DISSERTATION

Event Processing in QoS-Aware Service Runtime Environments

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

unter der Leitung von

Prof. Schahram Dustdar
Institut für Informationssysteme (E184)
Technische Universität Wien

begutachtet von

Prof. Carlo Ghezzi
Department of Electronics and Information
Politecnico di Milano, Italy

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Anton Michlmayr
Matr.Nr. 9925744
Fugbachgasse 11/25
A-1020 Wien

Wien, am 15.02.2010

Anton Michlmayr

Abstract

Service-oriented Computing (SOC) has recently received attention from both academia and industry. It introduces a new paradigm for addressing the complexity of distributed systems, by using loose coupling, platform-independent interface descriptions and well-established standards. The overall idea of service orientation is that software is provided as service, which can be found in service registries and invoked by service consumers. Web services represent the most common realization of Service-oriented Architecture (SOA) that build on the main standards SOAP, WSDL and UDDI. However, current service-oriented systems are often not as dynamic and adaptable as intended. For instance, the publish-find-bind-execute cycle of the SOA model is not always entirely realized since service registries are often missing.

In this thesis, we address some of the current challenges in SOC research and practice, such as service metadata and querying, as well as dynamic binding, invocation and mediation of services. The main focus is on event processing and asynchronous notifications in service-oriented systems. The aim is to enable clients to subscribe in order to receive notifications if certain events of interest occur. This can range from basic events (e.g., new service is published into the service registry) to more complex events regarding service invocations and Quality of Service (QoS). In contrast to existing approaches, we provide complex event processing mechanisms such as event patterns and sliding window operators.

The contribution of this thesis can be summarized as follows: Firstly, we describe the current challenges we see in SOC research and practice based on a motivating example. Secondly, we introduce the Vienna Runtime Environment for Service-oriented Computing (VRESCo) that aims at addressing some of these challenges to facilitate the development of service-oriented applications. Thirdly, we present how complex event processing principles can be integrated into service-oriented systems. Therefore, we describe the VRESCo event notification support in detail, by showing how events are processed and how clients can declare their interest to get notified when certain events occur. Fourthly, besides a detailed performance evaluation of the different VRESCo components, we give examples for the usefulness and applicability of this event mechanism. Among others, this includes notification-based rebinding, service provenance and monitoring of QoS and Service Level Agreements. Finally, we point to further application scenarios and future research directions which are enabled by this work.

Zusammenfassung

Service-oriented Computing (SOC) hat in der letzten Zeit sowohl in der Forschung als auch in der Industrie Beachtung gefunden. Es stellt ein neues Paradigma dar, um die Komplexität von verteilten Systemen zu adressieren. Dabei werden lose Kopplung, plattformunabhängige Schnittstellenbeschreibungen und etablierte Standards verwendet. Service-Orientierung beruht auf der Idee, dass Software als Dienst (*Service*) zur Verfügung gestellt wird, welches in Service Register (*Registries*) gefunden und von Klienten aufgerufen werden kann. Web Services repräsentieren die häufigste Realisierung von service-orientierter Architektur (SOA) und basieren auf den Standards SOAP, WSDL und UDDI. Gängige service-orientierte Systeme sind jedoch oft nicht so dynamisch und adaptierbar wie vorgesehen. Zum Beispiel wird der *publish-find-bind-execute* Kreislauf des SOA Modells nicht immer vollständig durchgeführt, weil Service Registries in der Praxis oft nicht vorhanden sind.

Diese Dissertation beschäftigt sich mit aktuellen Herausforderungen in der SOC Forschung und Praxis. Dazu zählen zum Beispiel Service Metadaten und Abfragen, sowie dynamische Bindung, Aufrufe und Mediation von Services. Der Hauptfokus liegt auf der Verarbeitung von Ereignissen (*Events*) und asynchronen Benachrichtigungen (*Notifications*) in service-orientierten Systemen. Das Ziel ist Klienten zu ermöglichen sich zu subskribieren, um Benachrichtigungen über bestimmte Ereignisse zu erhalten. Das reicht von einfachen Ereignissen (z.B., ein neues Service wird zur Registry hinzugefügt) bis zu komplexeren Ereignissen bezüglich Serviceaufrufe und Dienstgüte (*Quality of Service*, kurz *QoS*). Im Gegensatz zu existierenden Ansätzen stellen wir Mechanismen zur komplexen Eventverarbeitung zur Verfügung, wie zum Beispiel Ereignismuster (*Event Patterns*) und so genannte verschiebbare Fenster (*Sliding Windows*).

Der Beitrag dieser Arbeit kann wie folgt zusammengefasst werden: Zunächst wird anhand eines motivierenden Beispiels beschrieben, welche aktuellen Herausforderungen wir in der SOC Forschung und Praxis sehen. Danach stellen wir die Vienna Runtime Environment for Service-oriented Computing (VRESCo) vor, mit dem Ziel einige dieser Herausforderungen zu adressieren und das Entwickeln von service-orientierten Anwendungen zu vereinfachen. Darauf aufbauend wird erläutert, wie Prinzipien der komplexen Eventverarbeitung in service-orientierte Systeme integriert werden können. Dazu beschreiben wir den Event-Mechanismus von VRESCo im Detail. Wir zeigen die Verarbeitung von Ereignissen und wie Klienten ihr Interesse bekunden, um über das Auftreten von bestimmten Ereignissen benachrichtigt zu werden. Zusätzlich zur Leistungsevaluierung der verschiedenen VRESCo Komponenten werden Beispiele für die Nützlichkeit und Anwendung dieses Event-Mechanismus beschrieben. Unter anderem beinhaltet das ereignis-basiertes Binden, Service Provenienz und Überwachung von QoS und Service Level Agreements. Abschließend zeigen wir weitere Anwendungsszenarien und zukünftige Forschungsrichtungen, die durch diese Arbeit ermöglicht werden.

Acknowledgements

Doing a PhD is never the effort of a single person, but many people have contributed to make this possible. First and foremost, I am grateful to my advisor Prof. Schahram Dustdar for his support. He provided an open and stimulating environment to conduct research and helped me to put things on the right track. I am also very thankful to my examiner Prof. Carlo Ghezzi for interesting discussions and feedback that helped me to improve my thesis.

I want to thank my colleagues at the Distributed Systems Group who accompanied me during the last years, especially the VRESCo core members Philipp Leitner and Florian Rosenberg. I really enjoyed the great time we have spent exchanging views and turning our ideas into prototypes and papers. I am thankful to all students that contributed to VRESCo, especially Andreas Huber, Thomas Laner and Christian Marek. Besides that, I thank Benjamin Müller and all other teaching assistants that helped us to manage the teaching workload. Finally, I am also grateful to our secretaries for keeping most of the administrative tasks away from us.

I am very thankful to my friends who sustained my work-life balance and provided support when I needed it most. Among others, Gerd and Stefan for countless sports events and other fun activities. Bernhard, Markus and Wilfried for sharing the experience of pursuing a PhD. The core members of the volleyball team "Ninjas of Nasty" (Christine, Hannes, Jassi, Koarl, Laurent, Romy) for keeping me in shape and showing team spirit. The buddies from Steyr for long lasting friendship during all these years: Bertram, Dieter, Goggi, Herwig, Metzi, Michi. And many others.

Most of all, however, I want to thank my parents Christine and Leopold, and my sisters Birgit and Elke – for everything!

Anton Michlmayr
Vienna, February 2010

Publications

Parts of the work presented in this thesis have been published in scientific journals, books, conferences and workshops, which are listed in reverse chronological order below. This thesis partly contains verbatim text from these publications that are no longer cited throughout the thesis. The full list of publications by the author can be found in the references.

- Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. *IEEE Transactions on Services Computing (TSC)*, IEEE Computer Society. (forthcoming)
- Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. Selective Service Provenance in the VRESCo Runtime. *International Journal on Web Services Research (JWSR)*, IGI Global. (forthcoming)
- Michlmayr, A., Leitner, P., Rosenberg, F., and Dustdar, S. Event Processing in Web Service Runtime Environments. In *Principles and Applications of Distributed Event-based Systems*, Editors: A. Hinze and A. Buchmann, IGI Global. (forthcoming)
- Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. Comprehensive QoS Monitoring of Web Services and Event-Based SLA Violation Detection. In *Proceedings of the Fourth International Workshop on Middleware for Service Oriented Computing (MW4SOC'09), co-located with the 10th International Middleware Conference (Middleware'09)*, pp. 1–6, ACM, November 2009. DOI: 10.1145/1657755.1657756
- Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. Service Provenance in QoS-Aware Web Service Runtimes. In *Proceedings of the International Conference on Web Services (ICWS'09)*, pp. 115–122, IEEE Computer Society, July 2009. DOI: 10.1109/ICWS.2009.32
- Rosenberg, F., Leitner, P., Michlmayr, A., and Dustdar, S. Integrated Metadata Support for Web Service Runtimes. In *Proceedings of the Middleware for Web Services Workshop (MWS'08), co-located with the 12th International Enterprise Distributed Object Computing Conference (EDOC'08)*, pp. 361–368, IEEE Computer Society, September 2008. DOI: 10.1109/EDOCW.2008.38
- Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. Advanced Event Processing and Notifications in Service Runtime Environments. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)*, pp. 115–125, ACM, July 2008. DOI: 10.1145/1385989.1386004

- Michlmayr, A., Leitner, P., Rosenberg, F., and Dustdar, S. Publish/Subscribe in the VRESCo SOA Runtime (demo paper). In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)*, pp. 317–320, ACM, July 2008. DOI: 10.1145/1385989.1386031
- Leitner, P., Michlmayr, A., Rosenberg, F., and Dustdar, S. End-to-End Versioning Support for Web Services. In *Proceedings of the International Conference on Services Computing (SCC'2008)*, pp. 59–66, IEEE Computer Society, July 2008. DOI: 10.1109/SCC.2008.21
- Michlmayr, A., Rosenberg, F., Platzer, C., Treiber, M., and Dustdar, S. Towards Recovering the Broken SOA Trianlge – A Software Engineering Perspective. In *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering (IW-SOSWE'07), co-located 6th ESEC/FSE Joint Meeting (ESEC/FSE'07)*, pp. 22–28, ACM, September 2007. DOI: 10.1145/1294928.1294934

Contents

List of Figures

List of Tables

Listings

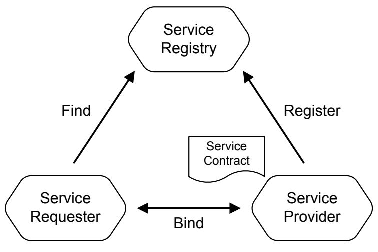
Chapter 1.

Introduction

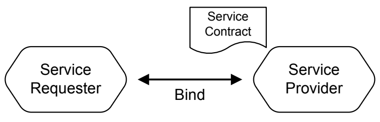
During the last few years, Service-oriented Architecture (SOA) [44] has gained acceptance as an architectural style for addressing the complexity of distributed applications by using loose coupling, platform-independent interface descriptions and well-established standards. In theory, the basic SOA model consists of three participants that communicate as shown in Figure 1.1a. *Service providers* implement services and make their descriptions available in *service registries*. *Service consumers* (also called *service requesters*) can query service descriptions and location information from the registry, bind to the corresponding service provider and finally execute the service. Due to platform-independent interface descriptions, SOA enables flexible applications with respect to manageability and adaptivity. For instance, services can easily be exchanged at runtime and service consumers can switch between alternative service providers seamlessly. In this regard, the term Service-oriented Computing (SOC) [144] has been used to describe the paradigm and corresponding research discipline, which proposes the use of SOA for building service-based and distributed applications.

