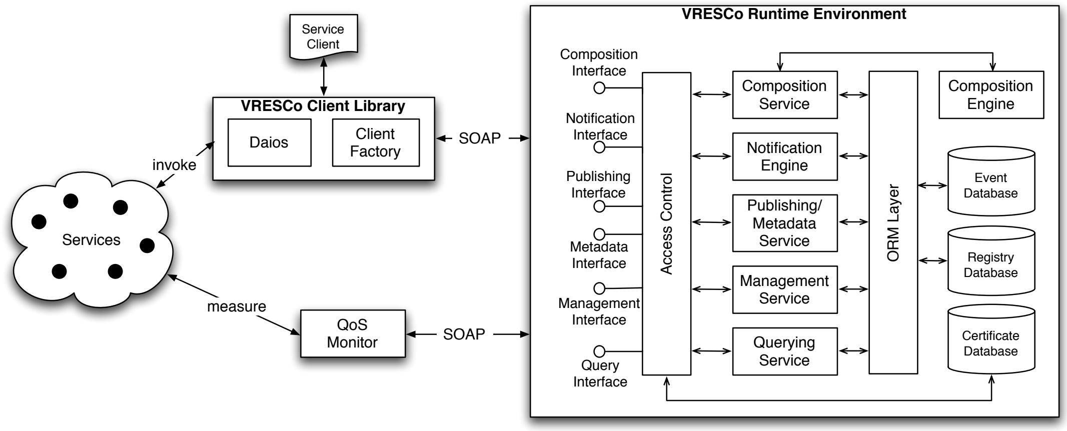


Figure 6.1: VRESCo Overview

The architectural overview of VRESCo is shown in Figure 6.1. The VRESCo core services are provided as Web services that can be accessed either directly using SOAP or by using the client library that provides a simple API. Furthermore, the DAIOS framework [80] has been integrated into the client library, and provides dynamic and asynchronous invocations of Web services. The access control layer guarantees that only authorized clients can access the core services. Services and associated metadata are stored in the service registry which is accessed internally using an Object-Relational Mapping (ORM) layer. Finally, the QoS monitor from Chapter 4 is integrated for regularly measuring the QoS values of services. The overall runtime environment is implemented in C# using the Windows Communication Foundation [82]. In order to enable a flexible development, the client library is currently provided for C# and Java. It provides a standard client implementation for the core services that can be used directly to work with VRESCo in an object-oriented way.

There are several VRESCo core services. The Publishing/Metadata Service is used to publish services and metadata into the registry database. Furthermore, the Management Service is responsible for managing user information (e.g., name, password, etc.) whereas the Querying Service is used to query all information stored in the database. The task of the Notification Engine is to inform users when certain events of interest occur inside the runtime, while the Composition Service finally provides mechanisms to facilitate QoS-aware service composition.

Before elaborating on the core services in detail, we provide a comprehensive insight into the VRESCo metadata model and its mapping to the concrete service model. This model is the foundation for VRESCo as well as for the composition approach. Following this, the core services are described in more detail, with a particular focus on querying, dynamic binding and invocation including mediation.

6.2 A Metadata Model for Services

In this section, we address the problem that current SOA runtimes lack an integrated mechanism allowing to express metadata about services as part of their core runtime functionality. This is necessary to achieve a high degree of decoupling of service consumers and providers, with the ultimate goal of binding to a given "feature" (functionality), that is implemented by concrete services, rather than to concrete service instances themselves. The service runtime environment has to provide the necessary abstractions and mechanisms for realizing it. Therefore, we propose a feature-driven metadata model for services, which enables application developers to describe the functionality that services offer, the input and output of a service operation and the pre- and postconditions of a service (typically defined in the domain model). A metadata description specifies an abstract service in terms of features that have to be mapped to concrete service instances (including possible transformations if the interfaces do not match). Additionally, the model provides a way to categorize services according to common business functionality. Please note that our metadata model is not intended to compete with approaches used in the Semantic Web services community (SWS) [86], such as OWL-S for describing semantics of services using ontologies. We aim at enterprise development where metadata is an important business asset which should not be accessible for everyone, as opposed to the SWS community where domain ontologies should be public to facilitate integration among different providers and consumers. It is therefore important to provide a deep integration of the metadata model with the core services provided by VRESCo.

6.2.1 Illustrative Example

In this case study we tackle the problem of building a composite service for *cell phone number portability*. Such a service is currently available to customers when they change the cell phone operator (CPO) and want to keep their old number, thus the telephone number has to be ported to the new operator. We assume a simplified process such as the one depicted in Figure 6.2. The process itself runs internally within the CPO where the customer recently signed a contract. After signing the contract, the new CPO has to port the customer's old number.

Therefore, the CPO has to coordinate with the customer's old provider in order to support this feature.

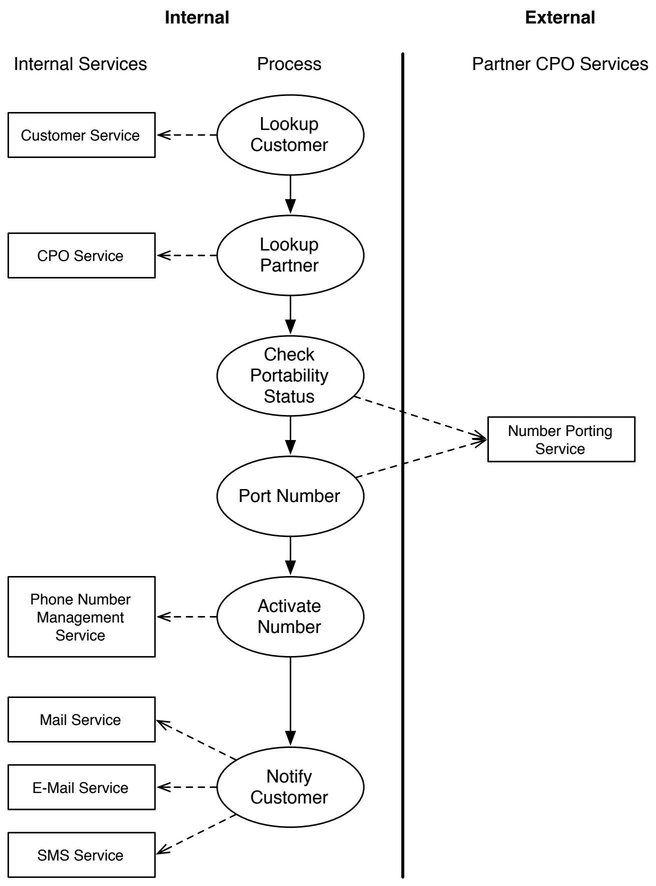


Figure 6.2: Number Portability Process

The process starts by looking up the customer using the internal `Customer Service`. After finding the customer, the process has to check which CPO has served this customer in the past. This is done using the internal `CPO Service`. When the old provider is known the process has to use this provider's `Number Porting Service` to check if the porting operation is currently possible, and, if it is, initiate the porting process on the partner's side. If porting is currently not possible the process has to escalate (which is not shown in the example for reasons of brevity). After successfully communicating the port to the partner, the phone number is locally activated using the internal `Phone Number Management Service`, and, finally, the customer is notified. This is done using different messaging mechanisms, according to the preferences of the customer.

In this process, a number of dynamic service bindings exist: the external `Number Porting Service` that has to be used is an outcome of the result of the `Lookup CPO` activity and cannot be determined statically; the same is true for the notification service used to implement the last activity in the process. The possible alternatives for each of these activities are well-known, and their number is relatively small: in this example it is not reasonable to assume that the CPO wants to cooperate with unknown partners, or that a previously unknown notification service (e.g., a public service from the Internet) should be used. However, the possible alternatives are not static, new CPOs may enter the market while others leave, and new notification services may be implemented while others are deactivated. The process itself should not have to be adapted manually as a result of such changes in the environment. Additionally, we cannot assume that each of the services has the same interface, or relies on the same implementation-level data types, i.e., the services selected and bound at runtime may vary significantly in terms of both interfaces and implementation. This complicates the problem of dynamic binding, since mediation between per se incompatible invocations and interfaces may become necessary. Therefore, standard programmatic approaches to handle variability such as the well-known Strategy pattern [54] are not suitable even in this relatively simple illustrative example.

6.2.2 Metadata Model

In Figure 6.3, we have depicted our basic metadata model for modeling services, their features, pre- and postconditions using a slightly relaxed UML notation. In this model, we have to abstract from the technical service implementation to achieve a common understanding what a service does and what it expects and provides. In a typical SOA environment, there may be multiple services that facilitate the same business goal, therefore, we also need a way to group services according to their functionality. In the following, we use an *italic* font to represent model elements and `typewriter` to indicate instances of a model element.

The main building blocks of the VRESCo metadata model are *Concept*s. A *Concept* is the definition of an entity in the domain model. We distinguish between three different types of *Concept*s:

- *Feature*s represent activities in the domain that perform a concrete action. It can be understood as a functionality that a service has to implement. Possible features from the example process in Figure 6.2 are `Check_Portability_Status`, `Port_Number` or `Notify_Customer`.
- *Data Concept*s represent concrete entities in the domain (e.g., customers, addresses or bills) which are specified using pre-defined atomic concepts such as strings or numbers and/or other *Data Concept*s. For example, a concept `Customer` might consist of `Customer_Id`, `Customer_Name`, and `Address`.
- *Predicate*s represent domain-specific statements that either return `true` or `false`. Each *Predicate* can have a number of *Argument*s that express their input. An example of a state predicate for a *Feature* `Port_Number` could be `Port_Status_Ok(Phone_Number)`, expressing the portability status of a given phone number.

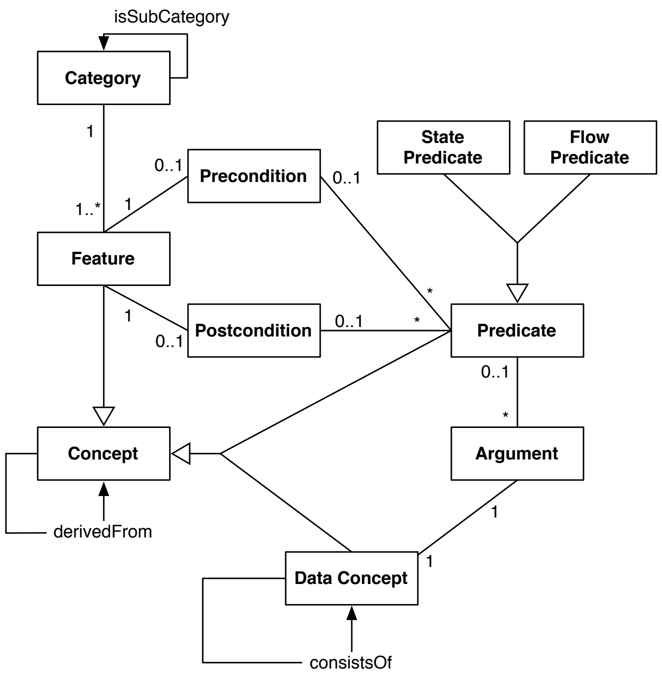


Figure 6.3: VRESCo Metadata Model

*Concept*s have a well-defined meaning specific to a certain domain. For example, the *Data Concept* `Customer` in one domain is clearly different from the concept `Customer` in another. Concepts may be derived from other concepts; that is specifically interesting for *Data Concept*s, e.g., it is possible to define the concept `Premium_Customer` which is a special variant of the more general concept `Customer`.

Each *Feature* in the metadata model is associated with one *Category* expressing the purpose of a service (e.g., `Phone_Number_Porting`). Each category can have additional subcategories to allow a more fine-grained differentiation. The semantics of subcategories is multiple inheritance, meaning that each subcategory inherits all *Features* from all of its parents. Each *Feature* can have a *Precondition* and a *Postcondition* expressing logical statements that have to hold before and after the execution of a *Feature*. Both types of conditions are composed of multiple *Predicate*s, each having a number of (optional) *Argument*s that refer to a *Concept* in the domain model (indirectly through a *Data Concept*). There are two different types of *Predicate*s:

- *Flow Predicate*: This type of predicate can be used in pre- and postconditions to indicate constraints related to the data flow, such as data required or produced by a feature. This is expressed by using two special variants of flow predicates called `requires` and `produces`. An example from our number porting process would be a *Postcondition* having a predicate `requires(Customer)`, expressing that a concept `Customer` is needed as an input for feature `Check_Portability_Status`. In case of a *Postcondition*, the predicate `produces(Portability_Status)` can be used to express that the aforementioned feature produces the data concept `Portability_Status` as output.

- *State Predicate*: This type of predicate expresses some global behavior that is valid either before (for a precondition) or after invoking a feature (for a postcondition). For example, the `Notify_Customer` feature can be subject to the following conditions. The precondition `exists(Customer)` expresses that a customer has to exist in the system before it can be notified. A postcondition `notified(Customer)` expresses that a customer has been notified after invoking the `Notify_Customer` feature. Both conditions express global state that hold either before or after invoking a feature.

These two types of predicates can be specified by the developer to explicitly define flow and state behavior, however, they are not required or enforced by the implementation upon execution time. This kind of metadata only provides knowledge which is required, when performing (semi-)automated service composition, where such pre- and postconditions are a required means to guide the composition process for stateful services.

6.2.3 Service Model and Metadata Model Mapping

In the following, we substantiate the VRESCo metadata model as described in the previous section by explaining the mapping of concrete Web services to features and their concepts in the metadata model.

6.2.3.1 Service Model

The VRESCo service model constitutes the basic information of concrete services that is managed by VRESCo and can be invoked by using the Daios framework. The service model depicted on the lower half of Figure 6.4 basically follows the Web service based notation as introduced by WSDL with extensions to enable service versioning [79], represent QoS and enable eventing on a service runtime level [90].

A concrete service (*Service*) defines the basic information of a service such as the name, description, owner and consists of at least one service revision. A service revision (*Revision*) contains all technical information that is necessary to invoke it (such as a reference to the WSDL file) and represents a collection of operations (*Operation*). Every operation may have a number of input parameters, and may return one or more output parameters (*Parameter*). A revision itself can have parent and child revisions to represent a complete versioning graph of a concrete service (for details see [79]). Both, revision and operation can have a number of QoS attributes (*QoS*) representing all service-level attributes from Chapter 3. The distinction in revision- and operation-specific QoS is necessary, because attributes such as response time depend the execution time of an operation, whereas availability is typically given for the revision itself (if a service is not available, all operations are not available too). In addition, a service, a revision and an operation can have a number of events associated with it (not shown in Figure 6.4). These events are raised by the runtime whenever an action is performed, e.g., invoking a service, publishing a new service or creating a new revision [90].

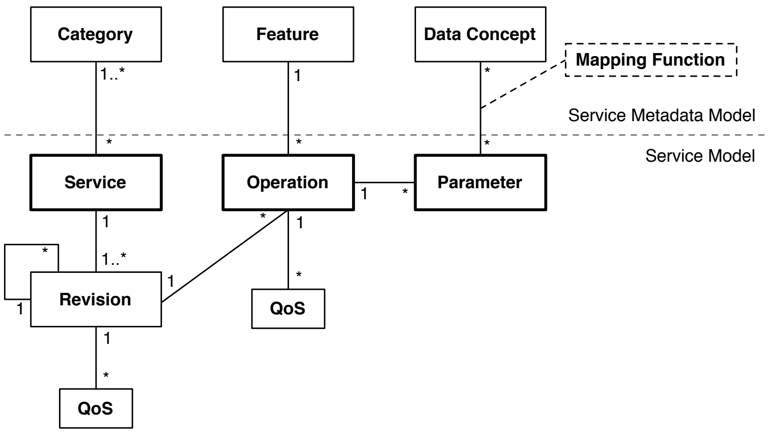


Figure 6.4: Service Model and Metadata Mapping

6.2.3.2 Mapping Metadata to Concrete Services

In order to associate metadata to concrete services in the service model we have to establish a mapping between the metadata and the services. The mapping is shown in Figure 6.4, where the dashed line represents the connections between the elements in the metadata model and the elements in the service model. Services are grouped into categories (*Category*), where every service may belong to several categories at the same time. Services within the same category provide at least one feature of this category.

Service operations are mapped to features (*Feature*). Currently, we assume a 1:1 mapping between features and operations; every feature is implemented in exactly one service operation, and every operation implements exactly one feature of a category. However, theoretically it is possible to implemented a 1:n mapping because a feature may encompass two or more service operations (but currently not supported in VRESCo). The input and output parameters of the service operations map to data concepts (*Data Concept*). Every parameter is represented by one or more concepts in the domain model. This means that all data that a service accepts as input or passes as output is well-defined using data concepts and optionally annotated with the flow predicates `requires` (for input) and `produces` (for output).

The concrete mapping of service parameters to concepts is described using *Mapping Function*s. In general, rules for both the mapping from the parameter to the concept and vice versa have to be specified. If an operation requires a certain state prior to its execution then this requirement can be modeled as a state predicate (*State Predicate*) in the domain model. The same is true for state changes as a result of the execution of an operation.

6.2.3.3 Mapping Example

Figure 6.5 illustrates an example based on the `NotifyCustomer` activity introduced in Section 6.2. In this example, a category `PortingServices` with a feature `Notify_Customer` is defined. Two different SMS providers offer a *SMSService* that belongs to the category `PortingServices` and their operations "implement" the feature `Notify_Customer`.

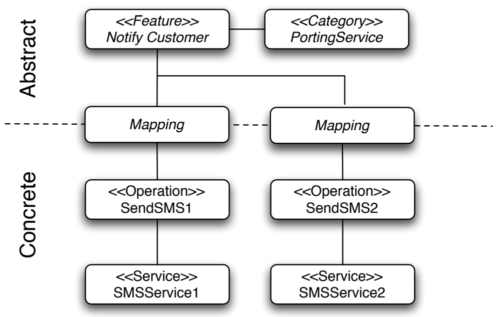


Figure 6.5: VRESCo Metadata Model – Mapping Example

In order to map these service operations to the `Notify_Customer` feature, VRESCo needs to provide the necessary means to define these mapping functions to enable a runtime mediation between the abstract metadata and concrete service implementations. The VRESCo Mapping Framework (VMF) introduced in Section 6.3.3 implements the necessary mechanisms and depicts the source code of this mapping example in Listing 6.2.

6.3 Core Runtime Services

In this section we give an overview of all the VRESCo runtime services that are used by the client library implementation to ease its use across the two target platforms Java and Microsoft .NET/C#.

6.3.1 Overview

The architecture of VRESCo follows a service-oriented design by providing the core functionality as dedicated Web services. This enables a platform independent use by leveraging SOAP as a transport protocol, however, for convenience and ease-of-use, a client library is provided that implements a standard client for VRESCo's core services. Whenever, a core service is invoked, the access control layer checks the authentication and authorization of the caller by using a claim-based approach on different authorizable resources (e.g., services, categories, etc). After granting access, the core service logic is executed. Internally, the core architecture is based on a dedicated data access layer that encapsulates the core entity classes (such as `Service` or `ServiceRevision`) from the database and achieves database independence. VRESCo also leverages a powerful eventing support that defines a number of complex events that are emitted and correlated internally, whenever a certain action occurs such as the publishing of a new service or the invocation of the service through DAIOS. All events are persisted into the registry database and can thus be queried and analyzed. Additionally, these events form the basis for service provenance, an approach to analyze which user performed which actions in the system. For a detailed description of the eventing infrastructure and the service provenance including the claim-based security approach, we refer to [90] and [92] respectively.

In the following, we provide an overview of the core services regarding their main purpose.

Publishing Service: It provides support for the basic service lifecycle operations. In particular, it allows to manage services and service revisions including support for service versioning [79] (i.e., branching and merging revisions, tagging revisions). The management of services includes functionality to create, update and delete services and revisions, adding operations and parameters as well as the activation and de-activation of revisions.

Metadata Service: It provides support for all aspects related to metadata management. In particular, the definition of categories, features, data concepts, pre- and postconditions and specific QoS properties that allow to define customized QoS models. Additionally, the metadata service implements the VMF (VRESCo Mapping Framework) allowing the specification of mediation rules when defining the mapping to facilitate runtime mediation as described in detail below.

Querying Service: It provides access to the registry database by using a specialized query language VQL (VRESCo Query Language). VQL provides a type-safe mechanism to query all information from the service- and metadata model, for example by querying a feature with name `Send_SMS` and `QoS.ResponseTime` < 1000 msec. To this end, VQL implements different querying strategies allowing to express how a query should be executed. For example, the *exact* querying strategy tries to match all criteria whereas a *relaxed* querying strategy tries to match most of the specified criteria. For *priority* querying each criterion can be associated with a weight to express the importance.

Management Service: It provides an interface for a number of management operations, in particular user management and QoS management. The former focuses on typical CRUD (Create, Read, Update, Delete) operations for users and groups as well as the management of user claims to implement the claim-based access control. The latter provides an interface for external QoS monitors to deliver their monitoring data to VRESCo.

Notification Service: It provides an interface for clients to subscribe to certain events that occur inside the runtime. A number of events are triggered at runtime that can be of interest to a client to implement various kinds of applications scenarios. These event notifications are implemented using the Esper event engine [50] and WS-Eventing [162]. For an in-depth description of the eventing capabilities we refer to [90].

Composition Service: It provides a QoS-aware Composition as a Service (CaaS) approach by using a domain-specific language called VCL to facilitate the specification of QoS constraints that are resolved by the runtime. This approach builds upon the feature-driven notation of the VRESCo metadata model and leverages VQL and the meditation approach to realize adaptive compositions. The approach is described in detail in Chapter 7 and 8.

In the following, we focus on the query language VQL, the mapping and mediation approach VMF and dynamic invocation using the DAIOS framework as they are relevant for the composition approach that is described in the next chapters. For a detailed overview on the design and implementation of VQL, VMF and DAIOS, we refer to [75], [60] and [77] respectively.

6.3.2 VRESCo Query Language

The VRESCo Query Language (VQL) provides a means to query all information stored in the registry database, i.e., information about services and service metadata in Section 6.2. The main requirements for the query language can be summarized as follows.

6.3.2.1 Requirements

View-based Querying. The internal VRESCo architecture implements the data access via a data access layer (DAL) using dedicated data access objects (DAO). However, these DAOs contain database specific attributes such as IDs (that map to the primary key of a relational database record) or versioning information for optimistic locking. Therefore, these DAOs are only used internally and referred to as *core objects*. For transmission over the network to the client application, these entities are transformed into so-called *user objects* that contain basically the same information but without any database specific fields. However, querying capabilities are used on both sides (internally and externally), therefore, VQL must handle both representations properly, without any changes how the queries can be specified.

Type-Safety and Security. Each VQL query should be type-safe in the sense that the result of a query should be parameterizable with specific data types from the service or metadata model (e.g., a `ServiceRevision` or a `Feature`). Additionally, all query attributes should be subject to runtime existence checks to rule out parameters that do not match a property in the corresponding core or user object. Additionally, queries should be fail-safe against well-know security issues such as SQL injection.

Object-oriented Interface and Expression Library. In order to generate VQL queries at runtime, an object-oriented API for specifying these queries should be available similar to query languages provided by major ORM frameworks such as the Hibernate Criteria API [58]. VQL does not provide a declarative language for specifying a query (such as SQL), which makes it simpler in terms of the implementation because no query parser is necessary. Therefore, a rich library of expressions that can be used to formulate the queries in an object-oriented manner is required.

Mandatory and Optional Criteria. When querying specific information about a service or certain aspects of the metadata model, it is often desired to differentiate between mandatory and optional expressions in a query. For example, one may issue a query to find all services implementing the feature `Send_SMS` which is active and optionally having the QoS attribute response time set to less than 1500 msec. In order to achieve these requirements, different querying strategies have to be provided.

6.3.2.2 Architecture

These requirements are addressed by implementing a query language and a querying service as part of VRESCo. On the client-side the user specifies a query using an object-oriented interface which is provided by the client library. It includes all the necessary code and a rich set of expressions to formulate VQL queries. The basic architecture of VQL is depicted in Figure 6.6.

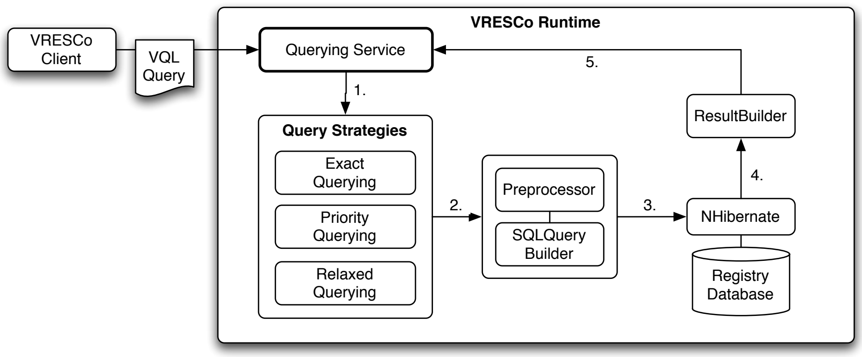


Figure 6.6: VRESCo Query Processing Architecture

After specifying the query (using a `VQuery<T>` object; `T` represents the type-safe query parameter for the resulting query object), it is sent to the `QueryingService` for execution. Depending on the client's *querying strategy*, VQL selects the corresponding querying strategy based on the strategy design pattern [54] and generates the query accordingly (step 1). VQL leverages SQL as its query execution language, therefore, a `VQuery` instance that was received from the client – representing an in-memory object graph of a query – is preprocessed using the `Preprocessor` component (step 2). This preprocessing inspects the query expressions, the criteria and whether the client queries the core or the user object. The property mapping between user and core objects is specified using source code annotations on the core and user object classes. The transformation of a core object to a user object is done in the constructor of a user object upon sending it to a client application.