Web services [1] represent the most common realization of SOA, building on the main standards SOAP [202] for messaging-based communication, WSDL [203] for service interface descriptions and UDDI [136] for service registries. Furthermore, WS-BPEL [140] can be used for composing multiple services to achieve higher-level functionality. Over time, several specifications have been introduced to address other issues in Web services such as policies and security [190].

However, practice has shown that SOA solutions are often not as flexible and adaptable as claimed. We argue that there are some issues in current implementations of the SOA model. First and foremost, service registries such as UDDI and ebXML [134] did not succeed as intended. We think this is partly due to their limited querying support using keyword-based matching of registry content, and insufficient support for metadata and non-functional properties of services. This is also highlighted by the fact that Microsoft, SAP and IBM have finally shut down their public UDDI registries in 2006. As a result, service registries are often missing, leading to point-to-point solutions where service endpoints are exchanged at design-time (e.g., using email/phone) and service consumers statically bind to these endpoints (see Figure 1.1b). Besides missing registries, other SOC challenges are presented in this chapter, while the main focus of this thesis is how event processing can be leveraged in service-oriented systems.



(a) SOA Model



(b) SOA Practice

Figure 1.1.: Basic SOA Model – Theory vs. Practice

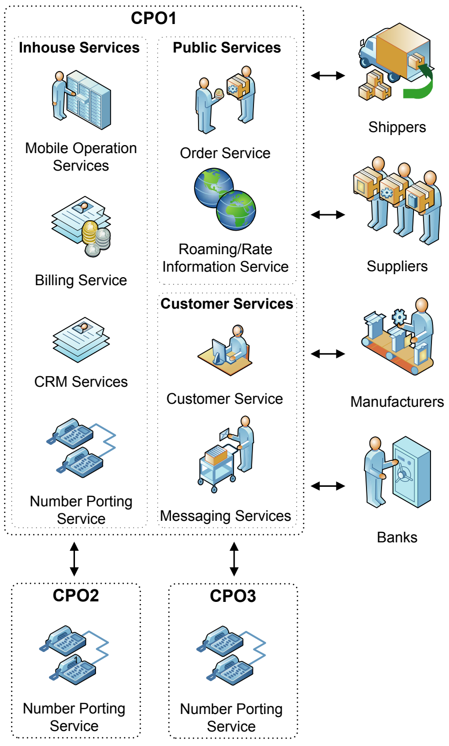
1.1. Motivating Example

Before describing the research challenges addressed in this thesis we want to introduce a simple motivating example. This example is used throughout the thesis for illustration purposes since it highlights several challenges that are common in service-centric software engineering.

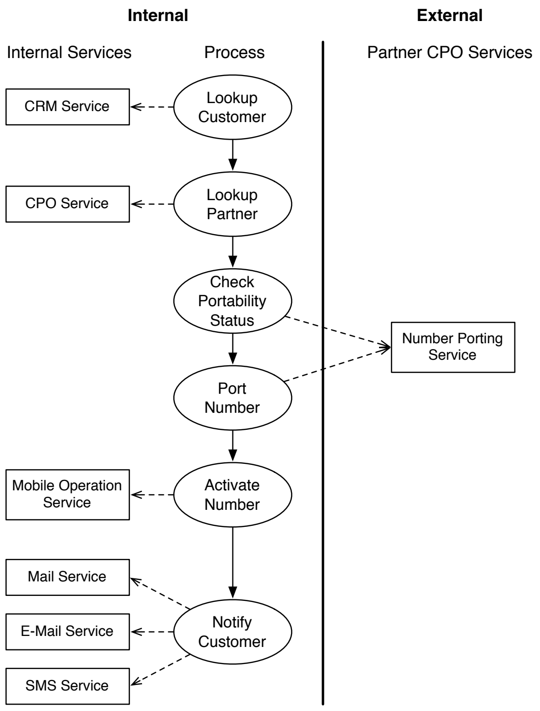
Figure 1.2 shows a typical enterprise application scenario from the telecommunications domain. The overview of this case study is depicted in Figure 1.2a where cell phone operator CPO1 provides different kinds of services: *Public Services* (e.g., Rate Information Service) can be used by everyone. *Customer Services* (e.g., Short Messaging Service) are used by customers of CPO1. Finally, *Inhouse Services* (e.g., CRM Service) represent internal services which should only be accessed by the different departments of CPO1. Besides that, CPO1 also consumes services from its partners (e.g., cell phone manufacturers and suppliers) and competitors (e.g., CPO2).

Figure 1.2b shows a simplified version of the number porting process (depicted as oval boxes). In Europe, CPOs have to provide number porting by law, in order to enable consumers to keep their mobile phone number when switching to another CPO. This process is interesting because it contains both internal and external services (depicted as rectangles) and multiple service candidates. After the customer has been looked up using the Customer Service, the Number Porting Service of the old CPO has to be invoked. If the number is portable the porting is executed by the old CPO. If this step was successful the new CPO is informed, which activates the new number using the Mobile Operation Service. Finally, a notification is sent to the customer using the preferred notification mechanism (e.g., SMS, email, etc.).

Please note that this scenario is dynamic and the process should ideally adapt to changes (e.g., new CPOs enter the market, number porting services of existing CPOs are updated, etc.). However, currently service consumers are not aware of such changes. In this regard, we aim at using event notifications to inform interested subscribers. Furthermore, when alternative services are available, service selection mechanisms can be applied to dynamically chose the best service. This decision may be based on the past performance and history of the service. Finally, for external services the Quality of Service (QoS) may be defined and monitored.



(a) Case Study Overview



(b) Number Porting Process

Figure 1.2.: CPO Case Study

1.2. SOC Research Challenges

Adaptive service-oriented systems have several requirements that lead to a number of research challenges. In this section, we summarize the current challenges we see most important. In the remainder of this thesis, we present an approach that was designed to address them.

- **Service Metadata:** Service interface description languages such as WSDL focus on the interface which is needed to invoke the service. However, from this interface it is often not clear what a service actually does, and if it performs the same task as another service. Service metadata [23] can give additional information about the purpose of a service and its interface (e.g., pre- and post-conditions). For instance, in the CPO case study without service metadata it is not clear if the number porting services of CPO2 and CPO3 actually perform the same task. We further distinguish between structured and unstructured metadata: Structured metadata allows to attach data according to the predefined service metadata model, while unstructured metadata enable service providers to attach unstructured information (e.g., tags) to services.

- **Service Querying:** Once services and associated metadata are defined, this information should be discovered and queried by service consumers. This is the focus of service registry standards such as UDDI [136] and ebXML [134]. In practice, the service registry is often missing since there are no public registries and service providers often do not want to maintain their own registry [119]. Besides service discovery, another issue is how to select a service from a pool of possible service candidates [210]. Service selection using querying languages or APIs can be either type-safe or not type-safe, depending on whether the query service returns specific types from the service metadata model.
- **Quality of Service:** In enterprise scenarios QoS plays a crucial role [212]. This includes both network-level attributes (e.g., latency, availability, etc.), and application-level attributes (e.g., response time, throughput, etc.). The QoS model should ideally be extensible to allow service providers to adapt it for their needs. Furthermore, the QoS must be monitored accordingly so that users can be notified when the measured values do not adhere to the Service Level Agreement (SLA). For instance, CPO1 may want to be aware of the current QoS of partner services when integrating them into its own business processes. If SLAs are violated the partners may have to pay penalties.
- **Dynamic Binding and Invocation:** One of the main advantages of service-centric systems has always been the claim that service consumers can dynamically bind and invoke services from a pool of candidate services (e.g., depending on the current QoS). However, in current practice this is only possible if the service interfaces are identical, which is often not the case especially when switching from one service provider to another. This raises the need for service mediation approaches that mediate between alternative services depending on the service metadata and mappings stored in the registry. Considering the CPO case study, the interfaces of CPO2's and CPO3's number porting services may differ, but the number porting process of CPO1 should still be able to seamlessly switch between them at runtime.
- **Event Notifications:** Service-centric systems are said to be flexible and dynamic. To support this flexibility, event processing mechanisms can be used to record which events occur within the system. This includes both basic service events (e.g., new service is published by CPO1) and complex events regarding QoS (e.g., average response time of CPO2's service X has changed) and invocations (e.g., CPO3's service Y has been invoked). However, most current approaches do not support mechanisms for complex event processing [102], such as event patterns or sliding window operators. Users can usually subscribe only to basic events and get asynchronously notified per email or Web service notifications (e.g., WS-Notification [138], WS-Eventing [199]). With appropriate runtime support, such notifications may trigger adaptive behavior (e.g., rebinding to other services, host new service instances, etc.). Since event processing represents the main contribution of this work, the challenges and solutions for complex event processing in SOC are discussed in more detail throughout the thesis.

- **Service Versioning:** Like any piece of software, services are subject to permanent change regarding their interfaces and implementations. For instance, in our CPO case study, CPO1 may provide several versions of the Number Porting Service in parallel (e.g., using different Web service technologies such as JAX-RPC and Microsoft WCF). Current registry standards provide limited support for versioning of registry data but cannot handle the differences between various service revisions. We thereby distinguish between metadata versioning (i.e., maintain versions of metadata), and end-to-end versioning support (i.e., enable service consumers to switch between different service revisions transparently).
- **Access Control:** When service information and metadata are stored in a registry, security issues come into play (especially in business scenarios). Clearly, service information should only be accessible for specific users which raises the need for appropriate authentication and authorization mechanisms. Such mechanisms are described in various standards such as WS-Security [139]. Considering our CPO case study, CPO1 might want to define that the Number Porting Service is only visible for competitors CPO2 and CPO3, but not for customers or public users.

1.3. Contributions

In this section, we highlight the research contributions of this thesis, which can be summarized in the following three questions that have driven our research efforts.

- **Q1:** *What are current challenges of flexible and adaptive SOC infrastructures in general, and service registries in particular? How can these challenges be addressed within a coherent system?*

  Section 1.2 introduces the current SOC challenges we have identified based on a motivating example. The aim of this thesis is to introduce a novel service runtime environment that addresses these challenges and provides appropriate runtime support.
- **Q2:** *How can event processing principles be seamlessly integrated in SOC infrastructures? What are specific challenges and approaches?*

  Event notifications represent one of these challenges in service-oriented systems. The main objective of this thesis is to address this specific challenge in more detail (besides addressing the other challenges). Therefore, the service runtime environment has been enhanced with advanced event notification support.
- **Q3:** *Which application scenarios are enabled by this work? What would be more difficult or less efficient to realize without event processing support?*

  The focus of this thesis has been put on event processing, since we argue that some challenges can be better solved using this mechanism. Consequently, we give examples that highlight the usefulness and applicability of the presented event processing support.