Additionally, the preprocessing checks if all expressions correspond to attributes in our service- and metadata model. The result of the preprocessing is a generated SQL query that corresponds to the initial `VQuery` (`SQLQueryBuilder` component). When the query is fully generated, a NHibernate session is created to execute the query (step 3). After a successful execution, the `ResultBuilder` component takes the result from the NHibernate session (step 4) and transforms it back into the resulting object `T` that was specified as a template parameter in the `VQuery` object (step 5).

Query Specification. In general, VQL queries consist of a set of criteria where each criterion has a number of expressions. Both criteria and expressions are specified using the querying API provided by the VQL library. Therefore, in contrast to other query languages such as SQL, VQL does not provide a declarative querying language which makes it easier to use. Query criteria can either be `Add` and `Match`. These criteria have different mapping semantics depending on the querying strategy (discussed below). However, the main motivation is to allow the specification of mandatory and optional query criteria when combined with the *priority* or *relaxed* querying strategy.

Besides that, VQL provides a set of expressions that can be used to express common query constraints such as comparison (e.g., smaller, greater, equal, etc.) and logical operators (e.g., AND, OR, NOT, etc.). These expressions are summarized in Table 6.1.

Expression	Description
`And`	Conjunction of two expressions
`Or`	Disjunction of two expressions
`Not`	Negation of an expression
`Eq`	Equal operator
`Le`	Less or equal operator
`Lt`	Less than operator
`Gt`	Greater than operator
`Ge`	Greater or equal operator
`Like`	Similarity operator
`IsNull`	Property is null
`IsNotNull`	Property is not null
`In`	Property is in a given collection
`Between`	Property is between two given values

Table 6.1: VQL Expressions

Listing 6.1 shows an example query to find services that implement the `NotifyCustomer` feature. In general, VQL queries are parameterized using an expected return type. In this case the type `ServiceRevision` (line 2) expresses that the result of the query is a list of revisions.

In the example, two `Add` criteria (lines 5–6) are used to state that a service has to be active and that each service has to implement the `NotifyCustomer` feature. Additionally, three

`Match` criteria are added (lines 7–11). The first criterion expresses that a resulting service should be in a category starting with "Porting". The second and third criterion define the optional QoS attributes (response time and availability). All three `Match` criteria use the priority value as third parameter to define the importance of a criterion.

```
1 // create a query object
2 var query = new VQuery(typeof(ServiceRevision));
3
4 // add query expressions
5 query.Add(Expression.Eq("IsActive", true));
6 query.Add(Expression.Eq("Operations.Feature.Name", "NotifyCustomer"));
7 query.Match(Expression.Like("Service.Category.Name",  "Porting", LikeMatchMode.
     Start), 5);
8 query.Match(Expression.Eq("QoS.Property.Name", "ResponseTime") &
9             Expression.Lt("QoS.DoubleValue", 1500), 3);
10 query.Match(Expression.Eq("QoS.Property.Name", "Availability") &
11             Expression.Gt("QoS.DoubleValue", 0.95), 1);
12
13 // execute the query
14 IVRESCoQuerier querier = VRESCoClientFactory.CreateQuerier("admin", "secret");
15 var results = querier.FindByQuery(query, QueryMode.Priority)
16   as IList<ServiceRevision>;
```

Listing 6.1: VQL Sample Query

The query execution is finally triggered by instantiating an `IVRESCoQuerier` and invoking the `FindByQuery` method using the specific querying strategy, e.g., `QueryMode.Priority` in our example (lines 14–16).

6.3.2.3 Querying Strategies

The querying strategy influences how a query is executed, thus, it defines the behavior of the SQL generation as done in the `SQLQueryBuilder`. In a nutshell, `Add` criteria are transformed to simple predicates within the SQL `WHERE` clause whereas `Match` are handled as SQL sub-selects. The query mapping semantics is summarized in Table 6.2.

Strategy	**`Add` Criteria**	**`Match` Criteria**
Exact	`WHERE` predicate	`IN` (sub-select)
Priority	`WHERE` predicate	`JOIN` (sub-select)
Relaxed	`WHERE` predicate	`JOIN` (sub-select)

Table 6.2: VQL Mapping Semantics

Exact Querying. The *exact querying* strategy forces all criteria to be fulfilled, irrespective whether this is `Add` or `Match`. As a consequence, it is not obvious why two different criteria are used to specify a query when using the exact querying strategy. However, there are scenarios where `Match` has to be used to get the desired results by influencing the SQL generation behavior to enforce sub-selects instead of `WHERE` predicates. In particular, when mapping N:1 and N:M associations (i.e., collection mappings in Hibernate terminology), a query cannot have the same collection more than once in the `WHERE` predicate. The use of sub-selects eliminates this effect in VQL, otherwise such a query would result in `null` because the association tables would have to be joined more than once. As an example reconsider the query in Listing 6.1 and assume that we use an exact querying strategy as opposed to priority. In this case, the last two `Match` criteria are required because `QoS` represents a collection that is used twice in the query.

Priority and Relaxed Querying. This strategy involves priority values for single criteria in order to accomplish a weighted matching. Therefore, each `Match` criterion allows to append a weight to specify the priority of a criterion. In Listing 6.1, the priority values are "5", "3" and "1". In contrast, `Add` criteria do not allow to specify a weight because they are obligatory anyway. The *relaxed querying* strategy represents a special variant of the *priority querying*, in the sense that each `Match` criterion is treated with the same priority. Thus, this strategy simply distinguishes between optional and obligatory criteria in this regard. This strategy allows to define fuzzy queries by relaxing the query constraints which can be useful when no exact match can be found for a given query.

6.3.3 VRESCO **Mapping Framework**

The VRESCO Mapping Framework (VMF) defines the necessary concepts and mechanisms to handle the mapping from abstract features to concrete service operations from the metadata model as described in Section 6.2. The general concepts of the mediation approach implemented in VRESCO are shown in Figure 6.7.

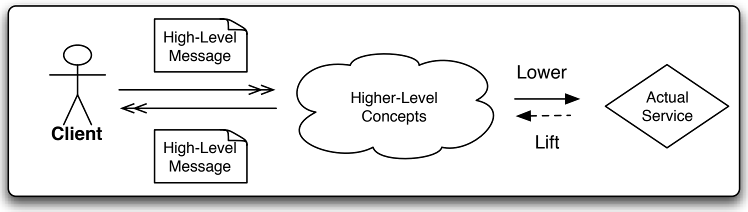


Figure 6.7: VRESCO Mediation Scenario (partly from [78])

A client who wants to invoke a service in VRESCO does not provide the input of the concrete service directly but already in the conceptual high-level representation, i.e., the feature input in VRESCO terminology. The runtime takes care of lowering and lifting of the feature input and output respectively. Lowering represents the transformation from high-level concepts into a low-level format (i.e., feature input to SOAP input) whereas lifting is the inverse operation (i.e., SOAP output to feature output). These mappings from features to concrete services are necessary to enable a runtime mediation when the abstract interface and the concrete service do not match. They can range from simple 1:1 mapping to very complex mappings requiring data transformation or custom mapping functions.

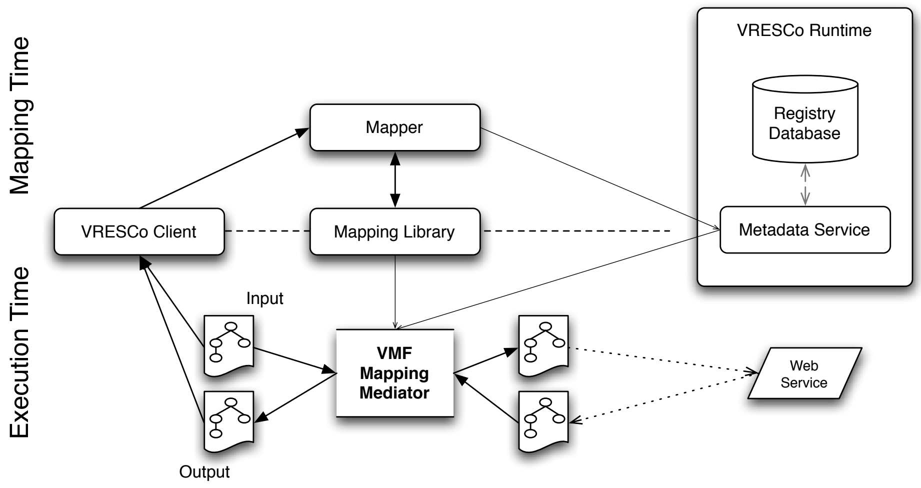


Figure 6.8: VMF Architecture

Figure 6.8 shows an overview of the VMF architecture. Generally, VMF comprises two main components. Firstly, at *Mapping Time*, the *Mapper* component is used to create lifting and lowering information for each service. These mappings may make use of the VMF *Mapping Library*, which includes a number of helpful predefined data manipulation operations. These operations implement some often-used data conversion functionality, such as data type conversion, string manipulation, mathematical functions or logical operators. We have summarized the groups of mapping functions provided in Table 6.3. Additionally, more complex mappings can be defined in the CSScript language [37]. Lifting and lowering information for services is stored in the VRESCo registry database using the *Metadata Service*. Secondly, at *Execution Time*, VMF provides a Daios mediator which is per default contained in the Daios chain of mediators of all clients using VRESCo. This mediator is responsible for the mediation itself. Therefore, it retrieves the stored lifting and lowering information from the *Metadata Service* at runtime, and interprets it.

Mapping Example. Listing 6.2 illustrates how a mapping (either lifting or lowering) is defined in VMF, based on the scenario in Figure 6.5. The feature `NotifyCustomer` requires the fields `Message`, `SenderNr` and `ReceiverNr` (data type *string*) as input. The `SendSMS1` operation of `SMSService1` requires the field `Message` (*string*), but sender and receiver number are splitted into area code and number (*integer*). Phone numbers contain an area code with four digits, followed by a number with eight digits. Line 4 shows how the mapper is created for feature `NotifyCustomer` and operation `SendSMS1`. Both objects have to be queried us-

Functions	Description
Constants	Define simple data type constants
Conversion	Convert simple data types to other simple data types
Array	Create arrays and access array items
String	String manipulation operations (`substring`, `concat`, etc.)
Math	Basic mathematical operations (`addition`, `round`, etc.)
Logical	Basic logical operations (`Conjunction`, `Equal`, `IfThenElse`, etc.)
Assign	Link one parameter to another (source and destination must have the same data type)
CSScript	Define custom C# mapping scripts which are executed by the engine

Table 6.3: Mapping Functions

```
1  // query NotifyCustomer and SendSMS1 instances using VQL
2  // ...
3  //create mapper from feature and operation
4  Mapper mapper = metadataService.CreateMapper(NotifyCustomer, SendSMS1);
5
6  //map feature message to operation message
7  Assign messageAssign = new Assign(
8    mapper.FeatInParams[0].GetChild("Message"),
9    mapper.OpInParams[0]);
10 mapper.AddMappingFunction(messageAssign);
11
12 //get area code, convert it to integer and map it to operation
13 Substring acSenderStr =
14   new Substring(mapper.FeatInParams[0].GetChild("SenderNr"), 0, 4);
15 acSenderStr = mapper.AddMappingFunction(acSenderStr);
16 ConvertToInt acSenderInt = new ConvertToInt(acSenderStr.Result);
17 acSenderInt = mapper.AddMappingFunction(acSenderInt);
18 mapper.AddMappingFunction(new Assign(acSenderInt.Result, mapper.OpInParams[1]));
19
20 //get sender number, convert it to integer and map it to operation
21 Substring senderNrStr =
22   new Substring(mapper.FeatInParams[0].GetChild("SenderNr"), 4, 8);
23 senderNrStr = mapper.AddMappingFunction(senderNrStr);
24 ConvertToInt senderNrInt = new ConvertToInt(senderNrStr.Result);
25 senderNrInt = mapper.AddMappingFunction(senderNrInt);
26 mapper.AddMappingFunction(new Assign(senderNrInt.Result, mapper.OpInParams[2]));
27
28 // the same mapping steps have to be done for RecipientNumber
```

Listing 6.2: VMF Mapping Example

ing VQL before the mapper can be created (not shown in Listing 6.2). The `Assign` function used in lines 7–10 acts as connector to link the `Message` from the feature to the `Message` of the operation, whereas `mapper.AddMappingFunction()` adds the function to the mapping. Lines 13–18 get the area code from the feature's `SenderNr` as substring and convert it with the `ConvertToInt` function to an integer which is finally assigned to operation's input field `AreaCodeSender`. From lines 21–26 the same is done to map the sender number.

6.3.4 Dynamic Binding and Invocation with DAIOS

DAIOS is a Web service invocation frontend for SOAP/WSDL-based and RESTful services. It supports fully dynamic synchronous and asynchronous invocations without any static components such as stubs, service endpoint interfaces or data transfer objects. It has been integrated as a main component in the VRESCO client library to facilitate dynamic invocation and client-side mediation as described in the previous section. In this section, we briefly discuss the DAIOS architecture and the available rebinding strategies to enable adaptive applications.

6.3.4.1 DAIOS Architecture

The overall architecture of the DAIOS framework is shown in Figure 6.9. It also illustrates how it fits into the SOA triangle of publish, find and bind. The framework internally consists of three functional components: the general DAIOS classes which are at the core of the framework and represent a facade for the other components, the interface parsing component which is responsible for preprocessing (binding) and the invoker component which conducts the actual Web service invocations using a REST or SOAP stack. Clients communicate with the framework frontend using DAIOS messages which are DAIOS' internal data representation format. The general structure of the framework is an implementation of the Composite Pattern for stubless Web service invocation (CPWSI) [25]. CPWSI separates the frameworks' interface from the actual invocation backend implementation, and allows for flexibility and adaptability.

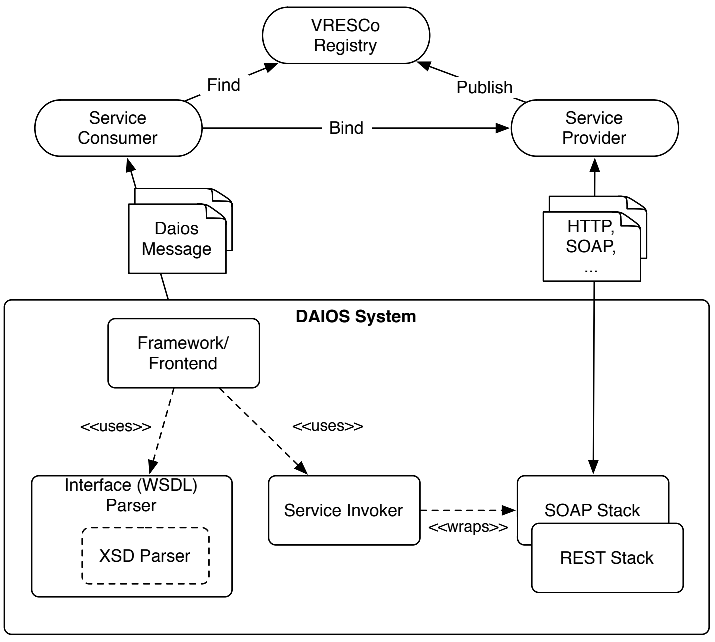


Figure 6.9: DAIOS Architecture

Daios is grounded on the notion of message exchange: clients communicate with services by passing messages to them; services return the invocation result by answering with messages. Daios messages are potent enough to encapsulate XML Schema complex types, but much simpler to use than working directly on XML level. Messages are unordered lists of name-value pairs, referred to as message fields. Every field has a unique name, a type and a value. Valid types are either built-in types (simple field), arrays of built-in types (array field), complex types (complex field) or arrays of complex types (complex array field). Such complex types can be constructed by nesting messages, therefore, arbitrary data structures can be built easily, without the need for a static type system.

Invoking a service in Daios is a three step process following the find-bind-execute cycle as shown on top of Figure 6.9. In the following, we emphasize the relation to VRESCo, however, it has to be noted that Daios is not natively coupled to VRESCo. All coupling code is achieved using specific interceptors.

1. First, a client has to find the corresponding services that need to be invoked. This step is not part of Daios, however, as shown above, this service selection step is achieved using VQL. The result of a VQL query is a `ServiceRevision` instance that can be directly transformed into a Daios proxy.

2. Second, the service has to be bound (preprocessing phase) by collecting and analyzing all necessary information such as the WSDL that will be compiled to retrieve the endpoint and type information.

3. The third and final step is the actual service invocation. In VRESCo, we never execute a service directly, however, we use the input of a feature to invoke the service. When invoking the service, the user has to provide the respective input according to the feature definition. Upon invocation, Daios invokes the VMF mediator to lower the input message for the actual target service. The response is lifted and then returned to the user.

```
1 var query = new VQuery(typeof(ServiceRevision));
2 query.Add(Expression.Eq("Operations.Feature.Name", "SendSMS"));
3 var proxy = querier.CreateRebindingMappingProxy(query, QueryMode.Exact, 0, new
      OnDemandRebindingStrategy());
4
5 DaiosMessage smsRequest = new DaiosMessage();
6 smsRequest.SetString("RecipientNumber", "0699-1234567");
7 smsRequest.SetString("SenderNumber", "0650-7654321");
8 smsRequest.SetString("Message", "Hello from VRESCo");
9
10 DaiosMessage smsResult = proxy.RequestResponse(smsRequest);
```

Listing 6.3: Daios Service Invocation

Listing 6.3 shows a simple example how to use Daios to dynamically invoke a service implementing a feature called `SendSMS` from the CPO example. In lines 1–2, a query is used for defining the feature that should be invoked (`SendSMS`). In line 3, this query is used to create a proxy using the *OnDemand* strategy. In lines 5–8, the input message for the `SendSMS` feature is built, and the corresponding service is finally executed in line 10 using the request-response pattern.

Once a service is successfully bound to a specific revision, clients can of course issue any number of invocations without having to re-bind again. Service bindings only have to be renewed if the interface contract of the service changes or the client explicitly decides to release the binding for some reason. In order to support different binding scenarios, VRESCo provides dedicated support for different scenarios as discussed in the next section.

6.3.4.2 Dynamic Binding Strategies

In service-oriented systems, dynamic binding is one of the key advantages to realize adaptive behavior without manual intervention. In practice, however, services are often bound using pre-generated stubs which lead to hard-wired applications. Therefore, we have introduced the notion of rebinding strategies implemented by providing several proxy classes using the well-known strategy pattern [54]. These proxies leverage the Daios framework for dynamically invoking a service by using the provided rebinding strategy. Table 6.4 summarizes all available rebinding strategies in VRESCo.

Strategy	**Rebinding Semantic**
`Fixed`	never
`Periodic`	periodically
`OnDemand`	on client requests
`OnInvocation`	prior to service invocations
`OnEvent`	on event notifications

Table 6.4: Rebinding Strategies

Fixed proxies represent ordinary proxies which are bound to one specific service endpoint. They are used in scenarios where rebinding is not needed or desired (e.g., because of existing contractual obligations or the lack of alternative services). *Periodic* rebinding can be used to verify the binding of a proxy periodically. This can cause a constant overhead and, clearly, this is inefficient if invocations happen frequently. The *OnDemand* rebinding strategy considers rebinding whenever the client instructs the proxy to do so by using a specific method invocation on the proxy instance. This results in low overhead but has the drawback that the binding is not always up-to-date. In contrast to this, *OnInvocation* rebinding updates the binding prior to every service request. It guarantees accurate bindings but seriously degrades the service invocation time. Finally, *OnEvent* rebinding uses the event notification engine [90] to combine the advantages of all strategies by allowing users to precisely define in which situations rebinding should be performed.

6.4 Evaluation and Discussion

In order to demonstrate the effectiveness of VRESCo's core services, we evaluated their performance by using a series of different tests cases. In this thesis, we only depict the performance of two specific core services, namely querying and dynamic invocation with meditation. For more comprehensive performance numbers of the other parts we refer to [91] and to Chapter 8 for the composition service. All the performance tests were executed on an Intel Xeon Dual CPU X5450 3.0 GHz with 32 GB of memory.

6.4.1 Querying Performance

This section gives the performance results of the querying engine which have been measured by querying for service revisions from a specific service owner that belong to a given category and have a certain response time. All measurements represent the average values of 10 repetitive runs. Table 6.5a compares the querying strategies provided by VQL depending on the number of service revisions in the database. It shows that EXACT querying is faster than RELAXED and PRIORITY which have similar performance characteristics. However, the difference between EXACT and RELAXED/PRIORITY is almost constant.

Revisions	**EXACT**	**RELAXED**	**PRIORITY**
1000	67,8	81,9	81,2
2000	123,4	131,6	134,3
3000	215,7	238,7	242,1
4000	299,4	328,4	330,2
5000	403,1	419,9	415,4
6000	480,2	503,0	515,3
7000	553,2	606,3	597,7
8000	646,6	706,8	710,3
9000	756,0	793,2	802,4
10000	806,9	824,7	836,7

(a) VQL Querying Strategies (in msec)

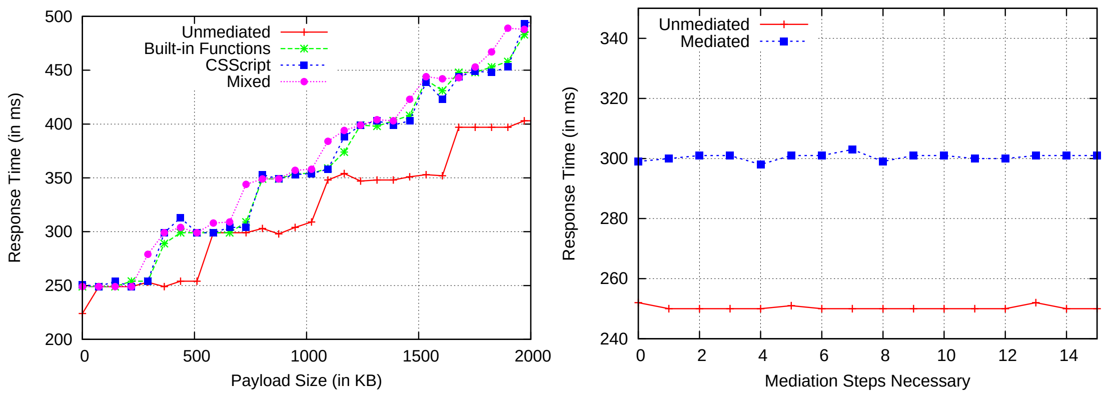
Revisions	**HQL**	**VQL**	**SQL**	**VQL/SQL**	**VQL/HQL**
1000	66,8	67,8	61,7	+9,89 %	+1,50 %
2000	118,6	123,4	116,6	+5,83 %	+4,05 %
3000	215,3	215,7	219,2	-1,60 %	+0,19 %
4000	301,2	299,4	294,9	+1,53 %	-0,60 %
5000	391,9	403,1	379,3	+6,27 %	+2,86 %
6000	464,6	480,2	463,9	+3,51 %	+3,36 %
7000	549,0	553,2	559,3	-1,09 %	+0,77 %
8000	645,6	646,6	642,0	+0,72 %	+0,15 %
9000	750,4	756,0	725,5	+4,20 %	+0,75 %
10000	822,6	806,9	771,2	+4,63 %	-1,91 %

(b) VQL Query Performance (in msec)

Table 6.5b shows the comparison between VQL, Hibernate Querying Language (HQL) and native Structured Query Language (SQL) using the EXACT strategy. For this experiment, we manually translated the query to both HQL and SQL to be able to compare the relative overhead of VQL compared to other query languages. The table shows that VQL queries are only slightly slower than native SQL queries, whereas VQL and HQL have similar performance characteristics. Due to the advantages in terms of flexibility and querying possibilities, this overhead is acceptable.

6.4.2 Mediation Performance

In the following subsection we have evaluated the overhead introduced by the VRESCo mediation facilities. Figure 6.10a depicts the response time of a single Web service invocation depending on the size of the message sent to the service. We have evaluated four different mediation scenarios: no mediation at all, mediation using only the VMF built-in functions, mediation using only CSScript and finally mediation using both built-in functions and CS-Script. Unsurprisingly, unmediated invocations are generally faster than any type of mediation. All types of mediation introduce a similar amount of overhead, which is dependent on the size of the message. For small messages the overhead is in the area of 25 msec, which is acceptable. However, the overhead slightly increases as the size of the data increases. This is due to data manipulation operations taking longer for bigger message sizes.



(a) Depending on Message Size

(b) Depending on Mediation Steps Necessary

Figure 6.10: Mediation Performance

In Figure 6.10b, we have evaluated how the overhead introduced by mediation depends on the amount of mediation necessary. As we can see, the mediation overhead is constant, and does not depend on the amount of mediation necessary, i.e., it is not relevant for the mediation overhead if only simple transformations or more complex ones are necessary. This result differs from what we have reported earlier in [78]. In this work, we have compared various DAIOS mediators including one based on SAWSDL [163] which is similar to the VMF approach from a conceptual point of view. Contrary to the constant overhead of the VMF mediator, the overhead of SAWSDL-based mediation increases (slightly) with the number of mediation steps.

Chapter 7

VCL - A Constraint-Based and QoS-Aware Composition Language

This chapter introduces the Vienna Composition Language (VCL), a domain-specific language designed for the purpose of specifying QoS-aware composite services on top of the VRESCo platform. We introduce the language and its model as a basis for the *Composition as a Service* approach.

Contents

7.1 Motivation

A number of service composition approaches exist for developing and executing composite services. Most languages are static in the sense that they only focus on functional aspects and they do not provide any adaptive behavior. This static nature poses a real problem, for example, when services need to be dynamically selected or exchanged from a pool of similar services based on changing QoS attributes such as response time, throughput or availability (cf. Chapter 3). Several approaches address adaptivity of compositions on a runtime level to increase availability or fault-tolerance of a composite application [51, 99], e.g., by providing custom extensions to BPEL engines. However, none of the existing approaches addresses the problem of how to specify QoS-aware service compositions on a microflow-level by allowing developers to express the required QoS of a composed service. In existing languages, requirements mostly focus on functional aspects. In this approach, we also consider non-functional aspects (QoS) as a major concern when developing composite services by proposing a constraint-based approach. It allows a natural way of specifying QoS requirements for composite services. In existing approaches, a major problem when specifying QoS as constraints is the fact that the runtime resolution will typically fail if a QoS attribute of a service cannot be satisfied (e.g., if the desired availability is 0.99, but the actual availability of a given service is 0.98). To deal with this issue, we leverage a combination of hard and soft constraints and combine these two constraint types in a so-called constraint hierarchy to allow a fine-grained distinction of the importance of a constraint [23].

These aforementioned issues are addressed by providing a domain-specific language (DSL) called VCL (Vienna Composition Language). This language is the basis for the "Composition as a Service" approach presented in Chapter 8 that provides the back-end for VCL. It implements the QoS-aware optimization and generation of the executable composition based on the VCL specification in a so-called *Composition Service*. This approach is implemented on top of VRESCo (cf. the *Composition Service* in Chapter 6).

7.2 Vienna Composition Language

The main goal of VCL is to provide an intuitive DSL for the purpose of composition within the VRESCo environment. It enables to capture what a composition should do and what QoS is required from a global perspective and also what QoS is required from individual services in the composition (local perspective). Additionally, constraints on individual services can be imposed, for example on inputs and outputs of each service, preconditions or postconditions that have to be fulfilled. A main concept in VCL is the fact that we group QoS constraints into constraint hierarchies to address the problem that not all QoS attributes can always be fully satisfied and not necessarily have to be. Therefore, constraint hierarchies provide a mechanism to combine hard and soft constraints by using different hierarchy levels that express the relative importance of a constraint over others (where constraints on the highest level are always required). Consider an example, where a user specifies two global constraints on a composition expressing that the response time should be $\leq 5000msec$ and the availability should be ≥ 0.95. Unfortunately, the composition system cannot fulfill the availability constraint because its actual value is 0.935, thus, the composition process fails due to a violating QoS constraint. By using constraint hierarchies one can add a strength value to QoS attributes to express its importance in a hierarchical way. Traditional constraint-based approaches usually fail when a constraint is violated and no solution exists that fulfills all constraints. Such systems of constraints are called over-constrained systems. Constraint hierarchies [23] have been proposed to solve such systems by associating a strength or preference value with each constraint. Formally, a constraint hierarchy H is a multi-set of labeled constraints. H_0 denotes the required constraints in H. The sets H_1, H_2,…,H_n are defined for the hierarchy levels 1,2,…,n representing the optional constraints with different strengths. Each level expresses constraints that are equally important. In VCL the hierarchy levels are labeled $\{required, strong, medium, weak\}$ but this can be adapted to represent different hierarchy levels if required.

Overall, a constraint-hierarchy based way of specifying QoS for composite services leads to more flexibility in terms of its specification. Many QoS attributes are "nice to have", therefore, hard constraints fail to deliver the expected results. Additionally, constraint hierarchies provide a sound and easy way for developers to specify the strength of a QoS attribute in a "symbolic way" relative to other QoS attributes. From an end-user perspective, it provides an easy way to specify the relative importance of a QoS constraint over another one without dealing with different numerical weighting values (that are used to directly influence to QoS optimization).

It is important to note that VCL does not aim at defining yet another composition language and engine. The focus is put on the QoS-aware specification, therefore, we do not provide a VCL execution environment. In order to execute a VCL specification, we leverage the capabilities of the Microsoft Windows Workflow Foundation [96] combined with VRESCo's registry capabilities and service runtime capabilities. Therefore, a VCL specification will be transformed to a Windows Workflow at runtime which is described in the next chapter.

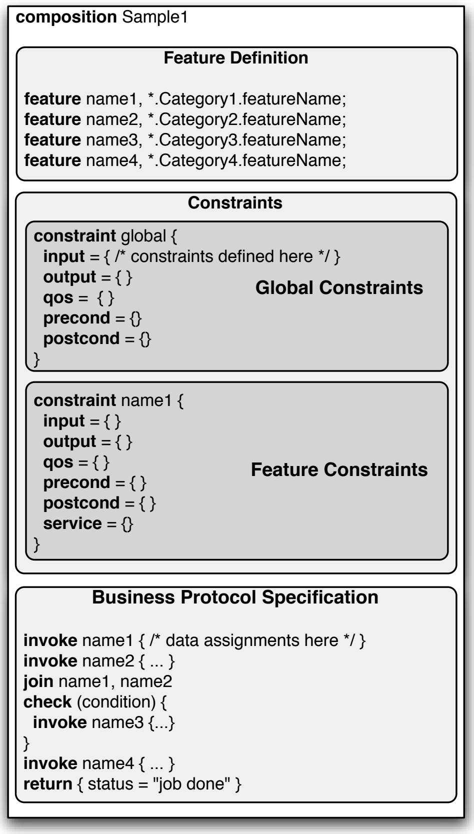
7.2.1 Overview and Structure

The language follows a constraint-based approach in the sense that all required functional and non-functional aspects of a composition are declared as constraints on *features* that should be composed. This notation is based on VRESCo's service- and metadata model presented in Section 6.2 which represents an integral part for the overall VCL approach (and also the CaaS approach presented in the next chapter). VCL has to rely on a strong runtime support that is able to generate an executable composite service from the VCL description. The core elements of a VCL specification are visualized in Figure 7.1 and described in detail in the subsequent sections.

More formally, a VCL specification of a composite service CS can be represented as a tuple $CS =< FD, FC, GC, BPS >$ with the following elements:

Feature Definitions. FD represents the *feature definitions* that specify which features will be composed. Each feature has an associated *category* and the definition of features and categories follows the notation we introduced in the VRESCo metadata model. Due to the fact that categories can have multiple subcategories (as denoted in Figure 6.3), the specification of categories allows a wildcard character $\star$ to refer to a specific category within the category tree without specifying the whole path.

Feature Constraints. FC represents a set of (optional) *feature constraints* that can constrain the input, output, QoS, precondition and postcondition of each feature. The input and output constraints restrict the data which is required as an input or output of a feature. Preconditions and postconditions can be used to express assertions that need to be valid before or after the execution of a feature, respectively. QoS constraints can be associated with a strength to



```
composition Sample1

feature name1, *.Category1.featureName;
feature name2, *.Category2.featureName;
feature name3, *.Category3.featureName;
feature name4, *.Category4.featureName;

constraint global {
  input = { /* constraints defined here */ }
  output = { }
  qos =  { }
  precond = {}
  postcond = {}
}

constraint name1 {
  input = { }
  output = { }
  qos = { }
  precond = { }
  postcond = { }
  service = {}
}

invoke name1 { /* data assignments here */ }
invoke name2 { ... }
join name1, name2
check (condition) {
  invoke name3 {...}
}
invoke name4 { ... }
return { status = "job done" }
```

Figure 7.1: VCL Language Schema

express its preference ("required" strength is the default). Additionally, a service constraint can be used to influence the service selection by defining explicitly which service should be selected.

Global Constraints. GC represents the *global constraints* for the overall composition. Similar to feature constraints, global constraints can be used to restrict input, output, QoS and specify pre- and postconditions for the resulting composite service CS. Input and output constraints are used to define input and output data of the CS, i.e., the composed service interface. Again, QoS constraints can be associated with a strength value.

Business Protocol Specification. BPS represents the *business protocol specification* [5] that defines what services should be invoked and it provides a means to specify simple conditional execution or loops. The business protocol specification does not have an explicit notation for specifying which parts are executed sequentially and which parts can be executed in parallel (i.e., we do not have AND-splits, XOR-splits as explicit constructs [150] as many graph-based approaches). Such an "unstructured composition" approach enables a simpler specification from a user perspective, however, it may lead to errors during the execution (e.g., deadlocks) if not properly validated. Thus, a transformation to a structured composition – originally referred to as structured process model [74], where each split has a corresponding join and all split-join combinations are properly nested – is desirable and performed as part of the composite service generation. For example in Figure 7.1, we have sketched several statements illustrating various aspects of a control flow definition, e.g., the invocation of the given feature using `invoke` (possible data assignments are not shown in the figure). They can either be executed sequentially or in parallel, depending on the data flow. If, for example, feature "name2" has no dependencies on "name1" they both can be executed in parallel. VCL is flexible in terms of the concrete BPS formalism. Currently, we use an unstructured approach to specify the BPS and transform it to a graph-based model at runtime. However, the BPS part can be exchanged to use a label-transition system or any other formalism if desired.

7.2.2 Grammar and Language Constructs

As illustrated above, each VCL specification consists of three major blocks: *feature definition*, (global and feature) *constraints* and the *business protocol*. The following grammar (in EBNF notation) in Listing 7.1 shows the formal specification of the core parts of VCL.

```
<Composition> ::= composition <Name>;
                  <Feature> { <Feature> }
                  <GlobalConstraint> { <FeatureConstraint> }
                  <BusinessProtocol>
<Feature> ::= feature <Alias>, <CategoryName>.<FeatureName>;
<GlobalConstraint>  ::= constraint global "{" <ConstraintBody> "}";
<FeatureConstraint> ::= constraint <Alias> "{" <ConstraintBody> "}";
<ConstraintBody>    ::= <InputConstraint> <OutputConstraint>
                        <PrecondConstraint> <PostcondConstraint>
                        <QoSConstraint> <ServiceConstraint>
```

Listing 7.1: VCL Grammar

Each composition (non-terminal symbol `<Composition>`) composes a set of VRESCo features (`<Feature>`), defines global constraints (`<GlobalConstraint>`) and optionally a set of feature constraints (`<FeatureConstraint>`). A feature is specified by an `<Alias>`, a category name followed by the name of the feature that is stored in the VRESCo registry.

Each constraint type (global and local) imposes restrictions on the input and output, pre- and postconditions and on QoS (cf., `<ConstraintBody>`). Input and output statements of a global constraint can be used to define the interface of the composed service, whereas input and output statements on a feature are used to constrain the feature input and output that is required in the composition. The final part is the specification of the business protocol (`<BusinessProtocol>`) describing how the services should be invoked and which data is assigned to the services. This is achieved by providing typical control flow constructs such as conditionals (`check` statement), loops (`while` statement) and a statement to invoke a service (`invoke` statement).

In the following, we use small examples from the *Number Porting Process* introduced in Section 6.2 to explain the different language constructs, without using EBNF. For a more complete version of the grammar we refer to the VCL specification [128].

7.2.2.1 Feature Definition

The first part of each VCL specification is – besides the obligatory definition of the name of a composition – the specification of features that are composed. Since VCL is targeted for the use with VRESCo, it is obvious that the features have to be defined in the VRESCo registry. In Listing 7.2, the definition of features for the *Number Porting Process* from Figure 6.2 is depicted.

```
composition NumberPorting;

feature customerLookup, *.CRMServices.CustomerLookup;
feature partnerLookup, *.NumberPorting.PartnerLookup;
feature portcheck, *.NumberPorting.PortabilityCheck;
feature port, *.NumberPorting.PortNumber;
feature activate, *.PhoneManagementServices.ActivateNumber;
feature notify, *.NotificationServices.Notify;
```

Listing 7.2: Feature Definition

For each of the six steps in the number porting process, a feature is defined which has to be available in VRESCo. If this is not the case, the composition service will fail and return that a given feature cannot be found. Each feature is referenced by using an alias e.g., `customerLookup` to refer to a given feature. Using aliases is necessary to use one specific feature more than once in the same VCL specification in different contexts (i.e., it is similar to variables in programming languages that can reference instances of a specific type). Following the alias definition, the corresponding category and the name of a feature have to be defined. We use a dotted notation to allow users to specify hierarchical category identifiers. Additionally, the $\star$ character can be used as a wildcard in the hierarchical category specification. In case of the `customerLookup`, we use `*.CRMServices` to specify that the feature should be from a category `CRMServices` irrespective of the hierarchy level (e.g., `CRMServices` can have an arbitrary number of parent categories). Furthermore, the user can also simply specify the wildcard to express that the feature can be in any arbitrary category. At the end of dotted category specification is the feature name, such as `CustomerLookup`.

7.2.2.2 Global Constraints

The goal of global constraint specification is twofold. On the one hand it is used to specify global QoS properties that have to be satisfied by the composition runtime. In our approach, we allow to use all service level QoS constraints from Chapter 3. On the other hand, it can be used to define the external interface of the composed service using input and output constraints. Additionally, pre- and postconditions for the composed service can be specified that define what global state has to hold before and after the execution of a composed service, however, we have not fully exploited the ability of using pre- and postconditions on a composition level.

In Listing 7.3, the global constraint specification for the number porting process is depicted. From lines 12–18 and 19–23 the input and output constraints are shown. Both implicitly define a message type (`NumberPortingRequest` and `NumberPortingResponse`) that is used as the input and output of the composed service.

```
 9 ### continued from Listing 7.2
10
11 constraint global {
12   input = {
13     NumberPortingRequest[
14       long customerId;
15       string providerName;
16       boolean keepPhoneNumber;
17     ]
18   }
19   output = {
20     NumberPortingResponse [
21       string status;
22     ]
23   }
24   qos = {
25     responseTime = 5000, required;
26     availability = 0.95, strong;
27     accuracy = 0.99, weak;
28   }
29 }
```

Listing 7.3: Global Constraints

From lines 24–28, the QoS constraints for the composition are defined. Basically, they state that the response time of the overall composition should not be higher than five seconds and the overall availability should be higher than 95% and the accuracy higher than 99%. It is important to note that VCL only uses the equal sign in the specification of QoS constraints. It is determined based on the QoS attribute dimension if a lower or a higher value is generally better (cf. the dimension column in the summary of service layer QoS attributes in Table 3.1).

In this example, the response time constraint is marked as `required` (which is default anyway), therefore, the QoS-aware optimization algorithm that is implemented by VRESCo's *Composition Service* has to find a selection of services that satisfy this hard constraint, otherwise, the composition process fails. The availability constraint is only marked as `strong`, thus representing a soft constraint and it will be satisfied if possible, however, it will not lead to an unsatisfiable specification if it cannot be satisfied. Additionally, a `weak` constraint for the `accuracy` with a value of 0.99 is specified. Because this constraint is specified as `weak`, preference will be given to the `availability` constraint during the optimization process.

7.2.2.3 Feature Constraints

This type defines, as the name implies, constraints on features which have been defined at the beginning of a VCL specification. The constraint types are basically the same as for the global constraints (input, output, precondition, postcondition), however, a service constraint is added to address the issue of selecting a specific service rather than letting the dynamic binding mechanism choose one of the available services. In Listing 7.4 an example feature constraint is depicted. Similar to the global constraint, it defines an input constraint (lines 33–40) to express the required input of a feature which is needed in the composition. The runtime has to find the required match or return an error if it cannot be found. In this example, no output constraint is required because it defines an asynchronous feature. The required QoS constraint (lines 41–43) specifies that the notify feature has to support an X.509 certificate. The service constraint (lines 44–46) expresses that we specifically want the service with the name `SMSNotificationService` due to the fact that we want SMS delivery instead of e-mail or any other notification type.

```
30 ### continued from Listing 7.3
31
32 constraint notify {
33   input = {
34     NotificationRequest[
35       long customerId;
36       string senderNumber;
37       string receiverNumber;
38       string message;
39     ]
40   }
41   qos = {
42     security = X509, required;
43   }
44   service = {
45     name = SMSNotificationService;
46   }
47 }
```

Listing 7.4: Feature Constraints

We are aware that this service constraint to some extend violates the principle of our feature-driven model, however, in some cases it is necessary to use a service constraint. An alternative for using a service constraint to specifically select an SMS service is to define different sub-categories for the `NotificationServices` category in VRESCo (e.g., for e-mail, fax and SMS). To this end, the service constraint could be removed and the feature definition has to be changed to reflect the selection of a `Notify` feature in the corresponding sub-category.

7.2.2.4 Business Protocol Specification

The final part of each VCL specification is the business protocol specification. For our purposes, this part is kept simple, because on a microflow-level the control-flow logic requirements are not very sophisticated. As mentioned earlier, the business protocol specification does not define any constructs to explicitly specify what parts have to be executed sequentially or in parallel. The execution order and semantics are determined in the *Composition Service* by performing a data flow analysis to check which invocations can be executed in parallel and which have to be executed sequentially (because they have a data flow dependency).

VCL supports different constructs to specify the business protocol as summarized in Table 7.1.

Statement	Purpose and Semantics
`invoke featureName { data assignment }`	Invokes a feature
`check(expression){ block }else[p]{ ...}`	Conditional execution
`while[c](expression){ block }`	Loops
`return parameter`	Returns some data at the end of a composite service
`throws errorMsg`	Used to raise an error
`join f1, f2, ...`	Explicitly waits for an invocation to finish

Table 7.1: Statements for the Business Protocol Specification

One of the most important statements is the `invoke` statement, which is used to perform an invocation of a service that implements a given feature. The runtime is responsible for resolving all services that implement a feature and match all the constraints (see Chapter 8). An invocation is issued by using the name of the feature and assign the required data as shown in Listing 7.5. It simply states that the customer feature should be invoked. The specification of the `CustomerLookupRequest` has to match the input that is associated with the feature in the VRESCo registry. It implicitly creates the necessary message type and assigns the `customerId`, which was defined in the global input constraint in Listing 7.3, to the `Cid` element (lines 53–57). The data type is implicitly determined, similar to existing scripting languages. The result of this invocation is implicitly available and accessible using the alias name as shown in line 78 and 80 where we assume that the `customerLookup` feature returns a type `CustomerDetails` that is then used to notify the customer about the outcome of the number porting operation. Its child elements can be accessed in a similar way as most programming languages access fields of an object.

Conditional execution can be handled by using the `check` statement (cf. lines 64–91). It evaluates a boolean expression and in case it is true it performs the containing block otherwise an optional `else` branch can be executed. The else branch can be annotated with an execution probability `[p]` (cf. Table 7.1) to influence the QoS aggregation as shown in Chapter 3. The execution probability of the check-branch is simply computed by 1-p.

Loops can be realized using a `while` statement and are specified in a similar way as the conditional statement. Again, the initial loop count can be annotated by using the loop count `[c]` (cf. Table 7.1). The flow of control can be terminated by using either the `return` statement or the `throw` statement. The former returns the control to the caller and additionally allows to associate data with the return whereas the latter statement is used to return an error including associated data to provide exception details. An example of using a return statement is shown from lines 86–90, where the outcome of the porting invocation is returned. An exception is shown from lines 94–97 to signal that the porting could not be performed including some application specific error code and a explanatory message.

The last statement from Table 7.1 is the `join` statement. It allows to explicitly specify a

```
### continued from Listing 7.4

# invoke the customer lookup feature as defined above
invoke customerLookup {
  CustomerLookupRequest[
    Cid = customerId;
  ]
}

invoke portcheck {
  # data assignments omitted
}

# check if porting can be done (i.e., old provider is ready)
check (portcheck.PortingResponse.Status = true) {
  # perform the porting
  invoke port {
    # omit data assignment
  }

  # activate number in new GSM network
  invoke activate {
    # omit data assignment
  }

  # notify the customer that the number has been ported
  invoke notify {
    NotificationRequest[
      customerId = customerLookup.CustomerDetails.Id;
      senderNumber = "+43-699-HELPDESK";
      receiverNumber = customerLookup.CustomerDetails.Number;
      message = "Your number has been ported and is now ready to use.";
    ]
  }

  # return the outcome of the porting
  return {
    NumberPortingResponse [
      status = port.PortingResponse.Status;
    ]
  }
}

# signal an error in case porting is not possible
throw NumberPortingFault [
   errorcode = "1034";
   message = "Porting is not yet possible";
]
```

Listing 7.5: Business Protocol Specification Example

synchronization point between different invoke statements. For example, if there are two consecutive `invoke` statements for feature *a* and *b* and they have no data dependency, they are executed in parallel. However, in some cases it is necessary to explicitly wait for one or more invocations (e.g., if a login feature has to be invoked before invoking some other feature).

7.3 Implementation

VCL is implemented as a domain-specific language on top of the Microsoft Oslo modeling framework [95], and MGrammar in particular. The Oslo framework aims at simplifying model-driven development and enables developers to visually define and interact with models. Additionally, it proposes MGrammar as a framework to build textual domain-specific languages.

In our approach we leverage MGrammar to specify the VCL grammar in the proprietary Mg format [94]. Based on the MGrammar definition, a dynamic parser component is available as part of the framework that takes the VCL grammar and provides dynamic parsing and validation capabilities for VCL input. Mg transforms VCL input into structured data. The shape and content of that data is determined by the VCL syntax rules.

There are two possibilities to process the Mg output: Firstly, one can implement a custom `GraphBuilder` to traverse the output graph (called MGraph) and process it accordingly. Secondly, one can leverage an automated mapping of the Mg output to an C# object model by using XAML (eXtensible Application Markup Language), an XML-based language from Microsoft to represent structured objects and their values. We leverage the second possibility because it makes it easier generate a C# object model without implementing a custom parser. To do so, MGrammar executes the following steps:

(i) it parses VCL code using the dynamic parser and generates an MGraph representation (it is basically a structured and hierarchical textual representation);

(ii) it converts the MGraph representation to a XAML-based representation;

(iii) finally, it converts the XAML representation to strongly typed C# objects.

```
module VRESCo {
  language VCL {
    // Initial rule
    syntax Main = c:Composition
                  f:FeatureDefinition+
                  con:ConstraintDefinition*
                  wf:(id:InvocationDefinition =>id | wd:JoinDefinition => wd |
                      cd:CheckDefinition =>cd | rd:ReturnDefinition =>rd |
                      td:ThrowDefinition =>td | wd:WhileDefinition =>wd)*
      => Composition {c, Features{f}, ConstraintCollection{con}, BusinessProtocol{
          wf}};

    // main rules
    syntax Composition = CompositionToken name:Identifier ";" => Name{name};
    // other syntax and token definitions go here\ldots
  }
}
```

Listing 7.6: VCL MGrammar Example

Listing 7.6 shows a small excerpt of VCL's Mg definition. From lines 4-10 the main syntax rule for VCL is defined. It is easy to see how the grammar reassembles the main EBNF rule from Listing 7.1. The part after the => defines the shape of the Mg output. For example the `Composition` defines the root of a hierarchical output containing a name (from the `Composition` production rule), a set of features, a collection of constraints and a business protocol. We do not process this output manually, however, we let MGrammar handle it by executing the three steps as illustrated above. Therefore, we have implemented a C# object model where the class names and their fields exactly match these output tokens (such as `Composition`, `Features`, `ConstraintCollection`, etc). Following that, the parser needs to know the mapping from the output to the C# types which is done by using a hashtable that is given to the Mg parser. This information is sufficient to parse and dynamically create this object model that is used for all further processing within the *Composition Service*.

7.4 Evaluation

We have evaluated the performance of the VCL parsing and the creation of the object model that is used for all further processing steps in the *Composition Service*. It has to be noted that we cannot directly influence the parsing performance, because the parsing component is implemented by the Oslo framework. The major time is not used for parsing the VCL input, indeed it is used for binding the MGrammar output to our C# object model using the XAML data binding.

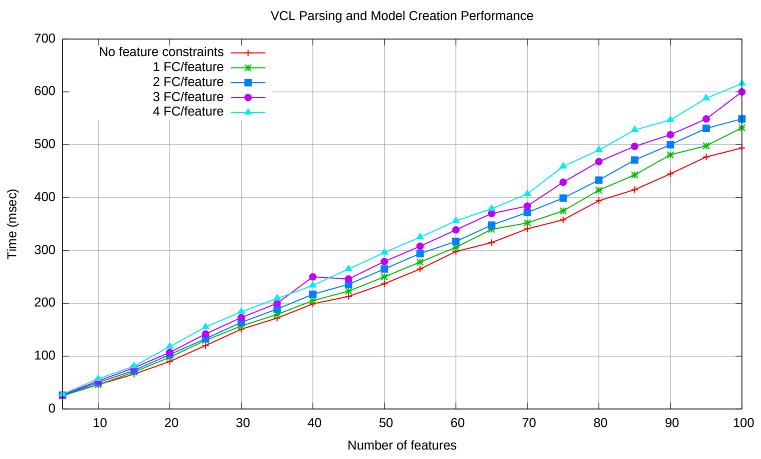


Figure 7.2: VCL Parsing and Model Creation Performance

All the performance tests were executed on an Intel Xeon Dual CPU X5450 3.0 GHz with 32 GB of memory (although memory is not an issue in our tests). We have generated 20 different VCL files, each containing a increasing number of features (from 5 to 100 in increments of 5 features). Each VCL file contains 4 global constraints a varying number of feature constraints (from 0 to 4), resulting in five different graphs in the plot in Figure 7.2.