The research contributions of this thesis are organized in three main layers that correspond to the research questions raised above (see Figure 1.3). In the following, we will briefly discuss these layers bottom-up, while the remainder of the thesis describes each layer in more detail.

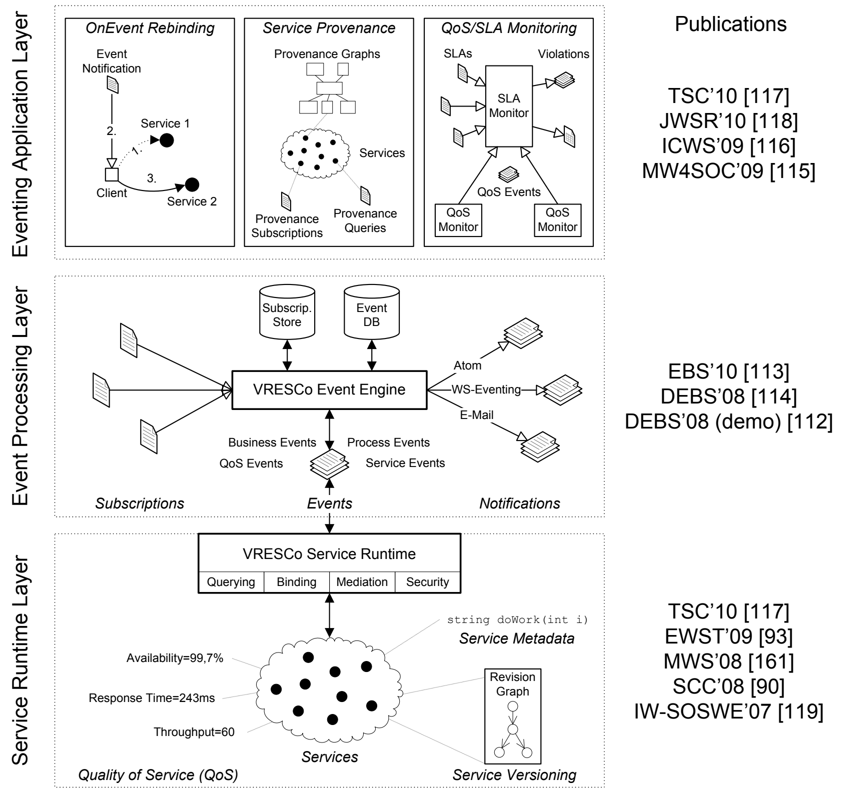


Figure 1.3.: Contributions

- **Service Runtime Layer:** The bottom layer aims at addressing the SOC challenges introduced in Section 1.2. This includes all challenges except for event notifications which represents the focus of the middle layer. We argue that the SOA triangle is broken since registry standards (e.g., UDDI and ebXML) did not succeed. Therefore, registries are often missing in current service-oriented systems. In this regard, we suggest that registries should be equipped with additional functionality and integrated with service runtime environments. More precisely, we find it crucial that such environments provide support for service metadata including versioning and QoS, service querying, access control, as well as dynamic binding, invocation and mediation of services. In this layer, we present the Vienna Runtime Environment for Service-oriented Computing (VRESCo) that addresses these issues. In contrast to the other two layers, the core features of VRESCo have been designed as a joint effort together with Florian Rosenberg [158] and Philipp Leitner.

- **Event Processing Layer:** The middle layer describes the Event Notification Engine, that is built on top of the VRESCo runtime environment. The general idea is that clients can subscribe to get notified asynchronously when events of interest occur. In contrast to basic notifications provided by existing approaches, we aim at complex event processing that includes events patterns and sliding window operators. Besides the actual matching of subscriptions to events and mechanisms to notify subscribers, this raises the issues of event ranking, correlation and persistence. In this thesis, we show how we have addressed these issues in VRESCo and which event types may be supported in service-centric systems. Furthermore, access control to events is of particular importance: Firstly, events should only be published by authorized parties. Secondly, and this is even more important, events should not be visible for all users. This requires fine-grained access control mechanisms, which are realized by introducing different event visibilities.
- **Eventing Application Layer:** The top layer builds on the Event Processing Layer and provides applications and usage scenarios that leverage event notifications in service runtime environments. We focus on three applications that have been implemented within the scope of this thesis: Firstly, events can be used to trigger adaptive behavior (e.g., if functionally equal services with better QoS get available, the service proxy should be notified to transparently rebind to them). Secondly, besides notifying subscribers events are also stored in the Event Database. The information stored in service events represents the provenance (i.e., the origin and history) of the corresponding service. We provide an approach that aims at visualizing this information in so called provenance graphs and enables clients to query and subscribe provenance information. Thirdly, QoS represents an important issue especially in business scenarios. We have integrated mechanisms for monitoring current QoS properties which are then published as QoS events. These events can be used to detect if the current QoS violates pre-defined SLAs. Besides this, application scenarios of our ongoing and future work are also briefly mentioned.

1.4. Organization of the Thesis

This thesis consists of three main parts which are aligned to the three layers described above. Before describing these layers in more detail, Chapter 2 briefly reviews the relevant state of the art. This includes standards, tools and specifications concerning event processing and Web services, as well as the combination of both paradigms. Related work about the different aspects covered by our approach is reviewed in dedicated sections in the three chapters.

Chapter 3 presents the VRESCo runtime environment in detail. This includes the overall architecture, service metadata and QoS model, service versioning, service querying and access control, as well as the service mediation approach. Finally, the evaluation of this chapter shows the performance of the different components and an end-to-end evaluation.

Chapter 4 describes the VRESCo event notification support. This includes the overall architecture, the different event types, as well as subscription and notification mechanisms. Furthermore, it also describes how events are persisted in the Event Database and how event access control is provided using event visibility. The evaluation of this chapter gives concrete subscription examples to highlight the expressiveness of the subscription language and depicts the performance and overhead of the Event Engine.

Chapter 5 presents the third part of the thesis, which includes the different application scenarios for event notifications in service-oriented systems. Among others, this includes notification-based rebinding, service provenance and event-based QoS/SLA monitoring. Furthermore, we describe other scenarios that have been left for future work.

Finally, Chapter 6 concludes this thesis by summarizing the research contributions. Therefore, we show how the research questions introduced in this chapter have been addressed. Furthermore, we briefly point to future research directions that are enabled by this work.

Chapter 2.

Review of the State of the Art

This chapter reviews the state of the art regarding topics that are important for this thesis. First of all, Section 2.1 introduces the Publish/Subscribe paradigm and event-based systems. Section 2.2 then briefly presents an event processing engine that is used in our work. Section 2.3 gives a brief overview of important Web service specifications, while Section 2.4 compares different Web service registry and repository approaches regarding the challenges introduced in the previous chapter. Section 2.5 presents the event notification support of current Web service registries in more detail, while Section 2.6 finally introduces the most important specifications and standards regarding event notifications in service-oriented systems.

Contents

2.1. Publish/Subscribe and Event Processing

In traditional object-oriented programming, interaction between objects is done point-to-point by invoking methods which are exposed using interfaces. For instance, object O_1 invokes method M_2 defined by interface I_2, which is implemented by object O_2. This invocation may change the internal state of object O_2. The Observer Pattern [57] introduces more dynamism by defining one-to-many dependencies between objects. More precisely, several observer objects can be notified if the state of one subject object changes.

The Observer Pattern can be seen as simplified version or ancestor of the Publish/Subscribe paradigm [47]. According to this, the basic interaction scheme of Publish/Subscribe (often referred to as Pub/Sub or P/S) is illustrated in Figure 2.1. *Subscribers* are enabled to express their interest in *events* (or event patterns), that are produced by *publishers*. If such events (or event patterns) occur, *notifications* are sent to interested subscribers. The *Event Notification Service* is responsible for managing subscriptions and efficient delivery of notifications.

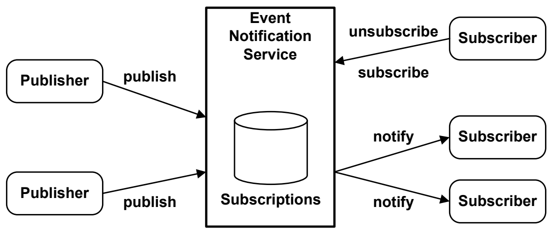


Figure 2.1.: Publish/Subscribe Interaction Scheme

In contrast to the Observer Pattern where publishers and subscribers interact directly, the Event Notification Service adds three dimensions of *decoupling* to Publish/Subscribe: *space*, *time* and *synchronization*. Space decoupling means that the interacting parties are independent of each other, in the sense that publishers do not know their subscribers and vice versa. Time decoupling refers to the fact that publishers and subscribers do not have to be online at the same time. Finally, synchronization decoupling means that publishers are not blocked when publishing events and subscribers can get notified asynchronously.

According to [47], there are mainly three different types of Publish/Subscribe systems. In topic-based Publish/Subscribe, events are published to certain topics, while content-based Publish/Subscribe enables to subscribe to the actual content of events using expressive subscription languages. Finally, type-based Publish/Subscribe [46] provides type-safety by taking the actual type of the event into consideration. Furthermore, different architectures can be used for implementing the Event Notification Service such as hierarchical client-server, peer-to-peer (P2P), or hybrid architectures [26]. The notion of distributed event-based systems [124] is used if publishers, subscribers and the Event Notification Service are distributed.

The loosely coupled nature of the Publish/Subscribe paradigm and event-based systems has several inherent research challenges that have been addressed in the past years. For instance, how applications that follow this paradigm can be modeled and specified [41, 50, 103]. These approaches are important to be able to define the behavior of such systems. Furthermore, such specifications are crucial for verification and validation. There are some approaches that aim at verifying event-based applications using model checking techniques (e.g., [12,13,60]). Prior to the present thesis, we have focused on how such applications can be validated using testing methods [110,111], while others try to bridge the gap between verifying and testing event-based applications [213]. Finally, another interesting research topic represents how transactions can be provided in such loosely coupled environments, which was addressed by several research efforts [97, 98, 109, 169, 180, 186].

The increasing interest in the Publish/Subscribe paradigm and event-based systems has led to several research prototypes, such as Siena [26], JEDI [35], Hermes [148], PADRES [51], STREAM [11], Gryphon [20], Elvin [168] and Narada Brokering [143] (just to name a few). An overview of QoS-aware event-dissemination middleware can be found in [104]. Furthermore, Publish/Subscribe is used in several specifications and standards (e.g., CORBA Event and Notification Service [127, 128], Jini [178], JMS [177], DDS [129], etc.), commercial tools (e.g., TIBCO Rendezvous [183], IBM WebSphere MQ [76], Oracle Fusion [132], etc.) and open source projects such as Esper [45]. In recent years, the term Complex Event Processing (CEP) [102] has emerged to describe how meaningful events and event patterns can be identified when processing and correlating large numbers of events.

2.2. Esper

To give a concrete example for such event processing engine, this section briefly introduces the open source engine Esper [45], which is later used as foundation for the VRESCo Event Notification Engine.

Esper supports several ways for representing events. Firstly, any Java/C# object may be used as an event as long as it provides getter methods to access the event properties. Event objects should be immutable since events represent state changes that occurred in the past and should not be changed. Secondly, events can be represented by Java objects that implement the interface `java.util.Map`. The event properties are those values that can be obtained using the map getter. Finally, events may be instances of the class `org.w3c.dom.Node` representing XML events. In that case, XPath [192] expressions are used as event properties. Additionally, Esper provides different types of properties that can be obtained from events. *Simple* properties represent simple values (e.g., *timestamp*). *Indexed* properties are ordered collections of values (e.g., *user[4]*) whereas *mapped* properties represent keyed collections of values (e.g., *user['firstname']*). Finally, *nested* properties live within another property of an event (e.g., *service.QoS*).

In Esper, subscriptions are done by attaching listeners to the engine, where each listener contains a query that defines the actual subscription. Listeners implement an interface which is invoked when the subscription matches incoming events. Queries use the Esper Event Processing Language (EPL) – formerly known as Esper Query Language (EQL) – which is similar to the Structured Query Language (SQL). The main difference is that EPL is formulated on event streams whereas SQL addresses database tables. The `select` clause specifies the event properties to retrieve, the `from` clause defines the event streams, and the `where` clause specifies constraints. Furthermore, similar to SQL there are aggregate functions (e.g., `sum`, `avg`, etc.), grouping functions (`group by`), and ordering structures (`order by`). Multiple event streams can be merged using the `insert` clause, or combined using joins. In addition to that, event streams can be joined with relational data using SQL statements on database connections.

EPL provides a powerful mechanism to integrate temporal relations of events using *sliding window operators*. These operators allow to define queries for a given period of time. For instance, if QoS events regularly publish the QoS values of services, then subscriptions can be defined on the average response time of a given service during the last 6 hours. Finally, EPL supports subqueries, output frequency and event patterns. The last allows to define relations between subsequent events (e.g., -> representing a 'followed by' relation). Section 4.10 gives some examples for EPL queries. More information on Esper and EPL can be found in [45].

2.3. SOC and Web Services

SOC [144] and Web services [1,190] have received increasing attention in the last years. Besides the most important standards SOAP [202], WSDL [203] and UDDI [136] that form the basic SOA triangle, a variety of additional Web service specifications has been proposed. One of the main ideas of SOC is to compose multiple services into business processes (often referred to as orchestrations) to achieve higher-level functionality. This is the focus of the Business Process Execution Language (BPEL) [140]. On a higher-level of abstraction, cross-organizational workflows (often referred to as choreographies) can be modeled using the Choreography Description Language (WS-CDL) [197]. In addition, there are some research efforts to bridge the gap between choreographies and orchestrations [39,108,162].

In this regard, Service Level Agreements (SLAs) can be defined between business partners to specify the obligations that services have to fulfill. For instance, provider P can negotiate with consumer C that the response time of service S is below 1 second. If this SLA is violated, P is obliged to pay a certain penalty to C. The WSLA framework [71] provides such agreements.

Over time a rich Web services stack has been built that provides specifications for additional aspects, such as policies (WS-Policy [198]), service metadata (WS-MetadataExchange [74]), security (WS-Security [139], WS-ReliableMessaging [133]), and transactions (WS-Transaction [141]). Additionally, several design patterns have been identified that are recurring in SOAs [211].

2.4. Service Registries and Repositories

As already stated, the idea of public service registries did not succeed and most of them have been shut down. However, registries are still an important component that is currently mostly used within organizations. There are several approaches and standards for service registries (often referred to as service repositories). Concrete examples are UDDI [136], ebXML [134], WSO2 ESB and Registry [205], Mule ESB and service repository [125], IBM WebSphere ESB and Service Registry and Repository [77], AWSR [184] and XMethods [207], just to name a few (some of them are integrated in Enterprise Service Bus solutions [27]). We have compared several approaches with the VRESCo runtime in Table 2.1, considering a carefully selected range of established standards, mature open-source frameworks and commercial tools.

Challenge		**UDDI**	**ebXML**	**Mule**	**WSO2**	**WebSphere**	**VRESCo**
Service Metadata	Unstructured	+	+	+	+	+	~
	Structured	~	~	~	~	+	+
Service Querying	Query Language/API	+	+	+	~	+	+
	Type-safe Query	–	–	–	–	~	+
Quality of Service	Explicit QoS Support	–	–	–	~	~	+
	QoS Monitoring	–	–	–	–	–	+
Dynamic Service Invocation	Binding & Invocation	–	–	+	–	~	+
	Service Mediation	–	–	+	+	+	+
Service Versioning	Metadata Versioning	–	+	+	~	~	–
	End-to-End Support	–	–	–	–	–	+
Event Processing	Basic Notifications	+	+	+	~	+	+
	Complex Event Processing	–	–	–	–	~	+

Table 2.1.: Related Enterprise Registry Approaches

Our findings are organized using the challenges introduced in Chapter 1. In general, all systems allow to store metadata about services. Mostly, this is done in an unstructured way (e.g., using tModels in UDDI). There is only limited support for structured metadata in most approaches, while VRESCo and WebSphere provide an extensive structured metadata model. To access data and metadata within the registry a query language or API is needed, which is provided by all approaches (WSO2 supports querying only based on Atom [166] resources). In contrast to VRESCo, type-safe queries are not supported by most approaches since querying is usually done on the unstructured service metadata model using languages such as SQL. Only WebSphere provides partial support by using XPath expressions for querying.

Currently, explicit support for QoS is not widely available. It is to some extent possible in WSO2 and WebSphere, and fully supported by VRESCo. WSO2 supports QoS only in terms of WS-Security [139] and WS-ReliableMessaging [133]. However, none of these frameworks except VRESCo has integrated QoS monitoring. Integration of dynamic binding, invocation and mediation of services is obviously not supported by pure registries such as UDDI or the ebXML registry. The other systems provide support in this respect due to their integrated ESBs (e.g., Mule provides an ESB and therefore supports dynamic binding and invocation).

All systems except UDDI and VRESCo allow to maintain multiple versions of metadata in the registry. However, only VRESCo provides end-to-end versioning support, which enables to seamlessly rebind and invoke different service revisions at runtime. Finally, all approaches provide support for basic notifications using email, Web service notifications or news feeds (e.g., Atom [166], RSS [165], etc.). Only WebSphere and VRESCo allow clients to subscribe to more complex events and event patterns using a rich subscription language.

2.5. Event Notifications in Web Service Registries

The core challenge addressed in this thesis is how event processing can be supported in current Web service runtime environments. The motivation of this work was the limited support for notifications in current Web service registry standards, which is summarized in more detail in this section.

- **Universal Description, Discovery and Integration (UDDI):** The UDDI Subscription API was introduced in UDDI v3 [136] and enables monitoring of activities in a registry. This is done by tracking new, changed, and deleted entries for the following UDDI entries: *businessEntity*, *businessService*, *bindingTemplate*, *tModel*, related *businessEntity*, and *publisherAssertion*. It should be noted that the Subscription API is optional and may be implemented entirely at the discretion of a node. To the best of our knowledge, however, most UDDI implementations do not support the Subscription API. There are two ways how subscriptions are provided in the UDDI specification:
  1. Asynchronous notifications: Subscribers are asynchronously notified by the node when registry data of interest changes. This is done by either registering a Web service listener (the Web service must be implemented by the subscriber) or by using emails as notification mechanism.
  2. Synchronous Change Tracking: Subscribers inquire registry data of interest that matches their subscription preferences by using synchronous requests.

  The subscription criteria are defined using the Inquiry API (i.e., `getXXX` and `findXXX`). Furthermore, nodes may restrict which Inquiry APIs are supported. Finally, subscriptions have a specific duration that defines how long the subscription is valid.
- **Electronic Business using Extensible Markup Languages (ebXML):** Similar to UDDI, ebXML [134] also provides two ways how subscriptions are supported:
  1. Asynchronous push-style notifications: Subscribers are asynchronously notified when registry data of interest changes. Subscribers can either receive emails, or they register a Web service listener which is invoked to deliver notifications.
  2. Pull-style event retrieval: Clients can also retrieve any pending events for their subscriptions by using the *AdHocQuery* protocol.

The event model of ebXML provides auditable events representing a long-term record of changes in a *RegistryObject* (e.g., *Created*, *Deleted*, *Updated*, *Versioned*, etc.). Among other attributes, subscriptions contain an action (i.e., what happens if a subscription matches to events) and a selector (i.e., pre-defined query that defines the user's events of interest, by using SQL as default). The subscriber can express if notifications should contain either registry objects or only object references.

The event notification mechanism in ebXML seems more mature than the UDDI Subscription API. Both specifications support the same notification mechanisms: emails or Web service notifications (however, not following the standards as described below). The *AuditableEvents* in ebXML provide more fine-grained control for tracking the changes of registry data. However, both approaches do not support event patterns or complex event processing, and can notify only about changes in the registry data.

2.6. Web Service Notification Specifications

There are several research efforts that address the combination of SOA and Publish/Subscribe [34]. The authors mainly focus on content-based Publish/Subscribe, and refer to this as content-based routing (CBR). In general, there are two main approaches for such asynchronous, event-based interactions using Web services, which are briefly introduced below together with other related specifications.

- **Web Services Eventing (WS-Eventing):** The WS-Eventing specification [199] represents a lightweight realization of Publish/Subscribe for Web services. Basically, *subscribers* define interest in events as *subscriptions* and send them to the *event source*, which finally delivers *notification messages* to the specified *event sinks* when events match to subscriptions. Furthermore, the event source may designate a *subscription manager* that handles subscriptions. All entities involved in this interaction are implemented as Web services. WS-Eventing provides different delivery modes (e.g., push vs. pull), while filters on notification messages (e.g., only events for a certain topic) can be defined using XPath [192].

  The WS-Eventing specification has been implemented by several frameworks such as JBossWS [156] and Sun Wiseman [179], and open source projects such as WS-Eventing for WCF [85] and Apache Savan/C [5] (just to name a few).

- **Web Services Notification (WS-Notification):** WS-Notification [138] represents another effort for addressing Publish/Subscribe for Web services. The basic concepts are similar to WS-Eventing, even though the terminology is slightly different. In addition to the features provided by WS-Eventing, WS-Notification addresses two further mechanisms which are given in two sub-specifications. The first addresses the organization of event topics in more detail, while the second describes how notification brokers can be used to decouple notification producers from consumers.

WS-Notification has been implemented by several frameworks. To give some examples, this includes Apache Servicemix [9], Apache Muse [4], Globus Toolkit [61] and IBM WebSphere Application Server [75].

- **Reconciling Efforts:** Basically, both of these specifications provide the same features and have similar architectures, even though WS-Eventing provides only a subset of WS-Notification [187]. The WS-Messenger project [69] bridges the gap between these two approaches by supporting both specifications and providing mediation between them.

  Furthermore, there have been some efforts to reconcile these specifications (together with functionality concerning Web service management and resources) into a new specification named WS-EventNotification [120]. However, it seems that these efforts have been abandoned. As a result, WS-Eventing and WS-Notification are still competing and it is unclear at this point which specification will finally prevail.