The results show that the time needed for parsing and creating the VCL object model grows linear with the number of features. The number of feature constraints per feature clearly has a small influence on the parsing performance since more input needs to be parsed and more objects need to be created. The difference for a VCL specification with 100 features is approx. 100 msec between no feature constraints at all and four feature constraints per feature. Since we use a first version of the Oslo framework and MGrammar, we assume that there is some room for improvement with regards to the parsing and model creation performance, however, for our purposes this performance is sufficient.

Chapter 8

Composition as a Service using VCL

This chapter introduces a "Composition as a Service" approach as a means to reduce the complexity of QoS-aware service composition on top of VRESCO. The approach leverages VCL as a specification language to enable developers to compose services on-the-fly without the need to provide their own composition infrastructure.

Contents

8.1 Motivation

The Composition as a Service (CAAS) approach is based on the idea of reducing the complexity involved when developing QoS-aware composite applications. Existing composition approaches are either purely static (e.g., BPEL) or they tend to be fully automated, e.g., approaches such as AI planning [123], situation calculus [85], automaton theory [20, 21], and hierarchical task networks [142]. From a technical perspective, many approaches are very complex and pose some strong requirements on the underlying services (e.g., fully annotated with ontologies) and the infrastructure (e.g., reasoning capabilities). Moreover, existing approaches do not support an explicit notation or language for specifying QoS-aware composite applications. In addition to the specification aspect of a composition, we provide composition as a service to reduce the need and the complexity involved with setting up a composition environment that combines hard and soft constraints to enact composite services.

The CAAS approach proposes a simpler composition model and runtime on top of VRESCO to facilitate rapid and low-cost QoS-aware composition on a microflow level. A *Composition Service* encapsulates the complexity of the composition approach and the management of the composition engine. The compositions are specified using VCL and are generated and deployed on-the-fly by using VRESCO's composition infrastructure. Once a composition is deployed, the developer directly receives the newly deployed endpoint as a response, thus, the deployed service is immediately available and usable.

8.1.1 CaaS Overview

An overview of the CAAS approach is depicted in Figure 8.1. From an end-user perspective, the developer (on the client-side) specifies the composition in VCL and uses the client library to invoke the composition service at the VRESCO runtime. Besides providing a convenient way to access the VRESCO core services (such as publishing, metadata, querying, etc), the client library compiles the VCL specification and checks for static errors to avoid invoking the composition service using invalid input. Once a statically correct VCL specification is sent to the VRESCO runtime, the five steps (a) to (e) in the gray box on the left side of Figure 8.1 have to be executed to successfully deploy and provision a composite service.

(a) *Feature Resolution:* This step comprises the resolution of features that are required for generating a valid composition. Resolving all features implies a translation of feature requirements into a VQL query executed by the VRESCO querying service.

(b) *Structured Composition Generation:* In this step, the VCL specification is transformed into a so-called structured composition [49] to enable QoS aggregation and provide the basis for transforming the VCL input into an executable composition. This involves a data flow analysis to generate a dependency graph for the features and the transformation to a structured composition.

(c) *QoS Aggregation and Optimization:* Once all features and data dependencies have been resolved, this step transforms the VCL specification into a QoS optimization problem and uses QoS aggregation to calculate the QoS of the overall composition. If no solution can be found an error will be raised by sending a notification back to the user to allow changes, e.g., by relaxing some constraints.

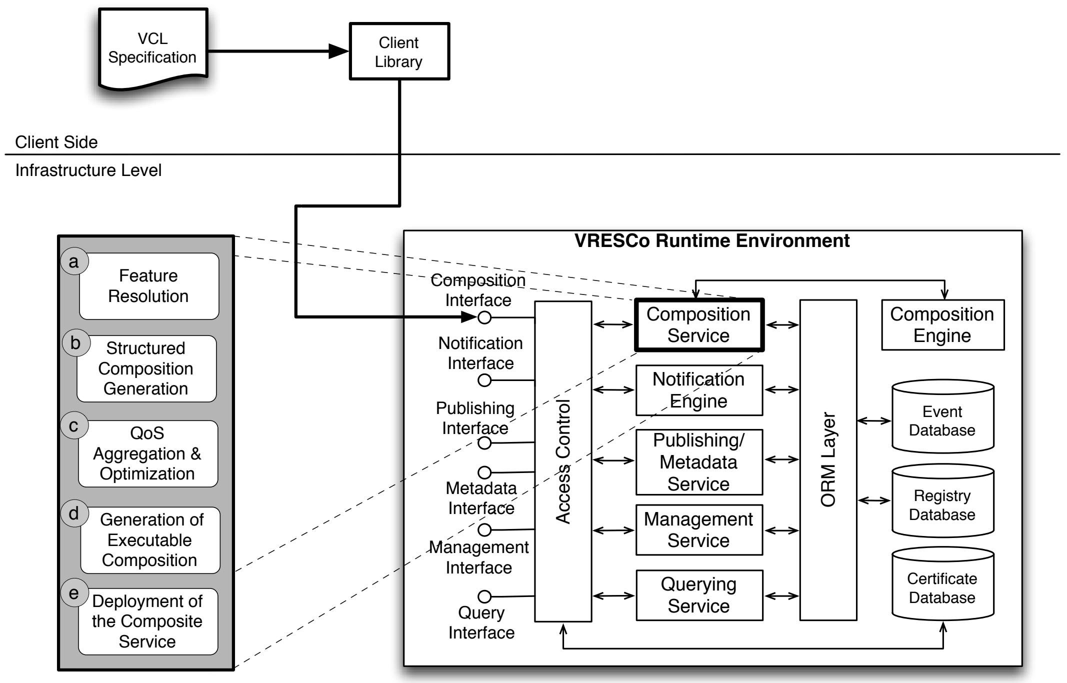


Figure 8.1: Architectural Overview of Composition as a Service with VCL

(d) *Generation of the Executable Composition:* Assuming all constraints are satisfied, the *Composition Service* triggers the generation of the executable composite service by using the structured composition to generate Microsoft Windows Workflow Foundation code.

(e) *Deployment of the Composite Service:* A successful composite service generation leads to the final step by deploying the generated composite service and sending the newly deployed service endpoint back to the user. Additionally, the new service is registered in the VRESCo registry using the publishing service.

The remainder of this chapter focuses on these five steps, their implementation and evaluation on top of the VRESCo runtime environment as introduced in Section 6.

8.1.2 Formal Composition Model

Before going into the details of the QoS-aware optimization approach, we need to formalize our composition model to have a common notation throughout this chapter. A VCL composition CS_{vcl} consists of a set of n features $F = \{f_1, f_2, \dots, f_n\}$ to be composed. For each feature f_j, a set of m service candidates $S_j = \{s_{1j}, s_{2j}, \dots, s_{mj}\}$ is available in VRESCo that implement a given feature. Each composition CS_{vcl} can be subject to global constraints $C_{gc} = \{I_{gc}, O_{gc}, P_{gc}, E_{gc}, Q_{gc}\}$. Each feature f_j can also have a set of constraints $C_{fc} = \{I_{fc}, O_{fc}, P_{fc}, E_{fc}, Q_{fc}\}$. These constraints represent a multi-set containing input constraints

I, output constraints O, preconditions P, postconditions E (effects), and QoS constraints Q. Constraints I, O, P, E specify restrictions on data of a feature or the composition itself and are not further considered in this thesis because they do not play a major role during the QoS optimization process. The QoS constraints Q_{gc} (global QoS constraints) and Q_{fc} (feature QoS constraints) are represented as a vector of labeled QoS constraints:

$$Q_{gc} = Q_{fc} = (\langle q_{pt}, s\rangle, \langle q_{ex}, s\rangle, \langle q_{l}, s\rangle, \langle q_{rt}, s\rangle, \langle q_{rtt}, s\rangle, \langle q_{tp}, s\rangle, \langle q_{sc}, s\rangle, \langle q_{av}, s\rangle, \\ \langle q_{ac}, s\rangle, \langle q_{ro}, s\rangle, \langle q_{rm}, s\rangle, \langle q_{sec}, s\rangle, \langle q_{rep}, s\rangle, \langle q_{c}, s\rangle, \langle q_{pl}, s\rangle) \quad (8.1)$$

The first element of each pair is the QoS attribute value and the second element $s \in H$ represents the constraint strength as defined in the hierarchy H (see Chapter 7). Additionally, each service candidate s_{ij}, implementing a given feature f_j, has a vector of QoS values (retrieved from the VRESCo registry). Based on [176], we merge the service candidates with their QoS attributes in a matrix $Q = (Q_{ij}; 0 \leq i < n; 0 \leq j \leq 14)$. Each row corresponds to a service candidate, whereas each column corresponds to all the QoS values of a service candidate s_{ij}. We assume that the QoS attributes are numbered from 0 to 14 according to their order in Table 3.1.

8.2 Feature Resolution and Pre-filtering

Feature resolution is the process of querying and matching all service candidates and its constraints as specified for each feature in VCL. We assume that each feature of an application is defined in the VRESCo metadata model as part of the requirements engineering process. Additionally, we apply a pre-filtering technique to filter service candidates that do not fulfill feature QoS requirements with strength `required`. This reduces the number of service candidates for each features that are later used in the optimization process, thus, speeding up the optimization process.

The feature resolution is achieved by using the VRESCo Query Language (as introduced in Chapter 6). In general, VQL can query any element from our metadata model (e.g., features, services, etc), however, in this specific case, we need to retrieve a list of `ServiceRevision` instances because it contains all the required information for further processing (such as QoS) within the *Composition Service*.

Algorithm 4 outlines the basic procedure for building a feature resolution expression. In particular, the function ResolveFeature is executed concurrently for every feature $f_j \in F$ to find its service candidates. Each query itself is constructed by adding various query criteria stored in the *query* variable.

The first query criterion in line 4 is `Expression.Eq("IsActive", true)`. It specifies that a service has to be activated to be invokable (`Eq` means equals). `IsActive` refers to a property in a `ServiceRevision` as defined in the VRESCo service model.

Line 5 specifies a more complex query criterion. Here `Operations` refers to a collection of operations for a `ServiceRevision`. Each operation has a reference to a `Feature` that it implements, therefore, we can simply query its `Name` and match it against the feature name f_j. We use the helper function $name(f_j)$ to retrieve the name of a feature.

Algorithm 4 Feature Resolution

```
1: function ResolveFeature(Feature $f_j$, $C = \{I_c, O_c, P_c, E_c, Q_c\}$)
2:     result ← ∅                                                    ▷a list of service candidates for $f_j$
3:     query ← ∅                                                     ▷contains a set of VQL expressions
4:     query ← query ∪ Expression.Eq("IsActive", true)
5:     query ← query ∪ Expression.Eq("Operations.Feature.Name", $name(f_j)$)
6:     if $f_j$ has a category defined then
7:         query ← query ∪ Expression.Eq("Service.Category.Name", $cat(f_j)$)
8:         i ← 0
9:         for all $sc \in supercat(f_j)$ and $sc$ is not a wildcard do
10:            query ← query ∪ Expression.Eq($createSuperCatString(i)$, $sc$)
11:            i ← i + 1
12:        end for
13:    end if
14:    for all $q \in Q_c$ such that $strength(q)$ = required do      ▷process all required QoS attributes
15:        if $descendingQoS(q) = true$ then
16:            query ← query ∪ Expression.And(
17:                                Expression.Eq("Operations.QoS.Property.Name", $name(q)$),
18:                                Expression.Le("Operations.QoS.DoubleValue", $value(q)$)))
19:        end if
20:        if $asecendingQoS(q) = true$ then
21:            query ← query ∪ Expression.And(
22:                                Expression.Eq("Operations.QoS.Property.Name", $name(q)$),
23:                                Expression.Ge("Operations.QoS.DoubleValue", $value(q)$)))
24:        end if
25:        if $exactQoS(q) = true$ then
26:            query ← query ∪ Expression.And(
27:                                Expression.Eq("Operations.QoS.Property.Name", $name(q)$),
28:                                Expression.Eq("Operations.QoS.Value", $value(q)$)))
29:        end if
30:    end for
31:    query ← query ∪ getInputExpression($I_c$) ∪ getOutputExpression($O_c$) ∪
32:                      getPrecondExpression($P_c$) ∪ getPostcondExpression($E_c$)
33:    result ← result ∪ executeQuery($query$)
34:    return result                                                 ▷A list of service candidates for $f_j$
35: end function
```

The categories and super-categories for a feature f_j are matched from lines 6–13. If a feature f_j belongs to a certain category, a query criterion is added to the $query$ variable. The helper function $cat(f_j)$ is used to get the name of the category for feature f_j. Besides that, a number of super-categories can be specified in VCL. For example, the category identifier

`Telco.Messaging.PortingServices` defines the category `PortingServices` and two super-categories, namely `Messaging` and the root category `Telco`. Therefore, the feature resolution has to add another criterion for every super-category that is different from the wildcard character `"*"`.

From lines 14–29, the required QoS attributes are pre-filtered. We distinguish three types of QoS attribute dimensions: *descending* (the lower the value the better – e.g., response time), *ascending* (the higher the value the better – e.g., availability) and *exact* (values have to match exactly), thus we have three different conditional statements to construct the query. The functions *name(q)* and *value(q)* are used to retrieve the name of a QoS attribute and its value respectively.

From lines 31–32, we simply invoke a set of helper functions for matching the input, outputs, pre- and postconditions. The criteria generated by these helper functions are similar to the other criteria, however, they try to match the required input and output data in our metadata model. However, a detailed discussion of the service matchmaking approach is out of scope of this thesis.

8.3 Generating Structured Compositions

Existing composition engines (such as WS-BPEL) are capable of enacting structured process models and they usually cannot deal with unstructured process models[1]. For generating a structured composition, we leverage the approach presented by Eshuis et al. [49]. According to their terminology, we refer to a composition that is based on a structured process model as *structured composition*. A structured model is desired and necessary for our approach based on the following reasons:

(i) it enables enactment of a structured composition on existing composition or workflow engines, thus, removing the need to implement an execution engine for VCL;

(ii) it allows to detect flaws in the unstructured composition such as deadlocks which may lead to runtime errors at a later stage;

(iii) it facilitates an efficient QoS aggregation based on well-known workflow and composition patterns [150]. QoS aggregation is needed during the optimization to determine an aggregation formula for a composition based on atomic aggregation formulas of the composition patterns [67].

In order to allow a fully-automated generation of a composite service, we significantly extend the work presented by Eshuis et al. because their original approach proposes a semi-automated technique. The user has to manually annotate the resulting structured composition, i.e., the branching types (conditional or parallel) need to be specified afterwards because they cannot be determined in advance.

[1]WS-BPEL has two exceptions: it allows cross links between parallel services and parallel blocks.

In the following, we substantiate the details of their original approach and describe the changes and additions to use the structured composition algorithm in a fully-automated way. In general the generation of a structured composition works as follows. Firstly, an abstract dependency graph (ADG) is generated to analyze the data flow of the business protocol as specified in VCL. The ADG can be generated simply by examining the input and output data concepts of each feature. Secondly, the structured composition is generated using the ADG as an input. Thirdly, the resulting structured composition is annotated with control flow decisions which are not present in the structured composition. These decisions are either AND (executed in parallel) or XOR (conditional execution).

8.3.1 Abstract Dependency Graph

Each feature in the VRESCo metadata model expects an input and an optional output message which is composed of well-defined data concepts. The ADG can be defined based on the input/output data concepts of each feature. If a feature f_2 expects an input that is the output of a feature f_1, then we say f_2 depends on f_1. All data dependencies are captured in a graph data structure by using an adjacency list. The functions $input(f)$ and $output(f)$ in the following definition are used to get the input and output data concept of a feature. An ADG is given by the tuple (F, E):

- A set of features $F = \{f_1, f_2, \ldots, f_n\}$
- $E = \{(f, f') \in F \times F \mid output(f) \cap input(f')) \neq \emptyset\}$

Additionally, Eshuis et al. impose two constraints on the ADG:

- C1: The dependency graph is acyclic.
- C2: If there is an edge from feature f_1 to f_2, then there is no path with length greater than 1 from f_1 to f_2.

Constraint C1 eliminates the use of loops in the ADG whereas C2 is needed to construct if-then-else compositions with an empty else branch. Fortunately, C2 is not very restrictive and an ADG can be simply repaired either by removing the violating dependency or putting an empty dummy feature between each violating pair of features.

In contrast to [49], we annotate the ADG with special nodes representing control flow information such as conditionals or loops that are present in VCL. One of the main reasons for annotating the ADG is to overcome the need to manually annotate the branching types in the resulting composition. This would not be possible in our case due to the fact that we leverage an automated approach to generate the composite service.

In Figure 8.2, the ADG for the cell phone number portability example from the preceding chapter is used. The VCL code for this example can be found in Listing B.1 in Appendix B. The ADG has a number of special nodes for annotating control flow information. At the beginning of the ADG, we add a ROOT node representing the composite service interface defined by the

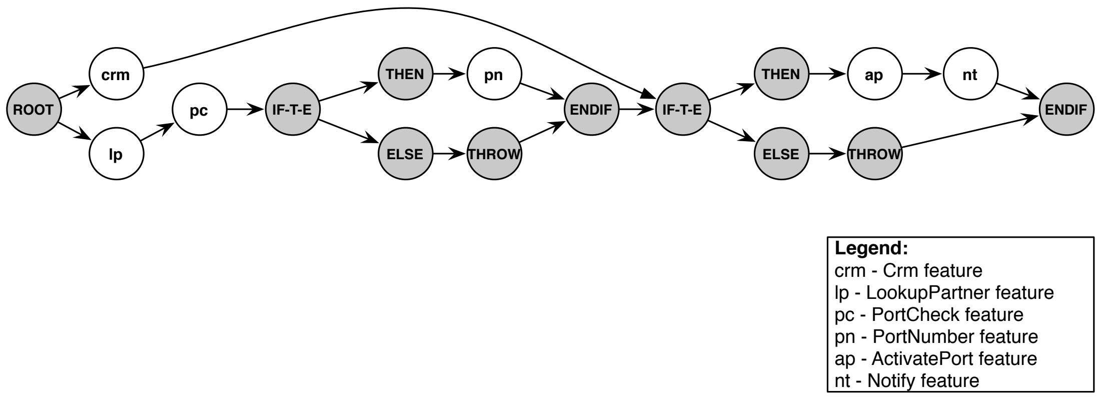


Figure 8.2: Annotated Abstract Dependency Graph

global input constraints in VCL. In the ADG, this node has no inputs and outputs all data concepts of the composite service to its descendants. In case of a conditional, the node IF-THEN-ELSE (abbrev. IF-T-E) with its immediate successors THEN and ELSE represents the two possible conditional branches. It is followed by an ENDIF node to annotate the end of a conditional. In case of loops, these special nodes can be used to emulate loops which were originally ruled out by constraint C1. By adding a "LOOP" start and end node, we do not need to add the dependency from the loop end to the beginning (thus not violating C1) However, this special loop node can be used to store the estimated loop count for the QoS aggregation and the composition generation component can successfully reconstruct the loop. The final special node is THROW to annotate an exception during the execution of a composite service.

8.3.2 Generating the Structured Composition

The generation of a structured composition is done based on the ADG generated in the previous step. Our approach builds upon the algorithm presented by Eshuis et al. [49], therefore, we briefly summarize its main idea. They introduce a simple structured composition language to illustrate their approach. It provides a hierarchical view, where a leaf node represents a service (a feature in our terminology) and a non-leaf node represents a block. Basically, the language considers two kinds of blocks: composite blocks (COMP) and sequential blocks (SEQ). COMP blocks are later annotated with either AND or XOR to reflect the branching type. The language uses the set notation in case of COMP and a list notation in case of SEQ to specify children of blocks. For example COMP{SEQ[A,B],SEQ[C]} specifies that A executes before B and both are executed in parallel with or exclusive to C.

The main idea of their algorithm is to find a set of initial features that do not depend on any other feature to construct a block of this initial set of features. A block is constructed by composing a set of features F into a single service (if F is a singleton), otherwise, a composite block consisting of a set of sequential blocks is constructed. In case of the ADG in Figure 8.2, the initial block is the ROOT node and is constructed as SEQ[ROOT].

The main part of the algorithm iteratively processes all features in F. However, the selection of features is not trivial, since these features cannot be processed one by one. They have to be processed as groups of maximal *influencing* subsets of features. For example, two features f_1 and f_2 influence each other if they directly influence each other (i.e., by depending on the same feature) or another feature f directly influences f_1 or f_2. A influencing subset is considered maximal, if adding another feature would result in a non-influencing set. Having identified these influencing subsets, they have to be analyzed and appended to the corresponding block based on various rules (e.g., whether the block is a sequence or composite). For details regarding these rules, see their algorithm [49].

In Figure 8.3, the graphical representation of the resulting structured composition is shown. The beginning and end of a block have a split and a join node (the node with C as label; C stands for composite). Additionally, the beginning and end of a composition also have a composite node because the whole composition is also treated as a block. Besides the graphical language, a full specification of the cell phone number portability process using the (textual) structured composition language can be found in Listing B.2 in Appendix B.

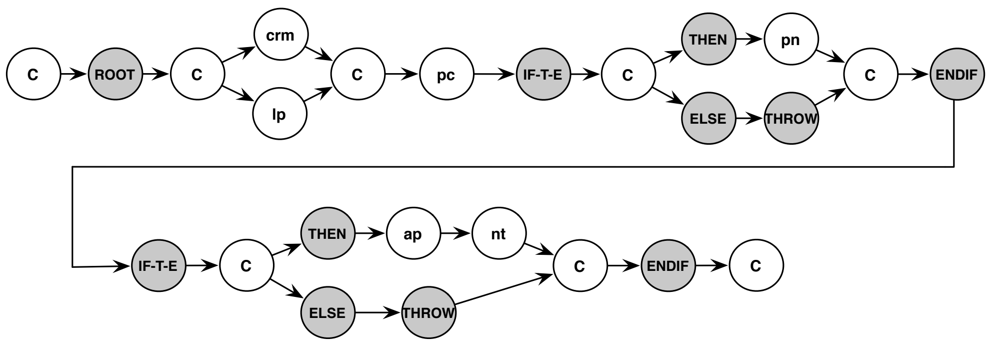


Figure 8.3: Structured Composition Graph

After having generated the structured composition, we annotate each C-block with either AND or XOR to specify the branching type. Our rule is simple: we annotate each block with AND except blocks where the parent node is a special IF-THEN-ELSE node. In addition to that, we associate all service candidates S_j from the feature resolving process with each feature f_j in the structured composition graph. This information provides the input for generating the QoS-aware optimization problem.

8.4 QoS-Aware Optimization

The goal of the QoS-aware optimization is an optimal selection of one service candidate $s_{ij} \in S_j$ for each feature f_j where all the required global and local constraints are fulfilled and a number of optional constraints from the constraint hierarchy H are satisfied. It is important to note that we do not search for the overall best solution in the search space, however, we search for the best solution within the QoS boundaries given by the user constraints. We present two different approaches for modeling the QoS-aware optimization problem, a constraint optimization problem (COP) and an integer programming problem (IP). The reason for devising an IP solution as an alternative is based on the fact that most constraint-based approaches have scalability problems when applied to medium and large-scale practical optimization problems [19]. However, the COP solution provides a simpler way of handling constraint hierarchies in terms of modeling the problem.

8.4.1 QoS Aggregation

QoS aggregation is an important means to calculate the QoS of the overall composition for each QoS attribute. In Table 3.1 and Table 3.2 in Chapter 3, we summarized all QoS attributes including their aggregation formulas for the composition constructs supported in our approach. These aggregation formulas are used during the optimization process to aggregate the value of a specific QoS attribute for the overall composition (e.g., when checking a global constraint). Therefore, we use a recursive QoS aggregation algorithm that traverses the structured composition from the previous step and applies the corresponding aggregation pattern based on the block in the composition, either a SEQ or a COMP node with AND or XOR. In case of a conditional (XOR), the value p_i in the aggregation formula is the probability that one specific path is chosen. In case of a loop, the value c represents the expected loop count. For security and reliable messaging we assume that all services in the composition have to support the same security protocol and reliable messaging, respectively, to enable it for the composite service. A more advanced mechanism will be considered in future work.

8.4.2 Constraint Optimization Problem

A COP is a constraint satisfaction problem (CSP) in which constraints are weighted and the goal is to find a solution maximizing a function of weighted constraints. A CSP is defined as a tuple $\langle X, D, C \rangle$ where X represents a set of variables, D_i represents the domain of each variable X_i and C represents a set of constraints over the variables X. A solution is an assignment of values from the variable's domain D to each variable $X_i \in X$ satisfying all the constraints C. For modeling our problem as a COP, we have to distinguish between *global constraints* and *feature constraints*. Both constraint types can have *required* and *optional* QoS constraints. Each required constraint has to be fulfilled, otherwise no solution can be found. All optional constraints (global and local) will be added to the objective function that has to be maximized. As aforementioned, all required feature constraints have already been pre-filtered, therefore, it is ensured that only service candidates have to be considered that fulfill all required feature constraints. This may reduce the number of constraints in the problem space.