- **SCA Extension for Event Processing and PubSub (SCA PubSub):** The Service Component Architecture (SCA) [130] defines a set of specifications describing a programming model for service-oriented systems, which is independent of the implementation technology (e.g., BPEL, Java, C++, etc.). The basic idea is that existing or new *services* are assembled as *components* to serve particular business functions. These functions are provided as services to other components, which use *references* to express that they depend on these services. Finally, applications are built as *composites*, which consist of services, components, references, *property declarations* and the *wiring* between the different elements.

  The SCA specifications have been enriched with an extension for event processing and Publish/Subscribe [131]. Therefore, every component can act as *producer* or *consumers* of events. Furthermore, *channels* are used as logical intermediary to decouple producers and consumers. In this regard, *scopes* are used to define if producers, consumers and events are visible outside of a composite. Finally, *filters* can be defined on channels or consumers to specify which subset of events are of interest for the consumer. This is done using different filter dialects such as XPath.

  Conceptually, the SCA PubSub specification provides similar means and mechanisms as the Web service notification specifications. However, it is targeted to SCA and does also rely on XPath as filter dialect by default. Therefore, there is no support for subscribing to complex events and event patterns. A prototype implementation of SCA PubSub was done as part of the Apache Tuscany project [10].

- **Web Services Distributed Management (WSDM):** Finally, the specifications Common Event Infrastructure (CEI) [73] and Common Base Event (CBE) [72] have been mainly driven by IBM's autonomic computing efforts [83]. CEI and CBE can be seen as predecessors of the Web Services Distributed Management (WSDM) specification [137]. A Java-based open source implementation of WSDM is provided by the Apache Muse project [4].

The basic idea of WSDM is to address distributed management of resources using Web services. This includes managing Web-enabled resources such as routers or printers, as well as managing Web services themselves. WSDM allows to subscribe to asynchronous notifications when certain events (i.e., significant state changes) occur. Thereby, every manageable capability is associated with a topic on which event notifications can be published following the WSDM Event Format (WEF), which is based on CBE. The actual event mechanism of WSDM builds on WS-Notification.

2.7. Conclusion

This chapter has reviewed the state of the art regarding Publish/Subscribe and SOA. We have briefly introduced the basic concepts of Publish/Subscribe and presented an open-source event processing engine that is used for our work. Furthermore, we have compared several service registries and repositories regarding their functionality with a special focus on support for event notifications, since this is one of the main contributions of this thesis. Finally, we have given an overview of current event processing approaches for SOA and Web services, by summarizing current specifications and standards.

The chapter has shown that current approaches are limited since only basic service management events are supported (e.g., new service is published, old service is deleted, etc.). Furthermore, subscriptions can usually only address single events without support for complex event detection. The goal of this thesis is to provide event processing in service-oriented systems. In addition to service management events, this also includes events regarding QoS and runtime information. Moreover, complex event processing mechanisms such as event patterns and sliding window operators should be supported. The Event Processing Layer is described in detail in Chapter 4. Before that, the VRESCo runtime environment is introduced in the following chapter, since our event processing approach has been integrated into VRESCo.

Chapter 3.

QoS-Aware Service Runtime Environment

This chapter describes the VRESCo service runtime environment that builds the foundation for the Event Notification Engine presented in Chapter 4. After a brief architectural overview in Section 3.1, this includes the service metadata model (Section 3.2) and QoS model (Section 3.3). Furthermore, service versioning is shown in Section 3.4. Querying of registry content is provided by the VRESCo Query Engine that is described in Section 3.5. Besides that, Section 3.6 discusses dynamic binding and invocation of services, while service mapping and mediation is described in Section 3.7. Moreover, access control mechanisms for providing authentication and authorization are introduced in Section 3.8. After describing the main components, the VRESCo runtime is evaluated in Section 3.9, while a brief overview of related work is given in Section 3.10. Finally, Section 3.11 concludes this chapter.

Contents

3.1. Overview

In this section, we present an architectural overview of the VRESCo runtime environment. The main goal of this project is to address some of the current challenges in SOC, which have been introduced in Section 1.2. Therefore, VRESCo represents a service runtime environment that aims at efficient development of service-oriented applications. In this regard, it provides a flexible environment that deals with the deficiencies of existing frameworks, namely the lack of dynamism as required to implement service-centric solutions [144].

The overall architecture of VRESCo is shown in Figure 3.1. The most important components are briefly summarized below, while the following sections describe them in more detail.

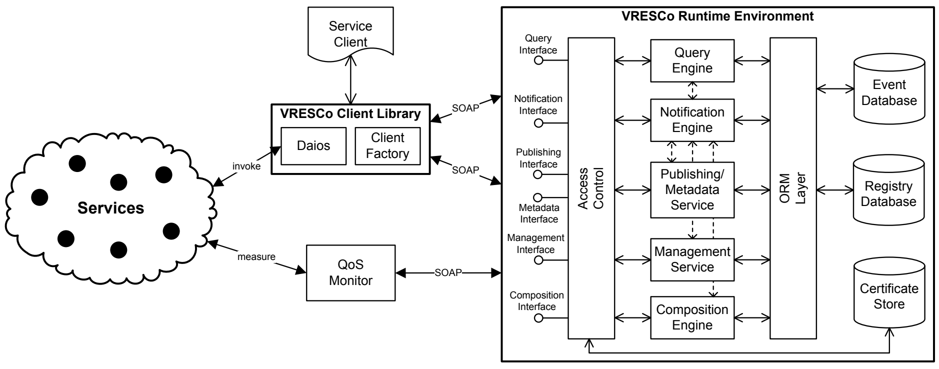


Figure 3.1.: VRESCo Architecture

- **Publishing/Metadata Service:** This service is used for publishing services and associated metadata into the Registry Database, which is done using the Object Relation Mapping (ORM) Layer. Service metadata include functional attributes (e.g., operations, messages, pre- and post-conditions, etc.) and non-functional attributes such as QoS (e.g., response time, availability, etc.).
- **Query Engine:** This engine allows to query for available services and metadata in the Registry Database. Therefore, VRESCo provides optional and mandatory query criteria, and different querying strategies that support exact or fuzzy matching.
- **Notification Engine:** This engine enables clients to subscribe to events of interest. Besides storing the events into the Event Database, notifications are sent to interested subscribers using different notification mechanisms. The Notification Engine represents the second core contribution of this thesis, which is the main focus of Chapter 4.
- **Management Service:** The management tasks involved in the service runtime are provided by the Management Service. Most notable, this includes mechanisms for maintaining users and their access control policies. Furthermore, this service also provides support for QoS management (e.g., by communicating with the external QoS Monitor).

- **Composition Engine:** Composing services to achieve higher level functionality represents a crucial task in SOC. Compositions in VRESCo are defined in a domain-specific language (DSL) called Vienna Composition Language (VCL). The overall idea is to define functional and QoS constraints which can make use of constraint hierarchies by leveraging hard (i.e., required) and soft (i.e., optional) constraints. The Composition Engine then tries to find an optimal solution for these constraints semi-automatically. Therefore, data flow analysis is applied to generate a structured composition model, while both constraint programming and integer programming can be used for solving the optimization problem. Finally, the optimized compositions are executed using Windows Workflow Foundation (WF) [171]. More information on VCL can be found in [158–160].
- **Access Control Layer:** Services and associated metadata represent sensitive information that should not be publicly available. Therefore, the Access Control Layer (ACL) verifies if clients have the required access rights. This is done using both username/password credentials and certificates, which are stored in the Certificate Store.
- **QoS Monitor:** This component is responsible for measuring attributes regarding performance and dependability. The QoS monitor presented in [163] implements a client-side approach based on a low-level TCP analysis and aspect-oriented programming (AOP). Additionally, server-side QoS monitoring has also been integrated into VRESCo, which is discussed in Section 5.3.
- **Daios:** Finally, the actual services maintained in VRESCo are invoked using the dynamic invocation framework DAIOS [92]. In addition to that, the VRESCo runtime provides mediation mechanisms, which are necessary to hide the heterogeneity of different services that perform the same task.

Following the Software as a Service (SaaS) paradigm, the core functionalities (i.e., the VRESCo core services) are provided as Web services. The overall runtime is implemented in C#/.NET on top of the Windows Communication Foundation (WCF) [101, 146], which provides a set of APIs to build and host service-oriented applications. Moreover, we use the ORM framework NHibernate [157] for persisting services and associated metadata in the Registry Database. The Client Library supports dynamic service invocation and provides an API for accessing the VRESCo core services. Since those are implemented as Web services, the Client Library can be provided for several platforms. However, currently it is only available for C#/.NET and Java.

3.2. Service Metadata Model

The VRESCo runtime provides a rich service metadata model capable of storing additional information about services. This is needed for capturing the purpose of services to enable querying and mediating between services that perform the same task. For the description of this metadata model, we give examples from our CPO case study introduced in Section 1.1.

3.2.1. Service Metadata

The VRESCo metadata model introduced in [161] is depicted in Figure 3.2. The main building blocks of this model are *concepts*, which represent the definition of an entity in the domain model. We distinguish between three different types of *concepts*:

- *Features* represent concrete actions in the domain that implement the same functionality (e.g., `Check_Status` or `Port_Number`). *Features* are associated with *categories* which express the purpose of a service (e.g., `PhoneNumberPorting`).
- *Data concepts* represent concrete entities in the domain (e.g., `customer` or `invoice`) which are defined using other *data concepts* and atomic elements such as strings or numbers.
- *Predicates* represent domain-specific statements that are either `true` or `false`. Each *predicate* can have a number of *arguments* (e.g., for *feature* `Port_Number` a *predicate* `Portability_Status_Ok(Number)` may express the portability status of the given *argument* `Number`.

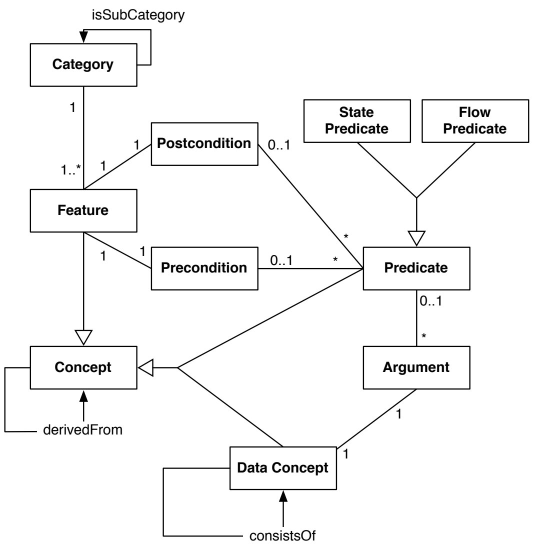


Figure 3.2.: Service Metadata Model

Furthermore, *features* can have *pre-* and *postconditions* expressing logical statements that have to hold before and after its execution. Both types of conditions are composed of multiple *predicates*, each having a number of optional *arguments* that refer to a *concept* in the domain model. There are two different types of *predicates*:

- *Flow predicates* describe the data flow required or produced by *features*. For instance, the *feature* `Check_Status` could have the *flow predicate* `requires(Customer)` as *precondition* and `produces(PortabilityStatus)` as *postcondition*.

- *State predicates* express global states that are valid either before or after invoking a *feature*. For instance, the *state predicate* `notified(Customer)` can be added as *postcondition* to the *feature* `Notify_Customer`.