8.4.2.1 Feature Constraints

Modeling feature constraints requires to add all service candidates S_j for each feature f_j as variables to the problem space. As we only want to select one service candidate from all available services S_j to execute a feature, we have to add the following *selection constraint* given that y_{ij} denotes the selection of a service candidate s_{ij} to execute a feature f_j (y_{ij} is modeled as a boolean decision variable[2]):

$$\sum_{i \in S_j} y_{ij} = 1, \forall j \in F \tag{8.2}$$

Each feature f_j can be subject to feature constraints, therefore, we need to add the following constraint for each feature constraint Q_{fc} to determine the selected QoS value q_{jk} of a feature (local selection). q_{jk} represents the selected QoS value for a given feature f_j.

$$q_{jk} = \sum_{i \in S_j} Q_{ik} * y_{ij}, \forall k \in Q_{fc} \tag{8.3}$$

Depending on the QoS attribute dimension (ascending, descending, exact) we need to add the corresponding constraints for each QoS attribute to capture whether an optional QoS constraint c_{jk} is satisfied (c_{jk} is represented as a boolean decision variable). The value q_{jk} is the value from constraint (8.3). For descending dimension, $c_{jk} = (q_{jk} \leq Q_{fc_k})$ is added, for ascending dimension $c_{jk} = (q_{jk} \geq Q_{fc_k})$, and for exact dimension the resulting constraint is $c_{jk} = (q_{jk} = Q_{fc_k})$.

Additionally, we use the following function (8.4) to map the constraint hierarchy levels to strength values that is then used in the objective function. Please note that these values are flexible and can be changed to reflect a different mapping (e.g., give more weight to *strong* constraints).

$$strength(c) = \begin{cases} 20 & if\ c \in H_1 \\ 10 & if\ c \in H_2 \\ 5 & if\ c \in H_3 \\ 0 & otherwise \end{cases} \tag{8.4}$$

All the aforementioned constraints describe the selection of an optional feature QoS value. These constraints are added for each feature f_j and maximized as part of the objective function:

$$\max \sum_{j \in F} \sum_{k \in Q_{fc}} c_{jk} * strength(Q_{fc_k}) \tag{8.5}$$

8.4.2.2 Global Constraints

In order to add global constraints (required or optional ones) to the problem space, we first need to create a global aggregation formula depending on the structured composition as

[2]When using a boolean decision variable, 0 is used for false and 1 for true.

shown previously in Figure 8.3 and the aggregation formulas are shown in Table 3.2. As describe above, we use a recursive algorithm to traverse the structured composition from the previous step and generate a global aggregation formula for each feature f_j. For example, when aggregating the response time q_{rt} for the first composite block from Figure 8.3 (containing the feature `crm` and `lp`), the following aggregation constraint applies (k is the index for the QoS constraint, in this example the index would be 3 for the response time):

$$a_k = \max_{j \in \{crm, lp\}} \{q_{jk}\} \tag{8.6}$$

In the following, we use a_k to represent the aggregation constraint of the k-th QoS attribute which is added for every global constraint that is specified by the user in the VCL specification. In case the global QoS constraint is required, we add another constraint depending on the QoS attribute dimension. For QoS with descending dimension, $a_k \leq Q_{gc_k}$ is added, for ascending dimension $a_k \geq Q_{gc_k}$, and for exact dimension the resulting constraint is $a_k = Q_{gc_k}$, where k is the QoS attribute index.

In case the global QoS constraint is optional, we have to add a decision constraint to check whether an optional constraint has been fulfilled. Again depending on the QoS attribute dimension, we add the following constraints: For descending dimension, $c_k = (a_k \leq Q_{gc_k})$ is added, for ascending dimension $c_k = (a_k \geq Q_{gc_k})$, and for exact dimension the resulting constraint is $c_k = (a_k = Q_{gc_k})$. Finally, we have to add these decision constraints multiplied with their strength value to the objective function from (8.5) to get the overall objective function:

$$\max \left(\sum_{j \in F} \sum_{k \in Q_{fc}} c_{jk} * strength(Q_{fc_k}) + \sum_{k \in Q_{gc}} c_k * strength(Q_{gc_k}) \right) \tag{8.7}$$

The objective function (8.7) is then maximized by the solver to find an optimal solution within the constraint boundaries set by the user in the VCL description. All the values in our COP are scaled to integers by multiplying them with 100. Due to the fact that we only allow two decimal places in VCL we do not have any precision loss.

8.4.3 Integer Programming Approach

Integer programming optimizes a linear objective function that is subject to linear equality and linear inequality constraints. Compared to the CSP approach, there are a few changes that are needed when modeling the QoS-aware composition problem as IP. We have to define a new objective function calculating an overall utility value for each feature f_j considering the user's QoS constraints and their strength. Additionally, we need to linearize the aggregation rules for q_{ac} and q_{av} because they use the product to aggregate the QoS for a sequence and parallel execution of features.

8.4.3.1 Feature Constraints

Feature constraints are handled by using a utility function that is calculated for each service candidate. The selection constraint (8.2) is still valid in the IP formulation. For calculating the QoS utility function for each service, we first need to scale all the QoS values to a uniform representation. Contrary to other approaches in this area [176], we do not use simple-additive weighting to scale the values, however, we scale all values between [0, 100] depending on the percentage to which a QoS attribute of a service candidate fulfills the optional constraint imposed by the user. For example if the user specifies an optional availability constraint on a feature f_j with the value 0.95 and the QoS value of the service candidate is 0.99, we set the value scaled value to 100 because the optional feature constraint is 100 percent satisfied (in fact is over-satisfied). The overall objective function is shown in (8.8):

$$max\left(\sum_{j\in F}\sum_{i\in S_j} y_{ij} * \sum_{k\in Q_{fc}} scale(Q_{ik}, Q_{fc_k}) * strength(Q_{fc_k})\right) \tag{8.8}$$

The function $scale$ scales the k-th QoS value Q_{ik} of a service candidate s_i between [0, 100] depending on the actual QoS feature constraint value Q_{fc_k} specified by the user in VCL and the QoS dimension (ascending, descending, exact). For modeling the constraint strength, we still use function (8.4).

8.4.3.2 Global Constraints

For adding the global constraints, we follow a similar approach as in the CSP solution. We first aggregate the QoS attributes using a similar function as in the CSP approach, with the exception that we linearize the product aggregation rules using the ln (as shown in [176]). Whenever a global QoS constraint is required, we add a linear equality or inequality to the problem space. If a global constraint is optional, we add it to the overall objective function that has to be maximized.

8.5 Generation and Deployment of the Composite Service

After the optimization phase, one service candidate has been assigned to each feature in the composition. This information is stored in our internal graph data structure that holds the structured composition, its service candidates and the selected "best" candidates for each feature. Based on this internal data structure, the *Composition Service* transforms this structured composition into an executable composition. In addition, we also dynamically generate the composite service configuration files and compile the generated code into a .NET DLL (Dynamic Link Library). This DLL then gets deployed on the Microsoft IIS (Internet Information Service), a Web server on the Microsoft platform that is capable of hosting .NET Web services.

Code Generation. As mentioned earlier, we leverage the Windows Workflow Foundation (WWF) [96] as the underlying composition engine, therefore, we have to generate code specifically for this platform. The available workflow constructs of WWF are somehow similar to BPEL, however, WWF supports the use of code activities to call some piece of .NET code. In general, there are two options for authoring or generating WWF code: The first one is to generate pure .NET source code containing the workflow by deriving from a base class called `SequentialWorkflowActivity`. The second option it to generate the code in XAML (EXtensible Application Markup Language), an XML dialect for declarative specification of .NET applications. Unfortunately, WWF in its current version (v3.5) does not fully support a declarative generation of workflows in XAML for some technical reasons. Therefore, we had to combine XAML with source code, so-called code-beside. This source code is used to specify properties which are used to hold data of a receive activity or the reply. Additionally, the code-beside contains all code activities that are then referenced from the XAML file.

SC Node	WWF Activity	Semantics
ROOT	`ReceiveActivity`	Entry point in composition
SEQ	`SequenceActivity`	
AND	`ParallelActivity` and `SequenceActivity` for each branch	
XOR	`IfElseActivity` and `IfElse-BranchActivity` for each branch	The if-then-else conditions are specified as `CodeCondition` referencing a method in the code-beside file.
LOOP	`WhileActivity`	Condition is specified using a `CodeCondition` referencing a method in the code-beside file.
THROW	`CodeActivity`	Code activity implements the exception handling by wrapping it into a SOAP fault message.
Complex data types	No WWF activity, but a C# class for each type	The input and output of a composition can contain complex types.
Service invocation	`CodeActivity`	The code activity uses DAIOS to invoke a service by using a `DaiosProxy`.

Table 8.1: WWF Generation

In order to transform the structured composition to WWF activities, the algorithm traverses every node in the graph and depending on the type, it generates the corresponding WWF activities. We do not present the algorithm because it is relatively straightforward. However, we show the mapping from the structured composition to WWF and explain further aspects that need to be addressed in Table 8.1.

The generation of the WWF file results in two files, a XAML and a C# source code file containing all the properties (e.g., variables for handling received data and data to return to the caller) and all the conditions from the if-then-else and while activities. After the generation, both files, the C# and the XAML file and the complex data type classes (generated using the *Reflection.Emit* classes from the .NET framework) are compiled by using the `WorkflowCompiler` class provided by the WWF framework. This results in a DLL that can then be deployed to the IIS.

Composite Service Deployment. The final deployment step is simple compared to the other steps. As a prerequisite, the IIS service needs to be installed and configured to be able to host .NET applications. In order to deploy the service, the DLL needs to be copied into a so-called virtual directory which has to be configured once in the IIS to know where to check for application code. Besides copying the DLL, it also requires a special configuration file telling the IIS the name of the composite service and the name of the DLL that holds the code for executing the workflow. Since this configuration file only has two parameters (the name of the composite service and the location of the DLL), the composite service generation component has a pre-defined template that is populated with the corresponding values and also copies it to the target location. Finally, the composite service is ready to serve client requests. Additionally, the endpoint of the generated service is sent back to the caller of the *Composition Service* and is also stored in VRESCo.

8.6 Implementation and Evaluation

The CaaS approach was implemented in the *Composition Service* as part of the VRESCo runtime environment implemented by using .NET/C# and the Windows Communication Foundation (WCF) for realizing the Web service communication. For querying the services as part of the feature resolution we have used the VRESCo Query Language (VQL) to retrieve all deployed service instances. The optimization algorithms are implemented by using the Microsoft Solver Foundation [97], a recently released optimization library supporting CSP, LP, LIMP and Quadratic programming. For executing the generated composition we use the Microsoft Windows Workflow Foundation [96]. The overall implementation of the *Composition Service* consists of approximately 10000 lines of C# code (without the VRESCo code itself).

In order to demonstrate the effectiveness of our approach, we measure the performance of various crucial components such as the querying engine and the optimization approach. Therefore, we have written a tool to deploy features, categories, services and QoS values in VRESCo. Additionally, it generates VCL files with a given number of features and service candidates per feature. All the performance tests were executed on an Intel Xeon Dual CPU

X5450 3.0 GHz with 32 GB of memory (although memory is not an issue in our tests).

8.6.1 Feature Resolution

The first important aspect is the performance of the feature resolution step. A query encodes all feature constraints (except optional QoS) to find concrete service candidates in the VRESCo registry as described earlier this chapter. This step involves one query per feature in the composition using our VQL query language that are translated to SQL queries at runtime. We use a parallel query execution to speed up the performance as depicted in Figure 8.4 (the average of 10 repetitive runs). On the x-axis, we represent the number of features in the composition (from 5–100). The y-axis shows the query time depending on the number of candidates (10, 25, 50, 75, 100). For example, in a composition using 100 features, and each feature is implemented by 10 candidates, the overall query time is less than a second (734 msec) and grows to more than 4 sec for 100 service candidates. Fortunately, the number of candidates is usually low (approx. 1–5), therefore, resulting in a very good overall query performance.

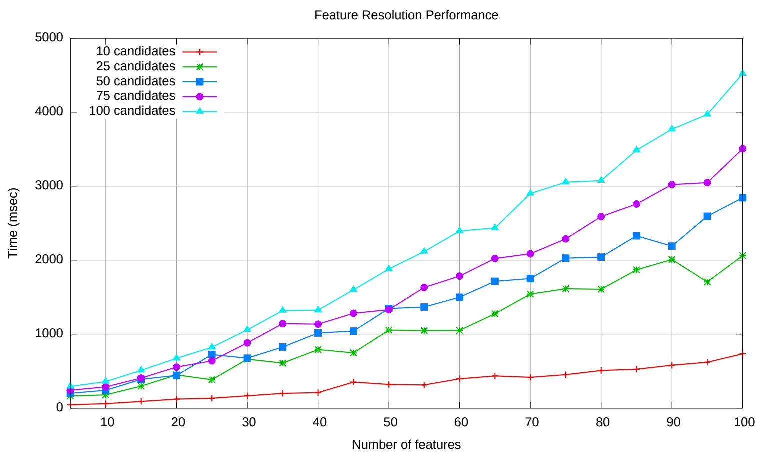


Figure 8.4: Feature Resolution Performance

The main reason for having a higher query performance when the number of features increases is the fact that VQL has to create a large number of C# objects for each query. For example, a query to retrieve 100 service candidates, returns 100 so-called `ServiceRevision` instances where each object has a number of other objects associated with it (e.g., operations, QoS, feature, etc.).

8.6.2 Structured Composition Generation

The generation of a structured composition from the original VCL input follows the approach proposed by Eshuis et al. and is adapted where needed. We have used a simple sequence of features from 5–100 in steps of five on the x-axis. The y-axis shows the time that is needed to generate the structured composition.

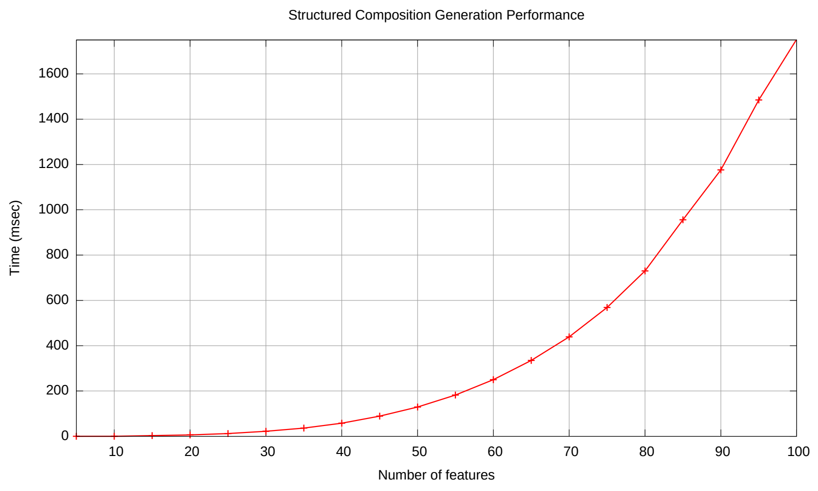


Figure 8.5: Structured Composition Generation Performance

Please note that the generation of the structured composition is not depending on the number of service candidates per features. The graph shows that the performance penalty compared to the other parts is low, typically less than a second for various compositions with different size and structure.

8.6.3 QoS-Aware Optimization

For measuring the performance of the COP-based solver, we have generated small datasets as we expected a slow performance based on the complexity of the COP in general (NP-hard), and our problem in particular. In Table 8.2 the performance of the COP approach is depicted. The first column shows the number of features (from 5–25). Columns 2–4 show the performance in msec based on the number of candidates per feature (3, 5, 7). Each VCL file used for measuring the performance contains two global constraints and one local constraint per feature.

The results clearly show the bad performance (and negatively exceeded our expectations) as soon as the number of features is greater than 15 (and 5 candidates per feature). The value "exceeded" expresses that our timeout value of 120000 msec was exceeded. This makes the solver impractical for large QoS-aware optimization problems, however, for small scale problems it is still usable. Moreover, the current solver allows to save the internal solver state so that it can be re-used and constraints can be added/changed or removed and then resolved. This will reduce the optimization time when the user specified constraints do not hold and the user has to relax some constraints.

Features	3 cand./feature	5 cand./feature	7 cand./feature
	time in msec		
5	16	23	28
10	22	77	203
15	782	106952	exceeded
20	58789	exceeded	exceeded
25	exceeded	exceeded	exceeded

Table 8.2: COP Performance

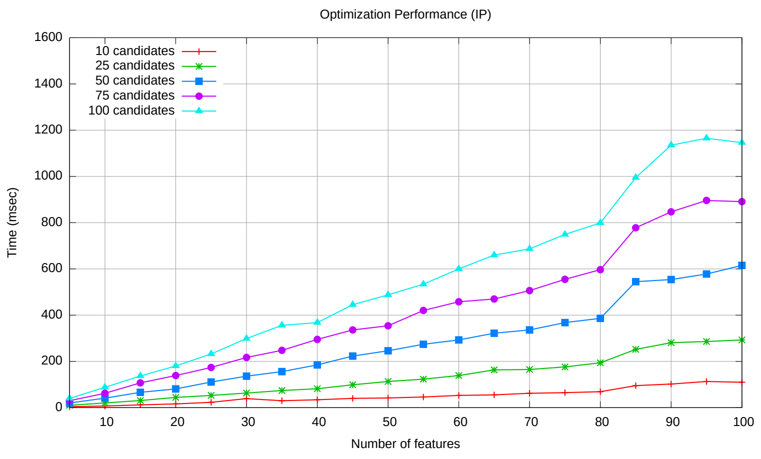


Figure 8.6: Optimization Performance (IP)

For measuring the performance of the IP solver, we have generated a number of VCL compositions (sequential and parallel constructs) with a different number of features (from 5 to 100) and with increasing number of service candidates per feature (5, 25, 50, 75, 100). Each of the VCL files contains 4 global constraints and each feature has 4 feature constraints. We do not use any pre-filtering as part of the feature resolution in order to have the exact number of all the candidates per feature as given above. The performance results are depicted in Figure 8.6 (average of 10 repetitive runs). The x-axis shows the number of features, the y-axis shows the time in msec. Each function in the plot displays the runtime in msec based on the number of candidates per feature on the x-axis. The performance shows good results even for medium-size compositions (100 features), where a solution for 10 service candidates can be found in 110 msec and for 100 service candidates in 1145 msec.

8.6.4 Composite Service Generation and Deployment

The generation of the composite service code requires almost constant time. It includes the generation the core workflow in XAML, the code-beside file and all the complex types for every feature. Additionally, we need to compile all the complex types and the code-beside which adds an additional overhead to this task.

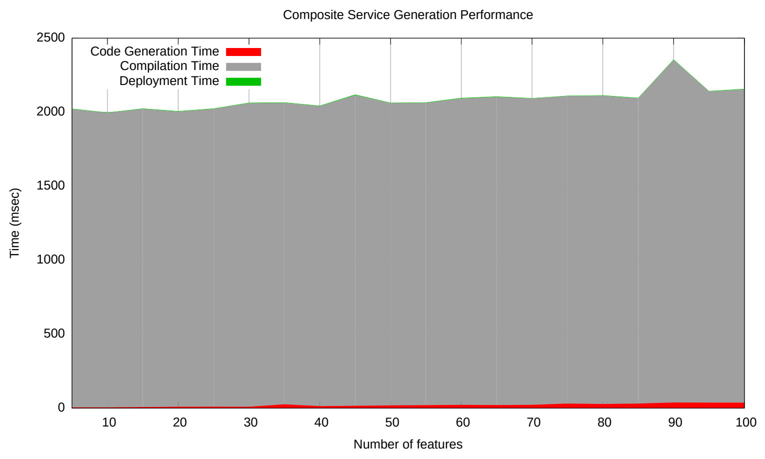


Figure 8.7: Composite Service Generation Performance

In Figure 8.7, the performance of the composite service code generation is depicted. Again, we use the number of features on the x-axis and the time on the y-axis. Basically, this process shows three parts. The first part and least time consuming part is the code generation which consists of traversing the structured composition and generating a corresponding WWF activity, generating the code-beside file and all C# classes for all complex types. In this example, each of the features in the original VCL file has a complex type as input and output, therefore, requiring 300 classes in case of 100 features (200 classes for the input and output and 100 for another complex type referenced by the input type). The second part is the compilation of the previously generated code. It is considered constant with an upper bound of approx. 2000 msec. The time for the deployment of the compiled composite service is not significant and constantly about 3–5 msec.

8.6.5 End-to-End Performance

After analyzing each component in detail, we finally present the end-to-end performance to compare all aspects involved to generate a composite service. In Figure 8.8, the performance of the VRESCo *Composition Service* is shown for an increasing number of features on the x-axis and 10 service candidates for each feature.

The performance of the different components is shown in different colors. Clearly, the feature resolution (in blue) requires most of the time, followed by the code generation and deployment, and the generation of the structured composition. The other aspects are not significant in terms of their overall performance. The overall time needed to dynamically create and deploy a composition is reasonable and making this approach a good candidate for runtime composition and re-composition.

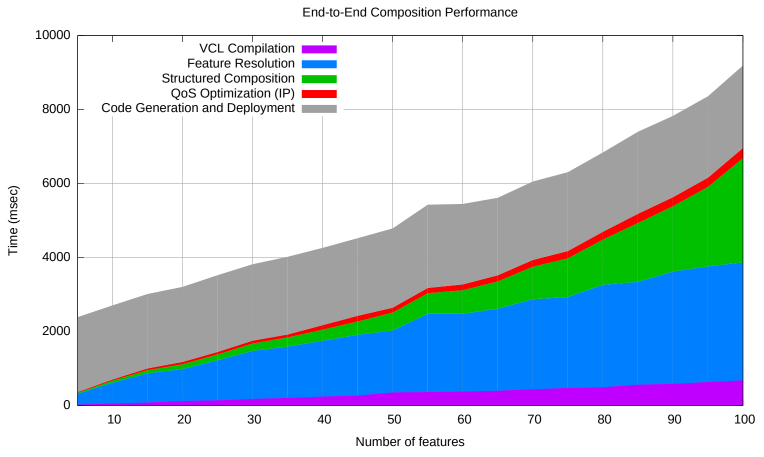


Figure 8.8: End-to-End Performance for 10 Service Candidates

8.7 Discussion

The presented QoS-aware composition approach is based on the idea that compositions are hosted and can, therefore, be outsourced to realize Composition as a Service (CAAS). The approach is integrated into the VRESCO runtime environment and combines a number of novel concepts such as support for hard and soft QoS constraints and the automated generation of an executable composite service without the need for setting up and maintaining a composition infrastructure.

Currently, one of the main application areas for the CAAS approach is to specify QoS-aware composite services on a microflow level. Compared to existing macroflow-based orchestration languages such as BPEL, these microflow services usually define a simple business protocol and are mainly used to define short running processes that do not have multi-party interactions. An efficient development and provisioning of QoS-aware microflows is important to implement larger, more coarse-grained business processes (macroflows). Therefore, the CAAS approach addresses adaptive behavior in a bottom-up manner by starting at the technical level of a business process implementation.

In particular, CAAS supports the development of adaptive behavior in two ways: Firstly, it leverages VRESCO as the implementation platform, therefore, we gain adaptiveness by using its dynamic binding and re-binding capabilities. In addition, by building upon VRESCO's feature-driven programming model, we additionally support adaptiveness by allowing to add new services or exchange existing ones transparently for higher-level macroflows (including runtime mediation). Secondly, adaptiveness is actively supported by the presented approach based on the fact that the QoS optimization can be re-executed for example by sending a request to the composite service or re-optimizing in an offline-mode. This ensures that the dynamically generated composite services always use the "best" service candidates that match all user constraints as initially specified in VCL.

Chapter 9

Conclusions and Future Research

Contents

9.1 Summary

This thesis has addressed several aspects related to the development of adaptive service-oriented systems by leveraging QoS as an important means to assess the quality of a service and to provide a main decision criteria when and what to adapt. It is important to understand that adaptivity is not simply a means that can be addressed by adding proprietary support for it to specific applications. However, we showed that adaptivity requires some generic concepts and mechanism that need to be integrated into the design process and in the middleware layer to provide a generic solution. In this thesis, we followed this approach by providing general concepts and tools that enable native support for adaptivity (such as the VRESCo environment) and allow developers to implement more advanced concepts by providing custom adaptation logic (e.g., by extending VRESCo's rebinding logic).

The first part of this thesis has dealt with all issues related to the integration of QoS into various layers of a service-oriented system. In particular, we have described an extensible multi-layer QoS model to address QoS requirements and guarantees on various layers such as choreography, orchestration and service layer. This model provides different views on QoS, in particular related to the development of service-oriented applications. Secondly, a QoS monitoring method has been described to bootstrap and constantly monitor QoS attributes. We have presented and evaluated a black-box QoS monitoring technique that provides the basis for monitoring non-deterministic QoS attributes. Thirdly, we have contributed an approach which addresses QoS issues already at the highest-level during the development of service-oriented systems, namely on the choreography layer. We leverage QoS in form of SLAs on the choreography layer and automatically transform the choreography into orchestrations for each partner including their enforceable QoS policies that reflect the SLA from the choreography layer.

The second part of this thesis has focused specifically on the QoS-aware composition and its execution. To this end, we have presented a novel Web service runtime called VRESCo which proposes a feature-driven programming model for implementing adaptive service-oriented applications. VRESCo addresses various research challenges in SOC such as dynamic binding and invocation, QoS-aware composition, and efficient service selection just to name a few. Based on the VRESCo environment, we have presented a novel QoS-aware composition approach by proposing a domain-specific language called VCL (Vienna Composition Language). This language puts a particular focus on a constraint-based specification of QoS requirements. As opposed to existing approaches, QoS can be specified globally for a composition or locally for each feature. Each QoS constraint can either be a hard or a soft constraint using constraint hierarchies to specify the relative importance of a constraint. This language is the input for the "Composition as a Service" (CaaS) approach which is proposed as final contribution of this thesis. It enables composition as a service, thus reducing the need to provide its own composition infrastructure. It uses VCL as the language and VRESCo as a backend to dynamically generate and deploy a composition based on the input specified by the user.