3.2.2. Service Model

The VRESCo service model constitutes the basic information of concrete services that are managed by VRESCo and can be invoked using the Daios dynamic invocation framework. The service model depicted on the lower half of Figure 3.3 basically follows the Web service notation as introduced by WSDL with extensions to enable service versioning, represent QoS and enable eventing on a service runtime level.

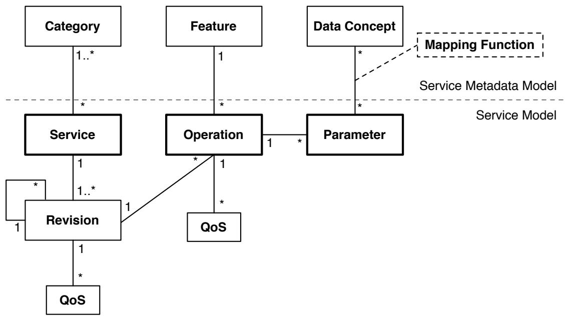


Figure 3.3.: Service Model to Metadata Model Mapping

A concrete service (*Service*) defines the basic information of a service (e.g., name, description, owner, etc.) and consists of a least one service revision. A service revision (*Revision*) contains all technical information that is necessary to invoke it (e.g., 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*). Revisions can have parent and child revisions representing a complete service versioning graph as discussed below. Both revisions and operations can have a number of QoS attributes (*QoS*). The distinction in revision- and operation-specific QoS is necessary, because attributes such as response time depend on the execution duration of an operation, whereas availability is typically given for the revision itself (i.e., if a service is not available, all operations are also unavailable).

3.2.3. Mapping Concrete Services to Metadata

In order to associate elements from the service metadata model to elements from the service model, we establish a mapping between metadata and services. This mapping is shown in Figure 3.3, where the dashed line represents the connection between elements in the metadata model and elements in the service model.

The elements of this service model are mapped to our service metadata model as follows: Services are grouped into categories, 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, where every operation implements exactly one feature. However, we plan to provide support for more complex mappings using the Composition Engine (i.e., features can be represented as compositions of several service operations).

The input and output parameters of operations map to data concepts. Every parameter is represented by one or more concepts in the domain model. This means that all data a service accepts as input or passes as output are well-defined using data concepts and annotated with the flow predicates `requires` (for input) and `produces` (for output). The concrete mapping of service parameters to concepts is defined by *Mapping Scripts* that make use of several *Mapping Functions*. In general, both scripts for mapping from parameter to concept and vice versa have to be specified. If an operation requires a certain state prior to its execution, this requirement can be modeled as a state predicate. The same is true for state changes as result of the execution of an operation.

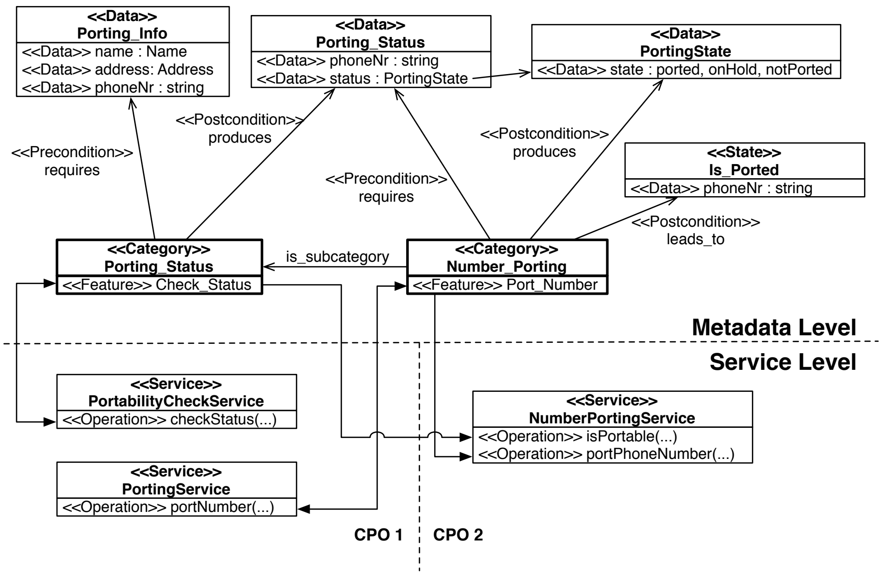

<<Data>> Porting_Info
<<Data>> name : Name
<<Data>> address: Address
<<Data>> phoneNr : string
<<Data>> Porting_Status
<<Data>> phoneNr : string
<<Data>> status : PortingState
<<Data>> PortingState
<<Data>> state : ported, onHold, notPorted
<<State>> Is_Ported
<<Data>> phoneNr : string
<<Precondition>> requires
<<Postcondition>> produces
<<Precondition>> requires
<<Postcondition>> produces
<<Postcondition>> leads_to
<<Category>> Porting_Status
<<Feature>> Check_Status
is_subcategory
<<Category>> Number_Porting
<<Feature>> Port_Number
Metadata Level
Service Level
<<Service>> PortabilityCheckService
<<Operation>> checkStatus(...)
<<Service>> PortingService
<<Operation>> portNumber(...)
<<Service>> NumberPortingService
<<Operation>> isPortable(...)
<<Operation>> portPhoneNumber(...)
CPO 1
CPO 2

Figure 3.4.: Service Mapping

Figure 3.4 gives a mapping example from our CPO case study in UML class diagram notation. In this example, we use two features that are mapped to concrete services by two cell phone operators CPO1 and CPO2: `Check_Status` and `Port_Number`. These features have several pre- and postconditions that refer to flow predicates (e.g., feature `Check_Status` requires data concept `Porting_Info` and produces `Porting_Status`) and state predicates (e.g., feature `Port_Number` leads to state `Is_Ported`).

The mapping from metadata to service level is done between features and operations. For instance, the operation `isPortable` of CPO2's `NumberPortingService` is mapped to the feature `Check_Status` of category `Porting_Status`. Clearly, the input and output of different implementations of one feature may differ. In that case, various mapping operators (e.g., ==, concat, stringToInt, etc.) can be used to mediate between different service interfaces. This service mediation approach is discussed in more detail in Section 3.7.

3.3. Quality of Service Model

Service metadata is essential for defining the purpose and semantics of services, which is often referred to as functional properties of services. In addition to that, non-functional properties regarding Quality of Service (QoS) also play a crucial role in service-oriented systems. They provide means to specify quality guarantees of services such as response time or throughput.

There are several definitions of QoS in literature [107, 155, 212]. According to Rosenberg [158], QoS in service-oriented systems can be defined on the *service*, *choreography* and *orchestration layer*. In addition, QoS attributes can be *deterministic* or *non-deterministic*, depending on whether they are known at service invocation time or not. For the purpose of this thesis and the VRESCo runtime environment, we focus on QoS on the service layer. The basic idea is to define the quality of atomic services without taking choreographies and orchestrations into consideration. In this section, we briefly introduce the VRESCo QoS model which was first introduced in [163], while Section 5.3 explains in detail how QoS is measured and how it can be used to define and monitor Service Level Agreements (SLAs).

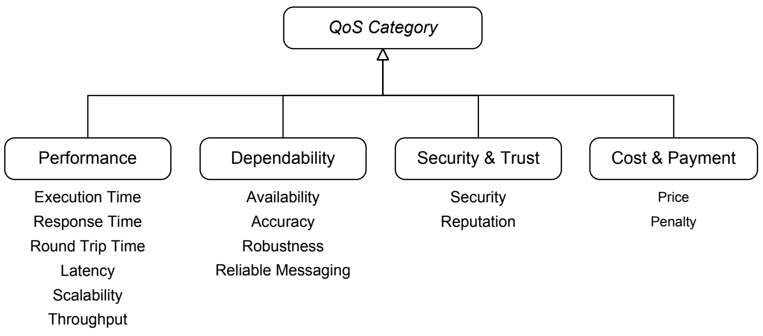


Figure 3.5.: VRESCo QoS Model

Figure 3.5 depicts the VRESCo QoS model which consists of the four categories *Performance*, *Dependability*, *Security/Trust* and *Cost/Payment*. However, the main focus is on performance- and dependability-related attributes since they can be measured automatically. Additionally, user-defined attributes can be added to the runtime which makes the QoS model extensible.

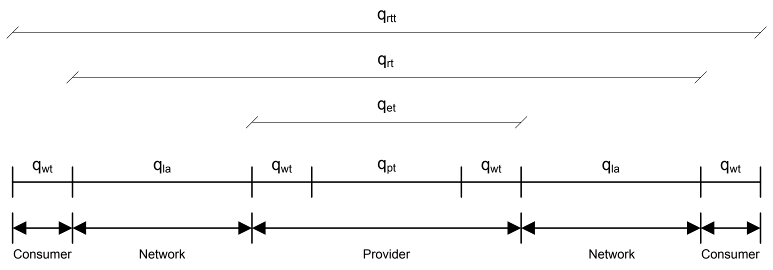


Figure 3.6.: Service Invocation Intervals

Performance-related attributes represent measurable attributes of services regarding performance. For this measurement, service invocations are divided into the time intervals shown in Figure 3.6. In this figure, *Processing Time* q_{pt} represents the time needed for the actual service invocation at the provider. In addition to that, the service provider has to wrap and unwrap SOAP messages, which is indicated by *Wrapping Time* q_{wt}. *Execution Time* q_{et} reflects the entire duration needed to answer the service request. *Latency* q_{la} defines the time needed for the request to be sent over the network from consumer to provider. *Response Time* q_{rt} indicates the service invocation time at the service consumer (i.e., execution time plus latency), while *Round Trip Time* q_{rtt} measures the overall time needed for the service invocation at the service consumer (i.e., response time plus the time for wrapping SOAP messages on the client-side). Please note that the intervals presented so far represent single values that are measured for one invocation. These measurement points can be aggregated to receive meaningful QoS attributes.

Attribute	Formula	Unit
Latency	$q_{la}(n) = \frac{1}{n}\sum_{i=0}^{n} q_{la_i}$	ms
Response Time	$q_{rt}(n) = \frac{1}{n}\sum_{i=0}^{n} q_{rt_i}$	ms
Availability	$q_{av}(t_s, t_e, t_d) = 1 - \frac{t_d}{t_e - t_s}$	percent
Accuracy	$q_{ac}(r_f, r_t) = 1 - \frac{r_f}{r_t}$	percent
Throughput	$q_{tp}(t_s, t_e, r) = \frac{r}{t_e - t_s}$	invocations/s
Price	n/a	per invocation
Reliable Messaging	n/a	{true, false}
Security	n/a	{None, X.509, etc.}

Table 3.1.: VRESCo QoS Attributes

Table 3.1 shows all QoS attributes currently considered in VRESCo. For each attribute we list the distinct name, the formula how the attribute is calculated (or "n/a" if deterministic) and the unit. *Latency* $q_{la}(n)$ indicates the time a request needs on the wire. It is calculated as the average value of n individual measuring points q_{la_i}. *Response time* $q_{rt}(n)$ consists of the latency for request and response plus the execution time of the service (again the average of n individual values q_{rt_i}). *Availability* $q_{av}(t_s, t_e, t_d)$ stands for the probability that a service is up and running (t_s and t_e indicate the requested time period, while t_d represents the total time the service was down). *Accuracy* $q_{ac}(r_f, r_t)$ is the probability of a service to produce correct results where r_f denotes the number of failed requests and r_t denotes the total number of requests. Finally, *Throughput* $q_{tp}(t_s, t_e, r)$ represents the maximum number of requests a service can process within a certain period of time (denoted as $t_e - t_s$) where r is the total number of requests during that time.

Besides these performance- and dependability-related attributes VRESCo provides several other deterministic attributes related to security and cost, such as *Reliable Messaging* (i.e., whether reliable messaging is supported according to WS-ReliableMessaging [133]), *Security* (i.e., if the service provides security according to WS-Security [139] such as username/password, X.509 [67], SAML [135] or Kerberos [126]) and *Price* (i.e., the price for one invocation). It should be noted, however, that due to the extensible QoS model users are able to define additional QoS attributes if needed (e.g., reputation, trust, etc.).

3.4. Service Versioning

Like any kind of software system, services are subject to permanent change. Vendors constantly add new functionality or change the requirements of existing services, and strive to increase quality aspects such as reliability or security. This software adaptation process is usually referred to as software evolution and is subject to a vital research community [17].

Despite the need for versioning of services, there is still only limited support by existing registry standards. Therefore, we provide a simple, but powerful versioning mechanism in the VRESCo runtime environment. The main objective of this mechanism is to provide *version transparency*: Selection of service versions (also called service revisions), as well as mediation and rebinding between different revisions should be handled automatically by the service runtime.

In order to manage Web service evolution, service registries need to store not only the service revisions, but also how they relate to each other. We use the notion of *service version graphs* to represent these dependencies. For every service, there is exactly one service version graph. These graphs are directed, with nodes representing concrete service revisions and edges representing *predecessor-successor* relationships. The semantics of these relationships is that revision A is a predecessor of revision B, if B is the result of changes in A. All revisions in a version graph refer to the same base service, but are on different maturity levels and different stages in the base service's lifecycle. Furthermore, service version graphs may contain *branches* and *merges*.

Branches represent situations where two or more variants of a service evolve in parallel (e.g., a special version with specific behavior for a subset of users). Merge revisions consolidate two or more branches in the version graph. In terms of graph theory, branch revisions are defined by an out-degree greater than 1, while merge revisions have an in-degree greater than 1.

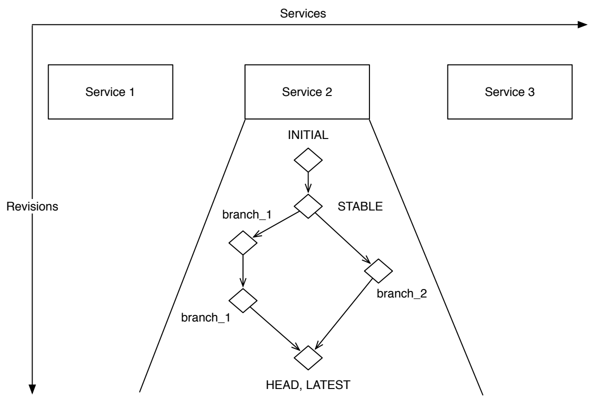


Figure 3.7.: Service Version Graph

In order to provide helpful information for the user, revisions in the service version graph may be tagged. Generally, a revision tag is a string attached to one or more service revisions that describes the revision's functionality, stability, maturity level or any other functional or non-functional aspect. Figure 3.7 shows an example service version graph. Several revision tags (e.g., `INITIAL`, `branch_1`, `branch_2`, etc.) have been assigned to the graph to identify revisions. The revision tagged `STABLE` represents a branch revision, while the two branches are again merged in the revision tagged `LATEST`.

As stated above, service clients should optimally be version-transparent (i.e., version selection, mediation between incompatible versions and automatic rebinding should be handled automatically). Clients should be able to switch between versions freely, and invoke any revision of the service without adapting the client code. In VRESCo, this version-transparency is achieved through service proxies. These proxies are bound to a selection strategy and update their target revision whenever there is a better match than their current target. Therefore, proxies are responsible for mediating between the user-provided data and the expected input of the target revision. Service mediation is discussed in more detail in Section 3.7.

Selection strategies are queries on the service version graph that use the defined revision tags and the inter-version relationships to select the most appropriate service revision for users. A user may, for instance, define a selection strategy that always binds to the most recent revision in the graph, to the latest stable one, or to a revision belonging to a specific branch. Unlike one-time queries, this selection is monitored by the proxy. If the service version graph changes (e.g., because of the insertion of a new revision) the result of the selection may also change, and the proxy may update its target service to reflect this change. We refer to this change in the proxies target service as dynamic rebinding. Dynamic rebinding is transparent to the client (i.e., it can be safely assumed that the most appropriate service according to the selection strategy is invoked). We describe dynamic binding in more detail in Section 3.6.

The implementation of our approach in VRESCo has been realized using versioning metadata. Such versioning metadata represent the relationships in the service version graph and are defined by the service provider when publishing new services or revisions. It should be noted that our implementation does not enforce any specific rules about the degree of change between two revisions in a service version graph. Consequently, it is the provider's decision whether the differences between two service versions are so fundamental that an entire new service should be created instead. Furthermore, any evolutionary step may contain multiple discrete changes (i.e., the actual difference between revisions may be arbitrarily complex). Establishing branch and merge revisions is also done by the service provider by defining the appropriate relationships in the service version graph. In this regard, our implementation does not impose any further restrictions on branching and merging (i.e., the service providers are free to use branching and merging at their convenience).

Tag	Description	Assigned by
`INITIAL`	The first version of this service	VRESCo
`STABLE`	A well-tested production-level service version	provider
`HEAD`	The most recent version in a branch	VRESCo
`LATEST`	The most recent version in the entire version graph; implies HEAD	VRESCo
`DEPREC`	The version is online, but should not be used anymore (deprecated)	provider
`OFF`	The version has been taken offline and is not available anymore	provider

Table 3.2.: Default Revision Tags

As already mentioned, VRESCo supports the concept of revision tags. Tags may either be default tags with a well-defined meaning, or arbitrary strings defined by the user. Table 3.2 gives a list of currently available default tags including their description. It should be noted that some of these default tags are assigned automatically by VRESCo (`INITIAL`, `LATEST`, `HEAD`) while others (`STABLE`, `DEPREC`, `OFF`) have to be assigned by the service provider.

3.5. Service Querying

The revision tags introduced above represent one way to mark revisions, so that they can be identified by service providers and consumers. The VRESCo Query Language (VQL) provides a querying framework that is capable of querying all information stored in the Registry Database (e.g., services, metadata, QoS, tags, etc.). In this section, we discuss the architecture of this querying framework, as well as query specification and query processing.

Before we go into the details of the architecture, we want to briefly mention the requirements we pose on our querying framework. First of all, declarative query languages such as SQL refer to database tables and columns, which makes queries invalid as soon as the database schema changes. Following the Query Object Pattern [56], queries can be built programmatically using query criteria that refer to classes and fields instead. These queries are finally translated into SQL statements, which makes them independent of the database schema. In this regard, VQL should provide such object-oriented querying interface and corresponding query expression library (similar to the Hibernate Criteria API [157]). Moreover, VQL queries should be type-safe (i.e., the query requester specifies the expected type of the query results) and secure (i.e., queries are protected against well-known security issues such as SQL injection [63]).

Another important requirement for our querying framework was the need for both optional and mandatory query criteria. This is important since some queries should return only results that match all criteria, while optional criteria enable fuzzy querying. The latter is often needed when exact querying does not return any results. For instance, optional criteria can be used to define QoS attributes that are nice to have, but not strictly required (e.g., service response time should be less than 500 ms).

3.5.1. Query Architecture

The architecture of the VQL framework is shown in Figure 3.8. In general, the Client Library is used to invoke VRESCo core services (e.g., Publishing Service). Since these invocations represent remote method invocations, the Data Transfer Object pattern [56] is used to reduce the information sent from clients to the core services. Therefore, the VRESCo runtime operates on the *core model* (which represents the service metadata model introduced in Section 3.2), while clients operate on the *user model*. The task of the Data Access Layer (DAL) is to convert *core objects* to *user objects* and vice versa. The mapping between user and core objects is defined at design time using .NET attributes [96]. More precisely, source code attributes are used to define which classes (attribute `MappedClass`) and properties (attribute `MappedProperty`) from the core model map to which classes and properties from the user model.

The advantage of this architecture is that clients operate on the *user model*, which represents a restricted view of the *core model*. Therefore, some information can be hidden from the clients (e.g., database IDs or versioning information for optimistic locking). Consequently, the VQL framework has to provide view-based querying, to be able to query on both models (depending on whether the query is issued client- or server-side). The task of the ORM Layer is then to map the entities of the *core model* to the *database model* (i.e., concrete database tables and columns), which is realized by NHibernate [157] using dedicated data access objects (DAOs).

According to this architecture, user queries are formulated using the Client Library, that provides an object-oriented querying interface to define query criteria, which is discussed in the next section. The query is then sent to the VRESCo runtime (step 1) and forwarded to the VQL

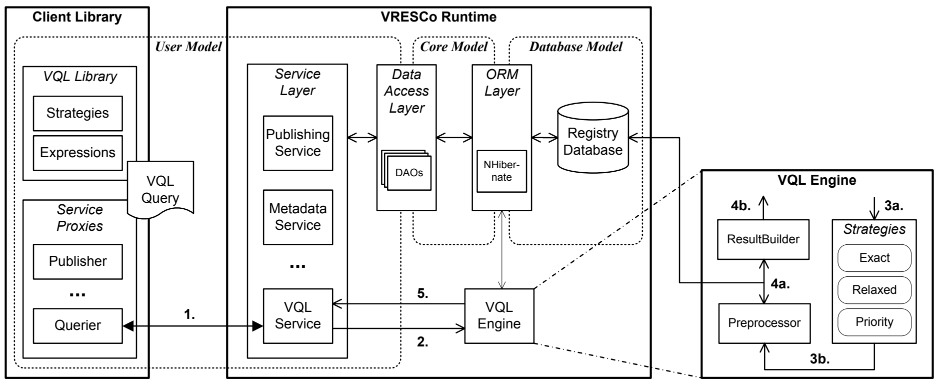


Figure 3.8.: VRESCo Query Architecture

Engine (step 2). The *Preprocessor* component is used to analyze the VQL query and generate the corresponding SQL query according to the client's querying strategy (step 3a+3b). This SQL query is then executed on the Registry Database (step 4a), while the results are converted by the *ResultBuilder* (step 4b). The details of query processing are described in Section 3.5.3. Finally, the results are sent back to the client (step 5).

3.5.2. Query Specification

From a user's perspective, the most important question is how queries are specified. As stated above, VRESCo provides a powerful and easy to use querying API for this purpose. In general, VQL queries consist of six elements:

- *Return Type R* defines the expected data type of the query results. The return type needs to be an element of the VRESCo metadata model (e.g., a list of `Feature` objects).
- *Mandatory Criteria C_m* describe constraints which have to be fulfilled by the query (e.g., response time must be less than 500 ms).
- *Optional Criteria C_o* add constraints which should optimally be fulfilled but are not required (e.g., service provider should be company *X*).
- *Ordering O* can be used to define a specific ordering of the query results (e.g., sort ascending by property ID).
- *Result Limit L* can be used to restrict the number of results that should be returned (e.g., only 10 results, or unlimited which is specified as 0).
- *Querying Strategy S* finally defines according to which strategy the query is executed (e.g., exact or fuzzy matches).