9.2 Assessment of the Research Questions

In this section, we assess the proposed methods and systems in this thesis based on our research questions from Chapter 1.

Research Question Q1. In order to support the full services lifecycle in a service-oriented system, we proposed a multi-layer QoS model to coherently integrate QoS information in one common model at various levels of abstraction (choreography, orchestration and service layer). The resulting multi-layer model explicitly supports different lifecycle phases during the development of service-oriented systems. One example is the development of Web service based business processes (macroflows) in a top-down manner or an efficient development of composite services (microflows) that implement parts of the macroflow process activities.

The modeling aspect is specifically addressed on the choreography layer by supporting the definition of SLAs between the stakeholders in a system or business process. On the orchestration layer, QoS policies are specified by using WS-QoSPolicy to enable monitoring and enforcement of these policies during runtime. The service layer defines the common "vocabulary" by proposing some well-defined and measurable QoS attributes categorized in four different groups (Performance, Dependability, Security and Trust, Cost and Payment).

In order to facilitate the development of Web service based business processes, the method proposed in Chapter 5 is an effective solution to include SLA and QoS aspects from the beginning of the modeling phase. This is achieved by defining SLA-aware choreographies and automatically derive orchestrations for each partner. During the transformation process, the QoS policies for the partner orchestrations are automatically generated to describe the required QoS according to the SLA as specified during the modeling phase. These orchestrations can be used as a starting point for the implementation of the "internal" business logic that is then deployed to the VieDAME enhanced BPEL engine. This automatic transformation method reduces the development complexity of long running business processes and avoids a manual specification of QoS policies. The VieDAME runtime enables a higher degree of adaptivity and SLA compliance within a business process because services that do not fulfill the SLA can be detected and replaced if necessary. It avoids SLA violations and the resulting penalties and provides a better QoS experience for the end-user of the process or the consumer application in terms of fault-tolerance and reliability. Additionally, the tradeoff between the degree of adaptivity gained through the use of VieDAME and their performance penalties is negligible (in detail reported in [99]).

A foundational and decisive factor for a successful development and adaptation of QoS-aware compositions is the availability of accurate QoS information. The QUATSCH monitoring approach presented in Chapter 4 provides the necessary mechanisms to bootstrap and monitor service layer QoS attributes. The approach to combine probing with low-level TCP sniffing has proven to be effective. The accuracy of the monitored values was demonstrated in the evaluation of our solution. We have shown that the monitored attributes, such as the service-side execution time, can be measured accurately from a client-side perceptive (e.g., with less than five percent deviation from the "real" execution time of a service). Being able to measure the QoS attributes allows to build a repository of historical measurements for detecting different patterns in the QoS data. This knowledge can be used to improve the accuracy of the QoS attribute calculation. We support such an analysis by providing a Web-based tool preparing the data for the user by providing dynamic chart generation of different QoS attributes over time or between specific periods of time.

Research Question Q2. A QoS-based composition language for developing microflow activities (Chapter 7) has proven to be effective for the implementation of QoS-aware composite services. The language enables developers to specify their functional and QoS requirements for composite services by using a textual DSL. The decision to use constraint hierarchies to support hard and soft constraints provides an effective way to address the problem of over-constrained systems. The composition process benefits from the use of soft constraints because they allow a more flexible specification of QoS requirements. Typically, QoS constraints are not always considered mandatory, however, existing approaches such as [9, 176] do not take this into account. If a QoS value is below a given user threshold, these aforementioned approaches fail to deliver a result even if it is only partially fulfilled or QoS is only declared as "nice to have".

By leveraging a feature-driven metadata model, the specification of a composite service in VCL is more flexible compared to existing approaches. The composer only specifies the features that are needed in the composition as requirements (in form of the expected input and output data concepts and optional pre- and postconditions). The task of finding and matching these features from all available features and their services is then performed by the composition runtime. We have shown that this process, the so-called feature resolution, is efficient even if the number of features in a composition grows (e.g., 100 features). In case a feature cannot be found or the data concepts are not compatible with the ones provided in the repository, the system informs the developer to relax or change some constraints.

Providing the composition approach as a service (Chapter 8) is an effective solution to reduce the need to build and setup the overall composition infrastructure and supporting runtime. This principle is aligned with recent approaches in the area of Software as a Service (SaaS), therefore, providing the possibility to completely outsource this approach into increasingly popular cloud computing infrastructures such as Amazon EC2. This approach and runtime encapsulate all required steps to deliver an executable composition from the VCL input. We showed that such a generation involves a number of steps that can be performed efficiently for medium-sized compositions (e.g., 50 features). The use of IP as a means to implement the QoS-optimization has proven to be a better approximation in terms of performance compared to a pure constraint-based approach.

The proposed CAAS approach requires a powerful and seamless support from a Web service runtime to enable the well-known operations from the SOA triangle, such as publish, find and bind. The VRESCO runtime environment provides the necessary foundations in a coherent framework. The essential components for the CAAS approach are the feature-driven programming model including the mapping framework VMF, the querying language VQL, and the dynamic invocation and mediation. The feature-driven programming model allows a seamless classification of services and abstraction from low-level differences between similar services. On the one hand, it greatly improves the flexibility of applications and allows a transparent adaptation of services in a composition but on the other hand it requires the specification of a mapping between a concrete feature and a Web service operation. This is a well-known tradeoff that is required to increase adaptivity. The specification of a mapping also requires capabilities to mediate such requests at runtime. The proposed dynamic invocation and mediation approach has proven to have little overhead compared to unmediated requests and is independent of the number of required mediation steps (which also eases the mapping). Additionally, VQL also shows good performance characteristics. The power of VQL is its ability to query all aspects of our metadata and service model, thus it justifies the little tradeoff in performance compared to existing query languages such as SQL or HQL.

9.3 Outlook and Future Research

The research addressed in this thesis is by no means complete and leaves enough room for further research. In the following, we provide an outlook on several aspects that remain open as future work. We plan to address some of these issues in our ongoing research work.

Combined QoS Monitoring. QoS monitoring, as discussed in the first part of this work, is a fruitful area which is not sufficiently solved with regards to the accuracy, overhead and scalability that is required by the monitoring component. In our approach, the overhead is significant in terms of the number of monitoring requests that need to be sent in order to "probe" the service to calculate QoS statistics. In scenarios where real-time QoS information is required, scalability is a major issue. A client-side approach would not be the optimal solution because it can only provide near real-time QoS information depending on the monitoring interval. In such cases a combination with a service-side monitoring is inevitable, however, this is much more intrusive than a pure client-side solution. It will be left to future work to devise flexible server-side monitoring capabilities which emit trusted QoS information that can be consumed by a Web service runtime such as VRESCo. A successful monitoring will always be a tradeoff between ease-of-use and general applicability versus scalability and real-time QoS information.

Semi-Automated Service Composition. Currently, our QoS-aware composition approach proposed in Chapter 8 focuses on the microflow level for specifying the compositions. It requires that the user specifies the input and output of the composed service and the business protocol. However, the VCL specification combined with the information stored in the VRESCo registry theoretically provides enough information to perform semi- or fully-automated composition without requiring the user to specify the business protocol but only the services and the expected QoS. However, this would require a rigorous specification of pre- and postconditions for each service as part of the publishing process of each service in the VRESCo registry. Moreover, it would require a considerable effort to extend the current service composition algorithms to leverage and reason on the information provided as pre- and postconditions. The semantic aspects to define how pre- and postconditions will be interpreted are a crucial aspect for the applicability of such approaches. It is arguable whether it is easier for the user to rigorously specify pre- and postconditions to enable an automated composition approach or simple require the user to specify the business protocol for a microflow (which is typically manageable in terms of complexity).

Dynamic Re-composition. Currently, our proposed dynamic QoS-aware composition approach has a sufficient performance for typical composition scenarios to be executed just before a composition is executed (which results in a minor but acceptable overhead). However, in large-scale scenarios such an approach is not feasible to re-query all the service candidates and run the optimization algorithms. Therefore, we plan to leverage VRESCo's eventing infrastructure to subscribe to service changes. This will result in an event whenever the QoS of a service in a composition changes or other services are published which provide a better QoS. These events will then trigger a re-composition by re-running only the QoS optimization. This will allow all further instances to bind to new services that provide a better QoS.

Service Matchmaking. In VRESCO, we currently use data concepts as part of our metadata model as a means to define entities and data. These concepts are then mapped to concrete input and output data of a service. Currently, we only use a naive matching of data concepts when querying for services (e.g., during the feature resolution as part of the QoS-aware composition) by performing a simple name and type comparison. However, it would be possible to do a complex service matchmaking to leverage the power of the data concepts provided in VRESCO. We plan to add extended service matchmaking capabilities as part of our future work.

VCL Feasibility Study. Another important aspect with respect to the second part of this thesis is the feasibility of VCL. We have designed and implemented VCL as an approach to specify QoS-aware compositions. However, it remains an open question whether VCL is capable of solving this problem in an efficient and effective way from an end-user perspective. Therefore, we envision to setup a small group of individual experts with different skill levels to perform several experiments in form of small exercises to asses their experience and problems they had during these exercises. Definitely, these results will provide some valuable feedback for improving VCL.

Bibliography

[1] Active Endpoints. ActiveBPEL Engine [online]. 2007. Available from: `http://www.active-endpoints.com/` [cited March 15, 2009].

[2] E. Al-Masri and Q. H. Mahmoud. QoS-based Discovery and Ranking of Web Services. In *Proceedings of 16th International Conference on Computer Communications and Networks (ICCCN'07), Honolulu, Hawaii, USA*, pages 529–534, Aug. 2007. `doi:10.1109/ICCCN.2007.4317873`.

[3] E. Al-Masri and Q. H. Mahmoud. Investigating Web Services on the World Wide Web. In *Proceeding of the 17th International Conference on World Wide Web (WWW'08), Beijing, China*, pages 795–804. ACM Press, 2008. `doi:10.1145/1367497.1367605`.

[4] M. Alrifai and T. Risse. Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In *Proceedings of the 18th International World Wide Web Conference (WWW'09), Madrid, Spain*, pages 881–890. ACM Press, Apr. 2009. `doi:10.1145/1526709.1526828`.

[5] G. Alsonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications*. Springer, 2004.

[6] Apache Software Foundation. Apache CXF: An Open Source Service Framework [online]. Available from: `http://cxf.apache.org/` [cited March 15, 2009].

[7] Apache Software Foundation. Axis [online]. Available from: `http://ws.apache.org/axis/` [cited March 20, 2009].

[8] D. Ardagna and B. Pernici. Dynamic Web Service Composition with QoS Constraints. *International Journal of Business Process Integration and Management (IJBPIM)*, 1(4):233–243, 2006. `doi:10.1504/IJBPIM`.

[9] D. Ardagna and B. Pernici. Adaptive Service Composition in Flexible Processes. *IEEE Transactions on Software Engineering*, 33(5):369–384, 2007. `doi:10.1109/TSE.2007.1011`.

[10] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions Dependable Secure Computing*, 1(1):11–33, 2004. `doi:10.1109/TDSC.2004.2`.

[11] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-Time Monitoring of Instances and Classes of Web Service Compositions. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, pages 63–71. IEEE Computer Society, 2006. doi:10.1109/ICWS.2006.113.

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

[13] L. Baresi, S. Guinea, and M. Plebani. Business Process Monitoring for Dependability. In *Proceedings of the Workshops on Software Architectures for Dependable Systems (WADS'06), Lecture Notes in Computer Science 4615*, pages 337–361. Springer, 2006. doi:10.1007/978-3-540-74035-3_15.

[14] L. Baresi, S. Guinea, and P. Plebani. Policies and Aspects for the Supervision of BPEL Processes. In *Proceedings of the 19th International Conference on Advanced Information Systems Engineering (CAiSE'07), Trondheim, Norway*, pages 340–354. Springer, 2007. doi:10.1007/978-3-540-72988-4_24.

[15] A. Barros, M. Dumas, and P. Oaks. A Critical Overview of the Web Services Choreography Description Language (WS-CDL). *BPTrends Newsletter*, 3(3), Mar. 2005.

[16] A. P. Barros, M. Dumas, and P. Oaks. Standards for web service choreography and orchestration: Status and perspectives. In *Proceedings of the Business Process Management Workshops (Revised Selected Papers), Nancy, France*, pages 61–74, 2005. doi:10.1007/11678564_7.

[17] R. Barták. Online guide to constraint programming, 1998. Available from: http://kti.ms.mff.cuni.cz/~bartak/constraints/ [cited April 6, 2009].

[18] L. Bass and R. K. Paul Clements. *Software Architecture in Practice*. Addison-Wesley Professional, 1 edition, 1997.

[19] K. M. Bayer, M. Michalowski, B. Y. Choueiry, and C. A. Knoblock. Reformulating Constraint Satisfaction Problems to Improve Scalability. In *Proceedings of the 7th Symposium on Abstraction, Reformulation and Approximation, Whistler, BC, Canada*, pages 64–79. Springer, 2007. doi:10.1007/978-3-540-73580-9.

[20] D. Berardi, D. Calvanese, G. D. Giacomo, R. Hull, and M. Marcella. Automatic Composition of Transition-based Semantic Web Services with Messaging. In *Proceedings of the Very Large Database Conference (VLDB'05), Trondheim, Norway*, pages 613–624. VLDB Endowment, 2005.

[21] D. Berardi, G. De Giacomo, M. Mecella, and D. Calvanese. Composing Web Services with Nondeterministic Behavior. In *Proceedings of the International Conference on Web*

Services (ICWS'06), Chicago, IL, USA, pages 909–912. IEEE Computer Society, 2006. doi:10.1109/ICWS.2006.45.

[22] M. B. Blake and D. J. Cummings. Workflow Composition of Service Level Agreements. In *Proceedings of the IEEE International Conference on Services Computing, Salt Lake City, Utah, USA*, pages 138–145. IEEE Computer Society, July 2007. doi:10.1109/SCC.2007.136.

[23] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5(3):223–270, 1992. doi:10.1007/BF01807506.

[24] I. Brandic, S. Pllana, and S. Benkner. Specification, Planning, and Execution of QoS-aware Grid Workflows within the Amadeus Environment. *Concurrency and Computation: Practice and Experience*, 20(4):331–345, Mar. 2008. doi:10.1002/cpe.v20:4.

[25] P. A. Buhler, C. Starr, W. H. Schroder, and J. M. Vidal. Preparing for Service-Oriented Computing: A Composite Design Pattern for Stubless Web Service Invocation. In *Proceedings of 4th International Conference on Web Engineering (ICWE'04), Munich, Germany*, pages 603–604. Springer, July 2004. doi:10.1007/b99180.

[26] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. An Approach for QoS-aware Service Composition based on Genetic Algorithms. In *Proceedings of the Genetic and Computation Conference (GECCO'05), Washington DC, USA*, pages 1069–1075. ACM Press, 2005. doi:10.1145/1068009.1068189.

[27] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani. QoS-Aware Replanning of Composite Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05), Orlando, FL, USA*, pages 121–129. IEEE Computer Society, 2005. doi:10.1109/ICWS.2005.96.

[28] M. Carbone, K. Honda, N. Yoshida, and R. Milner. Structured Communication-Centred Programming for Web Services. In *Proceedings of the 16th European Symposium on Programming (ESOP'07), Barga, Portugal*, pages 2–17. Springer, 2007. doi:10.1007/978-3-540-71316-6_2.

[29] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of Service for Workflows and Web Service Processes. *Journal of Web Semantics*, 1(3):281–308, 2004. doi:10.1016/j.websem.2004.03.001.

[30] F. Casati, S. Ilnicki, L. jie Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and Dynamic Service Composition in eFlow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAISE'00), Stockholm, Sweden*, pages 13–31. Springer, 2000. doi:10.1007/3-540-45140-4.

[31] C. Cavaness. *Quartz Job Scheduling Framework: Building Open Source Enterprise Applications*. Prentice Hall, 1st edition, June 2006.

[32] A. Charfi. *Aspect-Oriented Workflow Languages: AO4BPEL and Applications*. PhD thesis, Technische Universität Darmstadt, 2007.

[33] A. Charfi and M. Mezini. AO4BPEL: An Aspect-Oriented Extension to BPEL. *World Wide Web Journal: Recent Advances on Web Services (special issue)*, 10(3), 2007. `doi:10.1007/s11280-006-0016-3`.

[34] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Software Engineering for Self-Adaptive Systems: A Research Road Map. In *Dagstuhl Seminar Proceedings*, 2008. Available from: `http://drops.dagstuhl.de/opus/volltexte/2008/1500/pdf/08031.SWM.Paper.1500.pdf`.

[35] R. Cole, D. Shur, and C. Villamizar. *RFC1932 – IP over ATM: A Framework Document*. Network Working Group, Apr. 1996. Informational. Available from: `http://www.rfc-archive.org/getrfc.php?rfc=1932`.

[36] E. Crawley, R. Nair, B. Rajagopalan, and H. Sandick. *RFC2386 – A Framework for QoS-based Routing in the Internet*. Network Working Group, Aug. 1998. Informational. Available from: `http://www.rfc-archive.org/getrfc.php?rfc=2368`.

[37] CS-Script – The C# Script Engine. Available from: `http://www.csscript.net/` [cited April 9, 2009].

[38] F. Curbera and N. Mukhi. Metadata-Driven Middleware for Web Services. In *4th International Conference on Web Information Systems Engineering (WISE'03), Rome, Italy*, pages 278–286. IEEE Computer Society, 2003. `doi:10.1109/WISE.2003.1254493`.

[39] A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web Services on Demand: WSLA-driven automated management. *IBM Systems Journal*, 43(1):136–158, 2004.

[40] G. Decker, O. Kopp, F. Leymann, K. Pfitzner, and M. Weske. Modeling Service Choreographies using BPMN and BPEL4Chor. In *Proceedings of the 20th International Conference on Advanced Information Systems Engineering (CAiSE'08), Montpellier, France*, volume 5074 of *Lecture Notes in Computer Science*, pages 79–93. Springer, June 2008. `doi:10.1007/978-3-540-69534-9_6`.

[41] G. Decker, O. Kopp, F. Leymann, and M. Weske. BPEL4Chor: Extending BPEL for Modeling Choreographies. In *Proceedings of the IEEE International Conference on Web Services (ICWS'07), Salt Lake City, Utah, USA*, pages 296–303. IEEE Computer Society, July 2007. `doi:10.1109/ICWS.2007.59`.

[42] G. Denaro, M. Pezze, and D. Tosi. Designing Self-Adaptive Service-Oriented Applications. In *Proceedings of the Fourth International Conference on Autonomic Computing (ICAC'07), Jacksonville, FL, USA*, pages 16–17. IEEE Computer Society, 2007. `doi:10.1109/ICAC.2007.13`.

[43] L. Duboc, D. S. Rosenblum, and T. Wicks. A framework for modelling and analysis of software systems scalability. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06), Shanghai, China*, pages 949–952. ACM Press, 2006. Doctoral Symposium. `doi:10.1145/1134285.1134460`.

[44] S. Dustdar and M. P. Papazoglou. Services and Service Composition – An Introduction. *it – Information Technology*, 50(2/2008), Feb. 2008. `doi:10.1524/itit.2008.0468`.

[45] S. Dustdar and W. Schreiner. A Survey on Web services Composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005. `doi:10.1504/IJWGS.2005.007545`.

[46] G. Dyaz, M. E. Cambronero, J. J. Pardo, V. Valero, and F. Cuartero. Automatic generation of Correct Web Services Choreographies and Orchestrations with Model Checking Techniques. In *Proceedings of the International Conference on Internet and Web Applications and Services (ICIW'06), Guadeloupe, French Caribbean*. IEEE Computer Society, Feb. 2006. `doi:10.1109/AICT-ICIW.2006.53`.

[47] J. Epstein, S. Matsumoto, and G. McGraw. Software Security and SOA: Danger, Will Robinson! *IEEE Security and Privacy*, 4(1):80–83, 2006. `doi:10.1109/MSP.2006.23`.

[48] T. Erl. *Service-Oriented Architecture: Concepts, Technology & Design*. Prentice Hall/PearsonPTR, 2005.

[49] R. Eshuis, P. W. P. J. Grefen, and S. Till. Structured service composition. In *Proceedings of the 4th International Conference on Business Process Management (BPM'06), Vienna, Austria*, pages 97–112. Springer, 2006. `doi:10.1007/11841760_8`.

[50] Esper – Event Stream and Complex Event Processing Platform. Available from: `http://esper.codehaus.org/` [cited April 11, 2009].

[51] O. Ezenwoye and S. M. Sadjadi. RobustBPEL2: Transparent Autonomization in Business Processes through Dynamic Proxies. In *Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS'07), Sedona, Arizona*, pages 17–24. IEEE Computer Society, Mar. 2007. `doi:10.1109/ISADS.2007.65`.

[52] L. Fei, Y. Fangchun, S. Kai, and S. Sen. A Policy-Driven Distributed Framework for Monitoring Quality of Web Services. In *Proceedings of the 2008 IEEE International Conference on Web Services (ICWS'08), Beijing, China*, pages 708–715. IEEE Computer Society, 2008. `doi:10.1109/ICWS.2008.123`.

[53] K. Fujii. Jpcap – a Java library for capturing and sending network packets [online]. Available from: `http://netresearch.ics.uci.edu/kfujii/jpcap/doc/` [cited March 20, 2009].

[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[55] Grid Resource Allocation Agreement Protocol (GRAAP) WG. Web Services Agreement Specification (WS-Agreement), Mar. 2007. Available from: `http://www.ogf.org/documents/GFD.107.pdf` [cited March 15, 2009].

[56] D. Groenewegen and E. Visser. Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns. In *Proceedings of the International Conference on Web Engineering (ICWE'08), Yorktown Heights, New York*, pages 175–188. IEEE Computer Society, July 2008. `doi:10.1109/ICWE.2008.15`.

[57] Y. Guan, A. K. Ghose, and Z. Lu. Using constraint hierarchies to support QoS-guided service composition. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, IL, USA*, pages 743–752. IEEE Computer Society, 2006. `doi:10.1109/ICWS.2006.143`.

[58] Hibernate – Relational Persistence for Java and .NET. Available from: `http://www.hibernate.org/` [cited April 6, 2009].

[59] J. S. Hiroshi Wada, Paskorn Champrasert and K. Oba. Multiobjective Optimization of SLA-aware Service Composition. In *Proceedings of the Workshop on Methodologies for Non-functional Properties in Services Computing, co-located with the IEEE Congress on Services (SERVICES'08), Honolulu, HI, USA*. IEEE Computer Society, July 2008. `doi:10.1109/SERVICES-1.2008.77`.

[60] A. Huber. A Transformation Engine for Resolving Web Service Heterogeneities within the VRESCo Runtime. Master's thesis, Technical University Vienna, 2009. (to appear).

[61] IBM. Web Service Level Agreement (WSLA) Language Specification, Jan. 2003. Available from: `http://www.research.ibm.com/wsla/` [cited March 15, 2009].

[62] International Telecommunication Union (ITU) [online]. Available from: `http://www.itu.int/` [cited March 15, 2009].

[63] S. Jackowski. *RFC1946 – Native ATM Support for ST2+*. Network Working Group, May 1996. Informational. Available from: `http://www.rfc-archive.org/getrfc.php?rfc=1946`.

[64] M. C. Jaeger. *Optimising Quality-of-Service for the Composition of Electronic Services*. PhD thesis, Technische Universität Berlin, Dec. 2006.

[65] M. C. Jaeger, G. Mühl, and S. Golze. QoS-aware Composition of Web Services: A Look at Selection Algorithms. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05), Orlando, FL, USA*, pages 807–808. IEEE Computer Society, July 2005. `doi:10.1109/ICWS.2005.95`.

[66] M. C. Jaeger, G. Mühl, and S. Golze. QoS-aware Composition of Web Services: An Evaluation of Selection Algorithms. In R. Meersman and Z. Tari, editors, *Proceedings 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, Nov. 2005. `doi:10.1007/11575771`.

[67] M. C. Jaeger, G. Rojec-Goldmann, and G. Mühl. QoS Aggregation for Service Composition using Workflow Patterns. In *Proceedings of the 8th International Enterprise Distributed Object Computing Conference (EDOC'04)*, pages 149–159. IEEE Computer Society, Sept. 2004. `doi:10.1109/EDOC.2004.10027`.

[68] M. C. Jaeger, G. Rojec-Goldmann, and G. Mühl. QoS Aggregation in Web Service Compositions. In *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05), Hong Kong, China*, pages 181–185. IEEE Computer Society, March 2005. `doi:10.1109/EEE.2005.110`.

[69] R. Jurca and B. Faltings. Reputation-based Service Level Agreements for Web Services. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC'05), Amsterdam, The Netherlands*, volume 3826 of *Lecture Notes in Computer Science*, pages 396–409. Springer, 2005. `doi:10.1007/11596141`.

[70] R. Jurca, B. Faltings, and W. Binder. Reliable QoS Monitoring Based on Client Feedback. In *Proceedings of the 16th International World Wide Web Conference (WWW'07), Banff, Alberta, Canada*, pages 1003–1012. ACM Press, 2007. `doi:10.1145/1242572.1242708`.

[71] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11(1):57–81, Mar. 2003. `doi:10.1023/A:1022445108617`.

[72] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01), Budapest, Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, June 2001.

[73] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finnland*, pages 220–242. Springer, June 1997.

[74] B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler. On Structured Workflow Modelling. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAISE'00), Stockholm, Sweden*, pages 431–445. Springer, 2000. `doi:10.1007/3-540-45140-4_29`.

[75] T. Laner. A Semantically Enriched Querying Language for the VRESCo Metamodel. Master's thesis, Technical University Vienna, 2009. (to appear).

[76] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. *Journal on Digital Libraries*, 6(3):235–246, June 2006. doi:10.1007/s00799-006-0002-5.

[77] P. Leitner. The DAIOS Framework - Dynamic, Asynchronous and Message-oriented Invocation of Web Services. Master's thesis, Technical University Vienna, 2007.

[78] P. Leitner, A. Michlmayr, and S. Dustdar. Towards Flexible Interface Mediation for Dynamic Service Invocations. In *Proceedings of the 3rd Workshop on Emerging Web Services Technology (WEWST'08), co-located with the 6th IEEE European Conference on Web Services (ECOWS'08), Dublin, Ireland*, pages 45–59. Springer, Nov. 2008.

[79] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. End-to-End Versioning Support for Web Services. In *Proceedings of the International Conference on Services Computing (SCC'08), Honolulu, Hawaii, USA*, pages 59–66. IEEE Computer Society, July 2008. doi:10.1109/SCC.2008.21.

[80] P. Leitner, F. Rosenberg, and S. Dustdar. DAIOS– Efficient Dynamic Web Service Invocation. *IEEE Internet Computing*, 13(3):72–80, May/June 2009. doi:10.1109/MIC.2009.57.

[81] Y. Liu, A. H. Ngu, and L. Zeng. QoS Computation and Policing in Dynamic Web Service Selection. In *Proceedings of the 13th International Conference on World Wide Web (WWW'04), New York, NY, USA*, pages 66–73. ACM Press, May 2004. doi:10.1145/1013367.1013379.

[82] J. Löwy. *Programming WCF Services*. O'Reilly, 2007.

[83] A. Mani and A. Nagarajan. Understanding Quality of Service for Web Services, Jan. 2002. Available from: http://www.ibm.com/developerworks/library/ws-quality.html [cited March 15, 2009].

[84] A. Marconi, M. Pistore, and P. Traverso. Implicit vs. Explicit Data-Flow Requirements in Web Service Composition Goals. In *Proceedings of the 4th International Conference Service-Oriented Computing (ICSOC'06), Chicago, IL, USA*, volume 4294 of *Lecture Notes in Computer Science*, pages 459–464. Springer, Dec. 2006. doi:10.1007/11948148_40.

[85] S. McIlraith and T. Son. Adapting Golog for Composition of Semantic Web Services. In *Proceedings of the 8th International Conference on Principles of Knowledge Representation and Reasoning (KR'02), Toulouse, France*. Morgan Kaufmann, 2002.

[86] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53, 2001. doi:10.1109/5254.920599.

[87] D. A. Menasce. QoS issues in Web services. *IEEE Internet Computing*, 6(6):72–75, November/December 2002. doi:10.1109/MIC.2002.1067740.

[88] D. A. Menasce. Composing Web Services: A QoS View. *IEEE Internet Computing*, 8(6):88–90, November/December 2004. `doi:10.1109/MIC.2004.57`.

[89] J. Mendling and M. Hafner. From WS-CDL Choreography to BPEL Process Orchestration. *Journal of Enterprise Information Management (JEIM)*, 21(5):525–542, 2008. Special Issue on MIOS 2005 Best Papers. `doi:10.1108/17410390810904274`.

[90] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Advanced Event Processing and Notifications in Service Runtime Environments. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08), Rome, Italy*, pages 115–125. ACM Press, July 2008. `doi:10.1145/1385989.1386004`.

[91] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. End-to-End Support for QoS-Aware Service Selection, Invocation and Mediation in VRESCo. Technical Report TUV-184-2009-03, Technical University Vienna, June 2009. Available from: `http://www.infosys.tuwien.ac.at/Staff/rosenberg/papers/TUV-1841-2009-03.pdf`.

[92] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Service Provenance in QoS-Aware Web Service Runtimes. In *Proceedings of the International Conference on Web Services (ICWS'09), Los Angeles, USA*. IEEE Computer Society, July 2009.

[93] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar. Towards Recovering the Broken SOA Triangle – A Software Engineering Perspective. In *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering (IW-SOSWE'07), Dubrovnik, Croatia*, pages 22–28. ACM Press, 2007. `doi:10.1145/1294928.1294934`.

[94] Microsoft. MGrammar Language Specification. Available from: `http://msdn.microsoft.com/en-us/library/dd129869.aspx` [cited April 15, 2009].

[95] Microsoft. Oslo Modeling Platform. Available from: `http://www.microsoft.com/soa/products/oslo.aspx` [cited April 15, 2009].

[96] Microsoft Cooperation Inc. Windows Workflow Foundation [online]. 2006. Available from: `http://msdn2.microsoft.com/en-us/library/ms734631.aspx` [cited March 15, 2009].

[97] Microsoft Cooperation Inc. Solver Foundation [online]. 2008. Available from: `http://www.solverfoundation.com` [cited May 10, 2009].

[98] Microsoft Cooperation Inc. Performance Counters [online]. 2009. Available from: `http://msdn.microsoft.com/en-us/library/w8f5kw2e(VS.71).aspx` [cited March 18, 2009].

[99] O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Adaptation for WS-BPEL. In *Proceedings of the 17th International International World Wide Web Conference (WWW'08), Beijing, China*, pages 815–824. ACM Press, Apr. 2008. doi:10.1145/1367497.1367607.

[100] O. Moser, F. Rosenberg, and S. Dustdar. VieDAME – Flexible and Robust BPEL Processes through Monitoring and Adaptation (Informal Demo Paper). In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany*, pages 917–918. ACM Press, May 2008. doi:10.1145/1370175.1370186.

[101] A. Mukhija, A. Dingwall-Smith, and D. S. Rosenblum. QoS-Aware Service Composition in Dino. In *Proceedings of the Fifth European Conference on Web Services (ECOWS'05), Halle (Saale), Germany*, pages 3–12. IEEE Computer Society, Nov. 2007. doi:10.1109/ECOWS.2007.24.

[102] C. Nagl, F. Rosenberg, and S. Dustdar. VIDRE – A Distributed Service-Oriented Business Rule Engine based on RuleML. In *Proceedings of the 10th International Conference on Enterprise Computing (EDOC'06), Hong Kong, China*, pages 35–44. IEEE Computer Society, Oct. 2006. doi:10.1109/EDOC.2006.67.

[103] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Addison Wesley, 1st edition, 2005.

[104] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3-4):313–341, 2008. doi:10.1007/s10515-008-0032-x.

[105] OASIS. OASIS Web Services Quality Model TC [online]. 2005. Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsqm [cited March 15, 2009].

[106] OASIS. UDDI Version 3 Specification, 2005. Available from: http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv3 [cited March 15, 2009].

[107] OASIS. Web Service Business Process Execution Language 2.0, 2006. Available from: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel [cited March 15, 2009].

[108] OASIS Web Services Reliable Exchange (WS-RX) TC. Web Services Reliable Messaging Policy Assertion (WS-RM Policy) Version 1.1, June 2007. OASIS Standard. Available from: http://docs.oasis-open.org/ws-rx/wsrmp/200702/wsrmp-1.1-spec-os-01.html [cited March 15, 2009].

[109] OASIS Web Services Reliable Messaging TC. WS-Reliability 1.1, Nov. 2004. OASIS Standard. Available from: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm` [cited March 15, 2009].

[110] OASIS Web Services Secure Exchange (WS-SX) TC. WS-SecurityPolicy 1.2, July 2007. OASIS Standard. Available from: `http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html` [cited March 15, 2009].

[111] OASIS Web Services Secure Exchange (WS-SX) TC. WS-Trust 1.3, Mar. 2007. OASIS Standard. Available from: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-sx` [cited March 15, 2009].

[112] OASIS Web Services Security TC. Web Services Security v1.1, Feb. 2006. OASIS Standard. Available from: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss` [cited March 15, 2009].

[113] E. Oberortner, U. Zdun, and S. Dustdar. Domain-Specific Languages for Service-Oriented Architectures: An Explorative Study. In P. Mähönen, K. Pohl, , and T. Priol, editors, *Proceedings of ServiceWave 2008, Madrid, Spain*, pages 159–170. Springer, 2008. `doi:10.1007/978-3-540-89897-9`.

[114] E. Oberortner, U. Zdun, and S. Dustdar. Tailoring a Model-Driven Quality-of-Service DSL for Various Stakeholders. In *Proceedings of the Workshop on Modeling in Software Engineering (MiSE'09), co-located with the 31th International Conference on Software Engineering (ICSE'09), Vancouver, Canada*. IEEE Computer Society, May 2009.

[115] C. Ouyang, E. Verbeek, W. M. van der Aalst, S. Breutel, M. Dumas, and A. H. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, July 2007. `doi:10.1016/j.scico.2007.03.002`.

[116] I. V. Papaioannou, D. T. Tsesmetzis, I. G. Roussaki, and M. E. Anagnostou. A QoS Ontology Language for Web-Services. In *International Conference on Advanced Information Networking and Applications (AINA'06)*, pages 101–106. IEEE Computer Society, 2006. `doi:10.1109/AINA.2006.51`.

[117] 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. `doi:10.1109/MC.2007.400`.

[118] C. Pautasso. BPEL for REST. In *Proceedings of the 6th International Conference on Business Process Management (BPM'08), Milan, Italy*, volume 5240 of *Lecture Notes in Computer Science*, pages 278–293. Springer, Sept. 2008. `doi:10.1007/978-3-540-85758-7_21`.

[119] C. Pautasso and G. Alonso. The JOpera Visual Composition Language. *Journal of Visual Languages and Computing (JVLC)*, 16:119–152, 2005. Available from: `http://www.jopera.org`.

[120] C. Pautasso, O. Zimmermann, and F. Leymann. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International International World Wide Web Conference (WWW'08), Beijing, China*, pages 805–814, Apr. 2008. `doi:10.1145/1367497.1367606`.

[121] B. Pernici, editor. *Mobile Information Systems*. Springer, Apr. 2006.

[122] pi4 Technologies Foundation. pi4soa [online]. 2007. Available from: `http://sourceforge.net/projects/pi4soa/` [cited March 15, 2009].

[123] M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated Synthesis of Composite BPEL4WS Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 293–301. IEEE Computer Society, 2005. `doi:10.1109/ICWS.2005.27`.

[124] C. Platzer, F. Rosenberg, and S. Dustdar. Enhancing Web Service Discovery and Monitoring with Quality of Service Information. In P. Periorellis, editor, *Securing Web Services: Practical Usage of Standards and Specifications*. Idea Group Inc. (IGI), Nov. 2007.

[125] S. Ran. A model for web services discovery with QoS. *SIGecom Exchanges*, 4(1):1–10, 2003. `doi:10.1145/844357.844360`.

[126] L. Richardson and S. Ruby. *RESTful Web Services*. O'Reilly Media, Inc., May 2007.

[127] G. Rocher. *The Definitive Guide to Grails*. Apress, 2006.

[128] F. Rosenberg and P. Celikovic. *Vienna Composition Language (VCL) Specification*. Technical University Vienna. Available from: `http://www.infosys.tuwien.ac.at/staff/rosenberg/vresco/` [cited April 15, 2009].

[129] F. Rosenberg, P. Celikovic, A. Michlmayr, P. Leitner, and S. Dustdar. An End-to-End Approach for QoS-Aware Service Composition. In *Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference (EDOC'09), Auckland, New Zealand*. IEEE Computer Society, 2009.

[130] F. Rosenberg, F. Curbera, M. J. Duftler, and R. Khalaf. Composing RESTful Services and Collaborative Workflows: A Lightweight Approach. *Internet Computing*, 12:24–31, September/October 2008. `doi:10.1109/MIC.2008.98`.

[131] F. Rosenberg, C. Enzi, A. Michlmayr, C. Platzer, and S. Dustdar. Integrating Quality of Service Aspects in Top-Down Business Process Development using WS-CDL and WS-BPEL. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing*

Conference (EDOC'07), Annapolis, Maryland, USA., pages 15–26. IEEE Computer Society, Oct. 2007. doi:10.1109/EDOC.2007.23.

[132] F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar. Towards Composition as a Service - A Quality of Service Driven Approach. In *Proceedings of the First IEEE Workshop on Information and Software as Services (WISS'09), co-located with the 25th International Conference on Data Engineering (ICDE'09), Shanghai, China*, pages 1733–1740. IEEE Computer Society, Mar. 2009. doi:10.1109/ICDE.2009.153.

[133] F. Rosenberg, P. Leitner, A. Michlmayr, and S. Dustdar. Integrated Metadata Support for Web Service Runtimes. In *Proceedings of the Middleware for Web Services Workshop (MWS'08), co-located with the 12th IEEE International Distributed Object Computing Conference (EDOC'08), Munich, Germany*, pages 361–368. IEEE Computer Society, Sept. 2008. doi:10.1109/EDOCW.2008.38.

[134] F. Rosenberg, A. Michlmayr, and S. Dustdar. Top-Down Business Process Development and Execution using Quality of Service Aspects. *Enterprise Information Systems*, pages 459–475, November 2008. doi:10.1080/17517570802395626.

[135] F. Rosenberg, C. Nagl, and S. Dustdar. Applying Distributed Business Rules – The VIDRE Approach. In *Proceedings of the IEEE International Conference on Services Computing (SCC'06), Chicago, USA*, pages 471–478. IEEE Computer Society, Sept. 2006. doi:10.1109/SCC.2006.22.

[136] F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, pages 205–212. IEEE Computer Society, Sept. 2006. doi:10.1109/ICWS.2006.39.

[137] J. Skene, F. Raimondi, and W. Emmerich. Service-Level Agreements for Electronic Services. *IEEE Transactions on Software Engineering*, 2009. (Forthcoming).

[138] C. U. Smith and L. G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison-Wesley Professional, 1 edition, Sept. 2001.

[139] H. G. Song and K. Lee. sPAC (Web Services Performance Analysis Center): Performance Analysis and Estimation Tool of Web Services. In *Proceedings of the 3rd International Conference on Business Process Management (BPM'05), Nancy, France*, pages 109–119. Springer, 2005.

[140] W. R. Stevens. *TCP/IP Illustrated I: The Protocols*. Addison-Wesley, 1994.

[141] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[142] S. Thakkar, C. A. Knoblock, J. L. Ambite, and C. Shahabi. Dynamically Composing Web Services from Online Sources. In *Workshop on Intelligent Service Integration co-located with the 18th National Conference on Artificial Intelligence (AAAI'02), Edmonton, Canada*, pages 1–7, 2002.

[143] N. Thio and S. Karunasekera. Automatic measurement of a QoS metric for Web service recommendation. In *Proceedings of the Australian Software Engineering Conference (ASWEC'05), Brisbane, Australia*, pages 202–211. IEEE Computer Society, 2005. `doi:10.1109/ASWEC.2005.16`.

[144] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Schiller. A Concept for QoS Integration in Web Services. In *Proceedings of the 1st Web Services Quality Workshop (WQW'03), Rome, Italy*, pages 149–155. IEEE Computer Society, 2003.

[145] M. Tian, A. Gramm, H. Ritter, and J. Schiller. Efficient Selection and Monitoring of QoS-aware Web services with the WS-QoS Framework. In *Proceedings of the International Conference on Web Intelligence (WI'04), Beijing, China*, pages 152–158. IEEE Computer Society, Sept. 2004. `doi:10.1109/WI.2004.60`.

[146] Apache Tomcat. Available from: `http://tomcat.apache.org/` [cited April 24, 2009].

[147] V. Tosic, B. Pagurek, K. Patel, B. Esfandiari, and W. Ma. Management applications of the Web Service Offerings Language (WSOL). *Information Systems*, 30(7):564–586, 2005. `doi:10.1016/j.is.2004.11.005`.

[148] H.-L. Truong, R. Samborski, and T. Fahringer. Towards a Framework for Monitoring and Analyzing QoS Metrics of Grid Services. In *Proceedings of the International Conference on e-Science and Grid Computing (e-Science'06), Amsterdam, The Netherlands*, pages 65–72. IEEE Computer Society, 2006. `doi:10.1109/E-SCIENCE.2006.142`.

[149] T. Unger, F. Leymann, S. Mauchart, and T. Scheibler. Aggregation of Service Level Agreements in the Context of Business Processes. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC'08), Munich, Germany*, pages 43–52. IEEE Computer Society, 2008. `doi:10.1109/EDOC.2008.29`.

[150] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003. `doi:10.1023/A:1022883727209`.

[151] W. M. van der Aalst, M. Dumas, A. H. ter Hofstede, N. Russell, P. Wohed, and . H. M. W. Verbeek. Life After BPEL? In *Proceedings of the International Workshop on Web Services and Formal Methods (WS-FM'05), Versailles, France*, pages 35–50. Springer, 2005.

[152] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: yet another workflow language. *Information Systems*, 30(4):245–275, 2005. `doi:10.1016/j.is.2004.02.002`.

[153] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000. doi:10.1145/352029.352035.

[154] E. Visser. WebDSL: A Case Study in Domain-Specific Language Engineering. Technical Report TUD-SERG-2008-023, TU Deflt, The Netherlands, 2008. Available from: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2008-023.pdf [cited April 28, 2009].

[155] A. Vogel, B. Kerhervé, G. von Bochmann, and J. Gecsei. Distributed Multimedia and QS: A Survey. *IEEE MultiMedia*, 2(2):10–19, 1995. doi:10.1109/93.388195.

[156] M. Völter and T. Stahl. *Model-Driven Software Development : Technology, Engineering, Management*. John Wiley & Sons, June 2006.

[157] W3C. Web Services Description Language (WSDL) 1.1, 2001. Available from: http://www.w3.org/TR/wsdl [cited March 15, 2009].

[158] W3C. SOAP Version 1.2, 2003. Available from: http://www.w3.org/TR/soap [cited March 15, 2009].

[159] W3C. OWL-S: Semantic Markup for Web Services, 2004. Available from: http://www.w3.org/Submission/OWL-S/ [cited March 15, 2009].

[160] W3C. Resource Description Framework (RDF), 2004. Available from: http://www.w3.org/RDF/ [cited April 5, 2009].

[161] W3C. Web Services Choreography Description Language (WS-CDL), Nov. 2005. Available from: http://www.w3.org/TR/ws-cdl-10/ [cited March 15, 2009].

[162] W3C. Web Services Eventing (WS-Eventing), 2006. Available from: http://www.w3.org/Submission/WS-Eventing/ [cited Arpil 28, 2009].

[163] W3C. Semantic Annotations for WSDL and XML Schema, 2007. Available from: http://www.w3.org/TR/sawsdl/ [cited May 22, 2009].

[164] W3C. Web Services Policy Attachment, Sept. 2007. Available from: http://www.w3.org/TR/ws-policy-attach/ [cited March 15, 2009].

[165] W3C. Web Services Policy Framework v1.5, Sept. 2007. Available from: http://www.w3.org/TR/ws-policy/ [cited March 15, 2009].

[166] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, 1st edition, 2005.

[167] N. Wickramage and S. Weerawarana. A Benchmark for Web Service Frameworks. In *Proceedings of the IEEE International Conference on Service Computing (SCC'05)*, pages 233–240. IEEE Computer Society, July 2005. doi:10.1109/SCC.2005.9.

[168] WinPcap: The Windows Packet Capture Library [online]. Available from: `http://www.winpcap.org/` [cited March 22, 2009].

[169] Wireshark – network protocol analyzer [online]. Available from: `http://www.wireshark.org/` [cited March 22, 2009].

[170] T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(6):1–26, 2007. `doi:10.1145/1232722.1232728`.

[171] J. M. Zaha, A. P. Barros, M. Dumas, and A. H. M. ter Hofstede. Let's Dance: A Language for Service Behavior Modeling. In R. Meersman and Z. Tari, editors, *Proceedings of the International Conference on Cooperative Information Systems (CoopIS'06), Montpellier, France*, volume 4275 of *Lecture Notes in Computer Science*, pages 145–162. Springer, 2006. `doi:http://dx.doi.org/10.1007/11914853_10`.

[172] J. M. Zaha, M. Dumas, A. H. M. ter Hofstede, A. P. Barros, and G. Decker. Service Interaction Modeling: Bridging Global and Local Views. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC'06), Hong Kong, China*, pages 45–55, Oct. 2006. `doi:10.1109/EDOC.2006.50`.

[173] U. Zdun, C. Hentrich, and S. Dustdar. Modeling Process-Driven and Service-Oriented Architectures Using Patterns and Pattern Primitives. *ACM Transactions on the Web (TWEB)*, 1(3):14:1–14:44, 2007. `doi:10.1145/1281480.1281484`.

[174] U. Zdun, C. Hentrich, and W. M. van der Aalst. A Survey of Patterns for Service-Oriented Architectures. *International Journal of Internet Protocol Technology*, 1(3):132–143, 2006. `doi:10.1504/IJIPT.2006.009739`.

[175] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality Driven Web Services Composition. In *Proceedings of the 12th International Conference on World Wide Web (WWW'03), Budapest, Hungary*, pages 411–421. ACM Press, 2003. `doi:10.1145/775152.775211`.

[176] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004. `doi:10.1109/TSE.2004.11`.

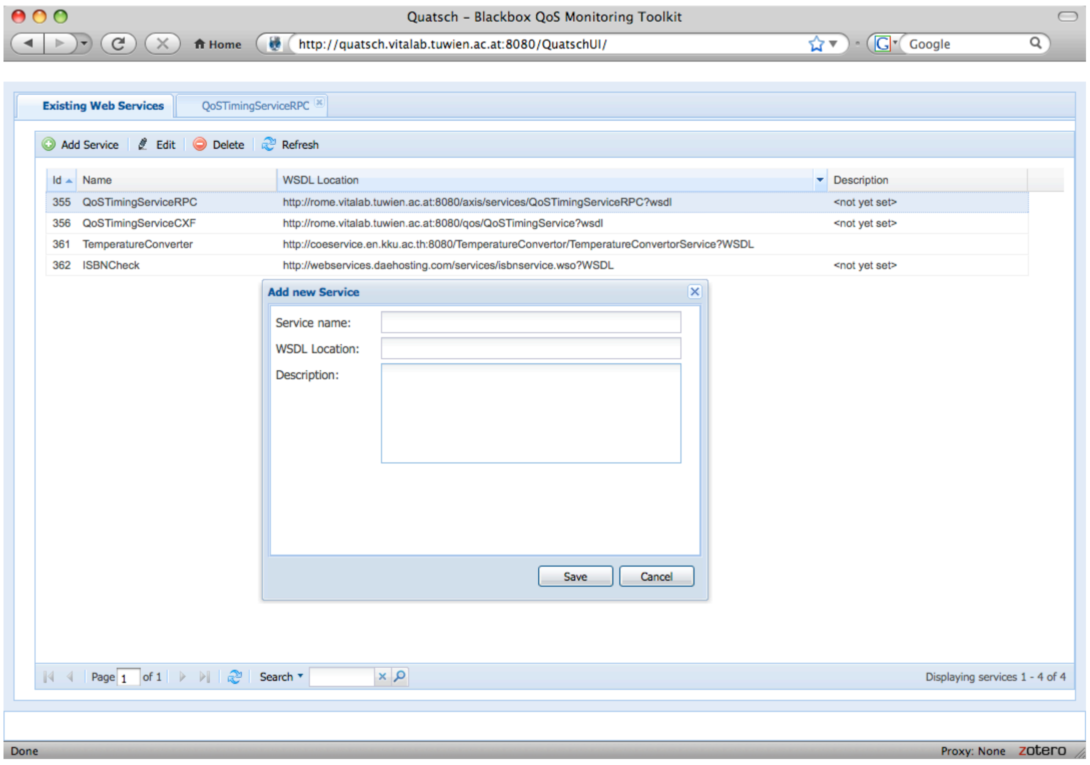
[177] L. Zeng, H. Lei, and H. Chang. Monitoring the QoS for Web Services. In *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC'07), Vienna, Austria*, pages 132–144. Springer, 2007. `doi:10.1007/978-3-540-74974-5_11`.

[178] C. Zhou, L.-T. Chia, and B.-S. Lee. DAML-QoS Ontology for Web Services. In *IEEE International Conference on Web Services (ICWS'04), San Diego, CA, USA*, pages 472–479. IEEE Computer Society, 2004. `doi:10.1109/ICWS.2004.1314772`.

Appendix A

QUATSCH Tool Support

In Figure A.1, a screenshot of the UI is depicted. The open dialog can be used to enter all the details about a service that should be monitored. After entering the details, QUATSCH starts preprocessing the service and it is then immediately available and several evaluation configurations can be defined.


Quatsch – Blackbox QoS Monitoring Toolkit
Home
http://quatsch.vitalab.tuwien.ac.at:8080/QuatschUI/
Google
Existing Web Services
QoSTimingServiceRPC
Add Service
Edit
Delete
Refresh

Id	Name	WSDL Location	Description
355	QoSTimingServiceRPC	http://rome.vitalab.tuwien.ac.at:8080/axis/services/QoSTimingServiceRPC?wsdl	<not yet set>
356	QoSTimingServiceCXF	http://rome.vitalab.tuwien.ac.at:8080/qos/QoSTimingService?wsdl	<not yet set>
361	TemperatureConverter	http://coeservice.en.kku.ac.th:8080/TemperatureConvertor/TemperatureConvertorService?WSDL	
362	ISBNCheck	http://webservices.daehosting.com/services/isbnservice.wso?WSDL	<not yet set>

Add new Service

Service name:
WSDL Location:
Description:

Figure A.1: QUATSCH UI - Add Service

In Figure A.2, the dynamic chart builder is shown. On the left-side, the data on the chart can be defined and customized. The chart itself is then visible on the right-side of the screen.

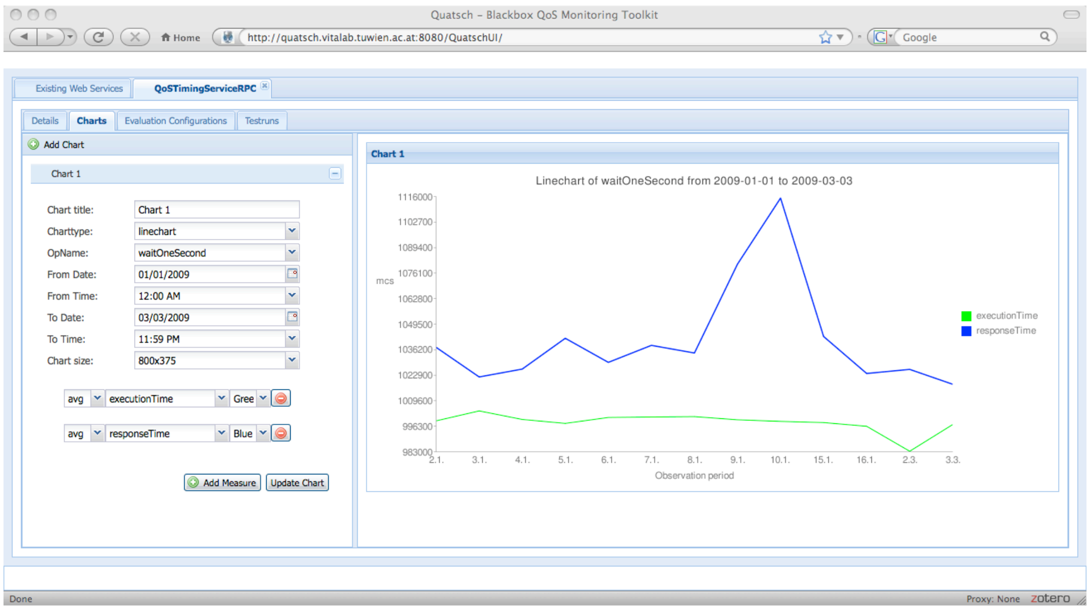


Chart title:	Chart 1
Charttype:	linechart
OpName:	waitOneSecond
From Date:	01/01/2009
From Time:	12:00 AM
To Date:	03/03/2009
To Time:	11:59 PM
Chart size:	800x375

avg	executionTime	Gree
avg	responseTime	Blue

Figure A.2: QUATSCH UI - Dynamic Charts

Appendix B

VCL Example Listing

Listing B.1 depicts the VCL code for the cell phone number porting process introduced in Chapter 6 (page 92).

```
composition TelcoCasestudy;

# define required features
feature Crm, ⋆.TelcoCasestudy.CustomerService.CustomerLookup;
feature LookupPartner, ⋆.PhoneManagementService.LookupPartner;
feature PortCheck, ⋆.PortingService.PortabilityCheck;
feature PortNumber, ⋆.PortingService.PortNumber;
feature ActivatePort, ⋆.PhoneManagementService.ActivatePortedNumber;
feature Notify, ⋆.NotificationService.NotifyUser;

# define global constraints
constraint global {
  input = {
    long customerId;
    string numberToPort;
  }
  output = {
    string status;
  }
  qos = {
    responseTime = 4500;
    availability = 0.90;
    reliableMessage = true;
  }
}

# define feature constraints
constraint Crm {
  input = {
    CustomerLookupRequest[
      int CustomerId;
    ]
  }
  output = {
    CustomerLookupResponse[
      string Firstname;
      string Lastname;
      string PhoneNumber;
      string Mail;
      string Street;
      string Zip;
      string City;
    ]
  }
  qos = {
    responseTime = 2500, required;
    availability =  0.95, required;
    reliableMessage = true, weak;
    security = X509, strong;
    accuracy = 0.9, weak;
    throughput = 100, weak;
    price = 10.5, weak;
  }
}

constraint LookupPartner {
  input = {
    LookupPartnerRequest[
      string NumberToPort;
    ]
  }
  output = {
    LookupPartnerResponse[
      string ProviderName;
    ]
  }
  qos = {
    responseTime = 2500, required;
    availability = 0.95;
    reliableMessage = true;
  }
}

constraint PortCheckOne {
  input = {
    PortabilityCheckRequest[
      string NumberToPort;
    ]
  }
  output = {
    PortabilityCheckResponse[
      int IsPortable;
    ]
  }
}

constraint PortNumberOne {
  input = {
    PortNumberRequest[
      string NumberToPort;
    ]
  }
  output = {
    PortNumberResponse[
      int IsPorted;
    ]
  }
}

constraint PortCheck {
  input = {
    PortabilityCheckRequest[
      string NumberToPort;
      ]
    }
    output = {
      PortabilityCheckResponse[
        int IsPortable;
      ]
    }
}
constraint PortNumber {
    input = {
      PortNumberRequest[
        string NumberToPort;
      ]
    }
    output = {
      PortNumberResponse[
        int IsPorted;
      ]
    }
}

constraint ActivatePort {
    input = {
      ActivatePortedNumberRequest[
        string CustomerId;
        string PortedNumber;
      ]
    }
    output = {
      ActivatePortedNumberResponse[
        string Status;
      ]
    }
}

constraint Notify {
    input = {
      NotifyUserRequest[
        string Mail;
        string Message;
      ]
    }
    output = {
      NotifyUserResponse[
        string Status;
      ]
    }
}

# business protocol specification
invoke Crm {
    CustomerLookupRequest[
      CustomerId = customerId;
    ]
}

invoke LookupPartner{
    LookupPartnerRequest[
      NumberToPort = numberToPort;
    ]
}
```

```

invoke PortCheck {
  PortabilityCheckRequest[
    NewProvider = LookupPartner.LookupPartnerRequest.ProviderName;
    NewNumber = numberToPort;
  ]
}

check (PortCheck.PortabilityCheckResponse.IsPortable = true)
{
  invoke PortNumber {
    PortNumberRequest[
      PortedNumber = numberToPort;
    ]
  }
}
else
{
  throw "Number can’t be ported by external provider";
}

check (PortNumber.PortNumberResponse.IsPorted = true)
{
  invoke ActivatePort {
    ActivatePortedNumberRequest [
      CustomerId = customerId;
      NumberToPort = Crm.CustomerLookupResponse.PhoneNumber;
    ]
  }
  invoke Notify {
    NotifyUserRequest [
      Mail = Crm.CustomerLookupResponse.PhoneNumber;
      Message = "Phone number ported";
    ]
  }
}
else
{
   throw "Problem occurred on external partner side";
}

return {
  status = "Ported";
}
```

Listing B.1: Telco Example Implementation

Listing B.2 shows the textual representation of the VCL composition from Listing B.1 according to the structured composition language introduced by Eshuis et al. [49].

```
{
  AND {
    SEQ [
      ROOT,
      AND {
        SEQ [ Crm(CustomerLookup) ],
        SEQ [
          LookupPartner(LookupPartner),
          PortCheck(PortabilityCheck),
          IFTHENBLOCK,
          XOR {
            SEQ [THENBLOCK, PortNumber(PortNumber)],
            SEQ [ELSEBLOCK, THROW]
          },
          SYNC
        ]
      },
      IFTHENBLOCK,
      XOR {
        SEQ [
          THENBLOCK,
          AND{
            SEQ [ActivatePort(ActivatePortedNumber)],
            SEQ [Notify(NotifyUser)]
          }
        ],
        SEQ [ELSEBLOCK,THROW]
      },
      SYNC
    ]
  }
}
```

Listing B.2: Structured Composition Representation

Curriculum Vitae

Florian Rosenberg

1 Personal Information

Current Position:	University Assistant (Faculty Member)
Address (Work):	Distributed Systems Group Information Systems Institute Technical University Vienna Argentinierstrasse 8/184-1, 1040 Vienna, Austria
Phone (Work):	+43 1 58801 18418
Fax (Work):	+43 1 58801 18491
Email:	`florian@infosys.tuwien.ac.at`
Web:	`http://www.florianrosenberg.com`
Data and Place of Birth:	19. May 1981, Steyr, Austria
Citizenship:	Austrian

2 Education

PhD Studies in Computer Science 02/2005 - 06/2009
Technical University Vienna, Austria

Thesis: *QoS-Aware Composition of Adaptive Service-Oriented Applications*
Supervision:
Prof. Dr. Schahram Dustdar – Technical University Vienna, Austria
Prof. M. Brian Blake, PhD – University of Notre Dame, USA

Software Engineering Studies (equivalent to MSc) 10/1999 – 10/2004
Upper Austrian University of Applied Sciences, Hagenberg, Austria
University of Linz (1st year)

Thesis: *A Configurable Deep Web MetaSearch Engine Based on Lixto*

3 Work Experience

University Assistant 12/2005 – now
Distributed Systems Group, Technical University Vienna, Austria

PhD Co-Op Student 06/2008 - 09/2008
IBM T.J. Watson Research Center, New York, USA

PhD Co-Op Student 06/2007 - 12/2007
IBM T.J. Watson Research Center, New York, USA

Research Assistant 01/2005 – 12/2005
Distributed Systems Group, Technical University Vienna, Austria

Java Developer 02/2004 – 08/2004
Lixto Software GmbH, Vienna, Austria

Java Developer 09/2003 – 01/2004
Siemens PSE, Vienna, Austria

Other summer internships and project work during undergraduate studies at IBM Austria, Racon Software GmbH and DaimlerChrysler Research.

4 Research Interests

- Software Engineering, especially Software Architecture and Software Composition
- Service-Oriented Computing (SOC)
- Quality of Service (QoS) issues in SOC
- Middleware and Web technologies

5 Publications

Journals

[1] Christian Platzer, Florian Rosenberg, and Schahram Dustdar. Web Service Clustering using Multi-Dimensional Angles as Proximity Measures. *ACM Transactions on Internet Technologies*, 2009. (forthcoming).

[2] Philipp Leitner, Florian Rosenberg, and Schahram Dustdar. DAIOS – Efficient Dynamic Web Service Invocation. *IEEE Internet Computing*, 13(3):72–80, May/June 2009. `doi:10.1109/MIC.2009.57`.

[3] Florian Rosenberg, Anton Michlmayr, and Schahram Dustdar. Top-Down Business Process Development and Execution using Quality of Service Aspects. *Enterprise Information Systems*, 2:459–475, November 2008. `doi:10.1080/17517570802395626`.

[4] Florian Rosenberg, Francisco Curbera, Matthew J. Duftler, and Rania Khalaf. Composing RESTful Services and Collaborative Workflows: A Lightweight Approach. *Internet Computing*, 12:24–31, September/October 2008. `doi:10.1109/MIC.2008.98`.

[5] Michael Mrissa, Chirine Ghedira, Djamal Benslimane, Zakaria Maamar, Florian Rosenberg, and Schahram Dustdar. A Context-based Mediation Approach to Compose Semantic Web Services. *ACM Transactions on Internet Technologies, Special Special Issue on Semantic Web Services: Issues, Solutions and Applications*, 8(1), 2007. doi:10.1145/1294148.1294152.

[6] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A Survey on Context-Aware Systems. *Journal on Ad Hoc and Ubiquitous Computing*, 2(4):263–277, 2007. doi:10.1504/IJAHUC.2007.014070.

Conferences, Workshops and Demos

[7] Florian Rosenberg, Predrag Celikovic, Anton Michlmayr, Philipp Leitner, and Schahram Dustdar. An End-to-End Approach for QoS-Aware Service Composition. In *Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference (EDOC'09), Auckland, New Zealand.* IEEE Computer Society, 2009.

[8] Branimir Wetzstein, Philipp Leitner, Florian Rosenberg, Ivona Brandic, Frank Leymann, and Schahram Dustdar. Monitoring and Analyzing Influential Factors of Business Process Performance. In *Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference (EDOC'09), Auckland, New Zealand.* IEEE Computer Society, 2009.

[9] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Service Provenance in QoS-Aware Web Service Runtimes. In *Proceedings of the International Conference on Web Services (ICWS'09), Los Angeles, USA.* IEEE Computer Society, July 2009. Acceptance rate: 15,6%.

[10] Florian Rosenberg, Philipp Leitner, Anton Michlmayr, Predrag Celikovic, and Schahram Dustdar. Towards Composition as a Service - A Quality of Service Driven Approach. In *Proceedings of the First IEEE Workshop on Information and Software as Services (WISS'09), co-located with the 25th International Conference on Data Engineering (ICDE'09), 29. March 2009, Shanghai, China.*, pages 1733–1740. IEEE Computer Society, March 2009. doi:10.1109/ICDE.2009.153.

[11] Florian Rosenberg, Philipp Leitner, Anton Michlmayr, and Schahram Dustdar. Integrated Metadata Support for Web Service Runtimes. In *Proceedings of the Middleware for Web Services Workshop (MWS'08), co-located with the 12th IEEE International Distributed Object Computing Conference (EDOC'08), Munich, Germany.*, pages 361–368. IEEE Computer Society, September 2008. doi:10.1109/EDOCW.2008.38.

[12] Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar. End-to-End Versioning Support for Web Services. In *Proceedings of the International Conference on Services Computing, Honolulu, Hawaii, USA*, pages 59–66. IEEE Computer Society, July 2008. Acceptance rate: 18%. doi:10.1109/SCC.2008.21.

[13] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram Dustdar. Advanced Event Processing and Notifications in Service Runtime Environments. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08), Rome, Italy*, pages 115–125. ACM Press, July 2008. doi:10.1145/1385989.1386004.

[14] Anton Michlmayr, Philipp Leitner, Florian Rosenberg, and Schahram Dustdar. Publish/Subscribe in the VRESCo SOA Runtime (Demo Paper). In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08), Rome, Italy*, pages 317–320. ACM Press, July 2008. `doi:10.1145/1385989.1386031`.

[15] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. VieDAME – Flexible and Robust BPEL Processes through Monitoring and Adaptation (Informal Demo Paper). In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany*, pages 917–918. ACM Press, May 2008. `doi:10.1145/1370175.1370186`.

[16] Oliver Moser, Florian Rosenberg, and Schahram Dustdar. Non-Intrusive Monitoring and Adaption for WS-BPEL. In *Proceedings of the 17th International World Wide Web Conference (WWW'08), Beijing, China.*, pages 815–824. ACM Press, April 2008. Acceptance rate: 11% (97 of 880). `doi:10.1145/1367497.1367607`.

[17] Florian Rosenberg, Christian Enzi, Anton Michlmayr, Christian Platzer, and Schahram Dustdar. Integrating Quality of Service Aspects in Top-Down Business Process Development using WS-CDL and WS-BPEL. In *Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC'07), Annapolis, Maryland, USA.* IEEE Computer Society, October 2007. Acceptance rate: 28% (33 of 115). `doi:10.1109/EDOC.2007.23`.

[18] 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. `doi:10.1145/1294928.1294934`.

[19] Lukasz Juszczyk, Anton Michlmayr, Christian Platzer, Florian Rosenberg, Alexander Urbanec, and Schahram Dustdar. Large Scale Web Service Discovery and Composition using High Performance In-Memory Indexing. In *Proceedings of the IEEE Joint Conference on E-Commerce Technology (CEC'07) and Enterprise Computing, E-Commerce and E-Services (EEE'07), Tokyo, Japan.* IEEE Computer Society, July 2007. `doi:doi/10.1109/CEC-EEE.2007.60`.

[20] Christoph Nagl, Florian Rosenberg, and Schahram Dustdar. ViDRE – A Distributed Service-Oriented Business Rule Engine based on RuleML. In *Proceedings of the 10th International Conference on Enterprise Computing (EDOC'06), Hong Kong, China*, pages 35–44. IEEE Computer Society, October 2006. Acceptance rate: 24%. `doi:10.1109/EDOC.2006.67`.

[21] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, pages 205–212. IEEE Computer Society, September 2006. Acceptance rate: 18%. `doi:10.1109/ICWS.2006.39`.

[22] Florian Rosenberg, Christoph Nagl, and Schahram Dustdar. Applying Distributed Business Rules – The ViDRE Approach. In *Proceedings of the IEEE International*

Conference on Services Computing (SCC'06), Chicago, USA, pages 471–478. IEEE Computer Society, September 2006. `doi:10.1109/SCC.2006.22`.

[23] Marco Aiello, Florian Rosenberg, Christian Platzer, Agata Ciabattoni, and Schahram Dustdar. Service QoS Composition at the Level of Part Names. In *Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM'06), Vienna, Austria, September 8-9, 2006*, volume 4184 of *Lecture Notes in Computer Science*, pages 24–37. Springer, 2006. `doi:10.1007/11841197_2`.

[24] Marco Aiello, Christian Platzer, Florian Rosenberg, Huy Tran, Martin Vasko, and Schahram Dustdar. Web Service Indexing for Efficient Retrieval and Composition. In *Proceedings of the IEEE Joint Conference on E-Commerce Technology (CEC'06) and Enterprise Computing, E-Commerce and E-Services (EEE'06), San Francisco, USA*, pages 63–65. IEEE Computer Society, June 2006. `doi:10.1109/CEC-EEE.2006.96`.

[25] Florian Rosenberg and Schahram Dustdar. Towards a Distributed Service-Oriented Business Rules System. In *Proceedings of the 3th European Conference on Web Services (ECOWS'05), Växjö, Sweden*, pages 14–23. IEEE Computer Society, November 2005. `doi:10.1109/ECOWS.2005.28`.

[26] Johann Oberleitner, Florian Rosenberg, and Schahram Dustdar. A Lightweight Model-Driven Orchestration Engine for e-Services. In *Proceedings of the 6th International Workshop on Technologies for E-Services (TES'05) Trondheim, Norway, September 2-3, 2005, Revised Selected Papers*, volume 3811 of *Lecture Notes in Computer Science*, pages 48–57. Springer, 2005. `doi:10.1007/11607380_5`.

[27] Florian Rosenberg and Schahram Dustdar. Business Rules Integration in BPEL – A Service-Oriented Approach. In *Proceedings of the 7th International IEEE Conference on E-Commerce Technology (CEC'05), Munich, Germany*, pages 476–479. IEEE Computer Society, February 2005. `doi:10.1109/ICECT.2005.25`.

[28] Florian Rosenberg and Schahram Dustdar. Design and Implementation of a Service-Oriented Business Rules Broker. In *Proceedings of the 1st IEEE International Workshop on Service-oriented Solutions for Cooperative Organizations (SoS4CO'05), Munich, Germany*, pages 55–63. IEEE Computer Society, February 2005. `doi:10.1109/CECW.2005.10`.

Book Chapters

[29] Philipp Leitner, Florian Rosenberg, Anton Michlmayr, Andreas Huber, and Schahram Dustdar. A Mediator-Based Approach to Resolving Interface Heterogeneity of Web Services. In Walter Binder and Schahram Dustdar, editors, *Post-proceedings of the 3rd Workshop on Emerging Web Services Technology (WEWST'08), Dublin, Ireland.* Birkhäuser, 2009. forthcoming.

[30] Anton Michlmayr, Philipp Leitner, Florian Rosenberg, and Schahram Dustdar. Event Processing in Web Service Runtime Environments. In Annika Hinze and Alex Buchmann, editors, *Handbook of Research on Advanced Distributed Event-Based Systems, Publish/Subscribe and Message Filtering Technologies.* IGI Global, 2009. forthcoming.

[31] Florian Rosenberg, Anton Michlmayr, Christoph Nagl, and Schahram Dustdar. Distributed Business Rules within Service-Centric Systems. In Kuldar Taveter Dragan Gasevic, Adrian Giurca, editor, *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*. IGI Global, 2009. forthcoming.

[32] Christian Platzer, Florian Rosenberg, and Schahram Dustdar. Enhancing Web Service Discovery and Monitoring with Quality of Service Information. In Panos Periorellis, editor, *Securing Web Services: Practical Usage of Standards and Specification*. Idea Publishing Group, 2007.

Theses

[33] Florian Rosenberg. *QoS-Aware Composition of Adaptive Service-Oriented Systems*. PhD thesis, Technical University Vienna, Austria, June 2009.

[34] Florian Rosenberg. A Configurable Deep Web MetaSearch Engine Based on Lixto. Master's thesis, Upper Austria University of Applied Sciences (Campus Hagenberg), Austria, September 2004.

6 Professional Activities

Conference Organization

1. Local Organization Chair of the 4th International Conference on Business Process Management (BPM'06), 5.-7. Sept. 2006, Vienna, Austria.

Program Committee Memberships

1. 2nd International Workshop on Dynamic and Declarative Business Processes (DDBP'09) in conjunction with the 13th IEEE International EDOC Conference (EDOC'09), 31. Aug. - 4. Sept. 2009, Auckland, New Zealand
2. BPM Demo Track, 7th International Conference on Business Process Management (BPM'09), 7.-10. Sept. 2009, Ulm, Germany
3. 1st International Workshop on Dynamic and Declarative Business Processes (DDBP'08) in conjunction with the 12th IEEE International EDOC Conference (EDOC'08), 15.-19. Sept. 2008, Munich, Germany
4. BPM Demo Track, 6th International Conference on Business Process Management (BPM'08), 2.-4. Sept. 2008, Milan, Italy
5. 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'08), 6.-8. Aug. 2008, Phuket, Thailand
6. IADIS International Conference on Applied Computing (AC'08), 10.-13. Apr. 2008, Algavre, Portugal
7. BPM Demo Track, 5th International Conference on Business Process Management (BPM'07), 25.-27. Sept. 2007, Brisbane, Australia

8. 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'07), 25.-27. Jul. 2007, Qingdao, China
9. IADIS International Conference on Applied Computing (AC'07), 17.-20. Feb. 2007, Salamanca, Spain
10. BPM Demo Session, 4th International Conference on Business Process Management (BPM'06), 5.-7. Sept. 2006, Vienna, Austria

Reviewer for Journals

ACM Transactions on Adaptive and Autonomous Systems (TAAS), ACM Transactions on the Web (TWEB), Data & Knowledge Engineering (Elsevier), IBM Systems Journal, IEEE Internet Computing, IEEE Systems Journal, IEEE Transactions on Services Computing, IEEE Transactions on Software Engineering, International Journal of Computers and Applications, International Journal of E-Business Research, Journal of System Architecture (Elsevier), Service Oriented Computing and Applications (Springer), Software: Practice and Experience (Wiley)

7 Teaching Activities

Courses

1. **Distributed Systems Lab**, Technical University Vienna
   Winter 2005-2009; 500 undergraduate students
2. **Technologies for Distributed Systems**, Technical University Vienna
   Summer 2007-SS 2009; 160 master students
3. **Project Lab**, **Computer Science Lab Work**, **Internet Computing Lab Work**, Technical University Vienna
   2005-2009; undergraduate and master students

Master Thesis Supervision

1. Predrag Celikovic: A Domain-Specific Language for QoS-Aware Service Composition (working title)
2. Christian Enzi: Modeling Web Service Choreographies with WS-CDL and WS-BPEL
3. Stephan Herzog: V-BSE – A Simulation Environment for WS-BPEL Processes
4. Andreas Huber: A Transformation Engine for Resolving Web Service Heterogeneities within the VRESCo Runtime
5. Thomas Laner: A Semantically Enriched Querying Language for the VRESCo Metadata Model
6. Philipp Leitner: DAIOS - Dynamic, Asynchronous and Message-oriented Invocation of Web Services

7. Daniela Malfatti (Univ. of Trento): A Meta-Model for QoS-Aware Service Compositions (assistant supervisor together with Prof. Marco Aiello)
8. Oliver Moser: A Non-Intrusive Monitoring and Adaptation Approach for WS-BPEL
9. Benjamin Mueller: Optimization of QoS-Aware Service Composition Algorithms (working title)
10. Christoph Nagl: ViDRE – A Service-Oriented Business Rule Engine Based on RuleML
11. Alexander Schindler: QoS Monitoring of Workflows within the Microsoft .NET Environment (working title)
12. Bernhard Schreder: Legacy Datasource Integration for the Semantic Web

8 Awards and Prices

- 3rd place at the Web Service Challenge 2007 (together with Lukasz Juszczyk, Anton Michlmayr, Christian Platzer, Alexander Urbanec and Schahram Dustdar), co-located with the IEEE Joint Conference on E-Commerce Technology and Enterprise Computing, E-Commerce and E-Services (CEC & EEE'07), 23. - 26. July 2007, Tokyo, Japan.
- 2nd place (out of 31 teams) at the First IEEE International Services Computing Contest, co-located with the 4th International Conference on Web Services and the 3th International Conference on Services Computing 2006, Chicago, IL, USA (together with Christoph Nagl and Schahram Dustdar).
- 2nd place at the Web Service Challenge, co-located with the 2006 IEEE Conference on E-Commerce Technology (CEC'06) and IEEE Conference on Enterprise Computing, E-Commerce and E-Services (EEE'06), San Francisco, CA, USA (together with Marco Aiello, Christian Platzer, Martin Vasko and Huy Tran and Schahram Dustdar).