The most important elements are criteria since they actually represent the constraints of the query. Moreover, criteria have different execution semantics depending on the querying strategy, which is discussed in Section 3.5.4. However, the main motivation is to allow the specification of mandatory and optional criteria.

Type	**VQL**	**SQL**	**Description**
C_m	`Add`	`WHERE`	Mandatory criteria
C_o	`Match`	`IN/JOIN`	Optional criteria
	`And`	`AND`	Conjunction of two expressions
	`Or`	`OR`	Disjunction of two expressions
	`Not`	`NOT`	Negation of an expression
	`Eq`	=	Equal operator
	`Lt`	<	Less operator
	`Le`	<=	Less or equal operator
E	`Gt`	>	Greater operator
	`Ge`	>=	Greater or equal operator
	`Like`	`LIKE`	Similarity operator for strings
	`IsNull`	`IS NULL`	Property is null
	`IsNotNull`	`NOT NULL`	Property is not null
	`In`	`IN`	Property is in a given collection
	`Between`	`BETWEEN`	Property is between two values
	`Order`	`ORDER BY`	Ordering of query results
O	`Asc`	`ASC`	Ascending ordering
	`Desc`	`DESC`	Descending ordering

Table 3.3.: VQL/SQL Translation

In general, criteria consist of a set of *expressions* E that are used to define common constraints such as comparison (e.g., smaller, greater, equal, etc.) and logical operators (e.g., AND, OR, NOT, etc.). Table 3.3 shows criteria (C), expressions (E) and orderings (O) which are currently provided by VQL. Furthermore, the table indicates how each of these elements is translated to SQL, which is described in more detail later. It should be noted that VQL is extensible in that further expressions can be added easily. Therefore, expressions have to extend the `AbstractCriterion` class and implement the `ToSqlString()` method to define how they are translated into SQL.

Listing 3.1 shows an example query for finding services that implement the `Notify_Customer` feature in our CPO case study. As described above, queries are parameterized using the expected return type. In this case, the type `ServiceRevision` (line 2) expresses that the result of the query is a list of service revisions. In our example, two `Add` criteria (lines 5–6) are used to state that services have to be active and that each service has to implement the `Notify_Customer` feature (by using the `Eq` expression). The first parameter of expressions is usually a string representing a path in the user or core model (e.g., `Service.Owner.Company` describes the

```
1  // create query object
2  var query = new VQuery(typeof(ServiceRevision));
3
4  // add query criteria
5  query.Add(Expression.Eq("IsActive", true));
6  query.Add(Expression.Eq("Service.Category.Features.Name", "NotifyCustomer"));
7  query.Match(Expression.Eq("Service.Owner.Company", "CompanyX"), 1);
8  query.Match(Expression.Like("Tags.Property.Name", "STABLE", LikeMatchMode.Start), 3);
9  query.Match(
10     Expression.Eq("QoS.Property.Name", "ResponseTime") &
11     Expression.Lt("QoS.DoubleValue", 1000.0), 5);
12
13 // execute query
14 var querier = VRESCoClientFactory.CreateQuerier("username", "password");
15 var results = querier.FindByQuery(query, 10, QueryMode.Priority) as IList<ServiceRevision>;
```

Listing 3.1: VQL Sample Query

company property of the service owner). These strings are central to VQL, and are referred to as *property paths*. Additionally, three `Match` criteria are added in the example (lines 7–11). The first criterion expresses that services provided by *CompanyX* are preferred, while the second criterion defines that revisions should have tags starting with 'STABLE' (`Like` expression). The third criterion specifies an optional QoS constraint on response time, which should be less than 1000 ms. The operator '&' in line 10 represents a shortcut for an `And` expression. All three `Match` criteria use priority values as third parameter to define the importance of a criterion.

The query is finally executed (lines 14–15) by instantiating a `querier` object using the Client Factory, and invoking the `FindByQuery` method using the desired querying strategy (e.g., `QueryMode.Priority`). Furthermore, the result limit of the query is set in order to return only 10 results.

3.5.3. Query Processing

Query processing is illustrated in Figure 3.8. When the query is sent to the VQL Engine (step 2), the specified querying strategy is executed (step 3a), which is implemented using the strategy design pattern [57]. The query is forwarded to the *Preprocessor* component (step 3b), which is responsible for analyzing the VQL query and generating the corresponding SQL query. Next, a NHibernate session is created to execute the generated SQL query on the database (step 4a). After execution, the *ResultBuilder* component takes the results from the NHibernate session context. Since these results represent *core objects*, they may have to be converted back into the corresponding *user objects* (i.e., if the return type refers to the *user model*). This is done dynamically by invoking the constructor of the corresponding object using reflection (step 4b). For both models, however, the *ResultBuilder* guarantees type-safety of the results, which are finally sent back to the client (step 5).

Algorithm 1 *processQuery*(R, C, S, O)

```
Require: $R \neq \{\}$
1: $query \leftarrow \{\}$
2: if ( isUserObject(R) ) then
3:     $R \leftarrow$MapUserToCoreObject(R)
4: end if
5: $assocInfo \leftarrow R$
6: for all ( $crit \in C$ ) do
7:     for all ( $expr \in$ GetExpressions(crit) ) do
8:         $assocInfo \leftarrow assocInfo \cup$ ResolveAssoc(expr)
9:         $propInfo \leftarrow params \cup$ ResolveProp(expr)
10:    end for
11: end for
12: $query \leftarrow$ BuildFrom(assocInfo, propInfo, S)
13: $query \leftarrow$ BuildWhere(query, assocInfo, propInfo, S)
14: $query \leftarrow$ BuildOrder(query, O)
15: return query
```

Algorithm 1 depicts the pseudo-code of the *Preprocessor*. If the query refers to the *user model*, it is first transformed to the *core model* (lines 2–4). The *Preprocessor* then iterates over all criteria and expressions (lines 6–11). The *ResolveAssoc* function recursively analyzes the property paths of each expression to determine the necessary table joins. Similarly, the *ResolveProp* function extracts the property values of each expression. To give an example, reconsider line 7 of Listing 3.1: The property path `Service.Owner.Company` represents two associations `Service` and `Owner` that will be resolved using table joins, and one property `Company` that will be compared with the expression's property value *CompanyX*. The concrete association/table and property/column names are retrieved using the ORM Layer. The collected information is finally used to build `FROM`, `WHERE` and `ORDER` clauses of the SQL query (lines 12–14), according to the VQL/SQL translation shown in Table 3.3.

3.5.4. Querying Strategies

The querying strategy influences how queries are executed. More precisely, it defines the *Preprocessor*'s behavior during SQL generation. The basic transformation process can be summarized as follows: `Add` criteria are transformed to predicates within the SQL `WHERE` clause, whereas `Match` criteria are handled as SQL sub-selects (`IN` or `JOIN`, see Table 3.3).

The *exact querying* strategy forces all criteria to be fulfilled, irrespective whether this is `Add` or `Match`. However, there are scenarios where `Match` has to be used instead of `Add` in order to get the desired results (i.e., by enforcing sub-selects using `IN` 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 queries would result in `null` since the associated tables are joined more than once. As an example reconsider the query in Listing 3.1 using the *exact* strategy. When having only one criterion with respect to `QoS`, `Add` can be used. However, if there would be a second QoS criterion, `Match` is required.

The *priority querying* strategy uses priority values for each optional criterion in order to accomplish weighted matching of results. Therefore, each `Match` criterion defines a priority weight, which is internally added if the criterion is fulfilled. The query finally returns the results sorted by the sum of priority values. To give an example, the query in Listing 3.1 uses the priority values "1", "3" and "5". This means that the constraint on response time is more important than the constraint on revision tags. More precisely, queries that fulfill only the third `Match` criterion are preferred over queries that fulfill the first and the second `Match` criterion (since $5 > 3 + 1$).

The *relaxed querying* strategy represents a special variant of *priority querying* where each `Match` criterion has priority 1. Thus, this strategy simply distinguishes between optional and mandatory criteria. Results are then sorted based on the number of fulfilled `Match` criteria. This allows to define fuzzy queries by relaxing the criteria, which can be useful when no exact match can be found for a query. To achieve the necessary behavior, *relaxed* and *priority* querying both translate `Match` criteria into sub-selects using `JOIN` predicates.

More details on query processing and querying strategies, together with illustrative examples for each translation step, can be found in [87]. Furthermore, Appendix A depicts SQL queries that are generated by our approach. This contains *exact* (Listing A.2), *priority* (Listing A.3) and *relaxed* strategy (Listing A.4). These listings illustrate how an example VQL query (Listing A.1) is translated into SQL following the VQL/SQL translation shown in Table 3.3.

3.6. Dynamic Binding and Invocation

In general, *dynamic binding* (or *late binding*) is the process of linking an abstract service to a concrete service instance at execution time. Ideally, this should be handled transparently by the service runtime environment. Dynamic binding in service-oriented systems is mostly used in combination with runtime service discovery. This means that service consumers try to find services that match given criteria. Based on the available services, the consumers select the service that best fulfills their constraints and bind to the service dynamically at runtime. Since, these constraints may change over time, it is often necessary to rebind to other functionally identical services which we refer to as *dynamic rebinding*. Dynamic rebinding additionally requires mechanisms that provide *dynamic invocation* as opposed to invoking Web services statically using pre-defined stubs.

3.6.1. Dynamic Binding

In our first attempt to address these problems, we have introduced the notion of *QoS-based* and *content-based* dynamic binding [119]. QoS-based dynamic binding basically uses criteria on QoS attributes during service selection, which is specified using queries. While the first version of VRESCo was built on simple queries using keyword-based matching, the querying framework has later been significantly extended, which was described in Section 3.5.

To achieve content-based dynamic binding, on the other hand, a mapping between some application logic and a distinct service has to be provided. The overall idea is to add service metadata that provide service identifiers (e.g., `PortingProvider.TELCO2`). These identifiers represent application-specific unique service names that can be used in the application code (instead of complex queries).

3.6.2. Rebinding Strategies

Over time, we have stepwise refined our initial ideas regarding dynamic rebinding in combination with service versioning as presented in [90]. In this regard, we have finally put our main focus on QoS-based (or more general speaking, query-based) dynamic binding of services. Please note that content-based dynamic binding basically can be seen as a simplified version of query-based binding.

In general, service selection mechanisms (either queries or identifiers) are used to define constraints on the services to invoke. However, services are changing over time or may even be deleted from the service registry. Moreover, the user-defined constraints on services may also change over time. Taken together, these two issues raise the need for dynamic rebinding. The basic idea is to transparently rebind to functionally equal services according to user-defined properties. For instance, clients may want to rebind to the best available service regarding some QoS attribute, or to the most recent version of a service. Therefore, VRESCo provides different *rebinding strategies* that are implemented using the well-known strategy pattern [57]. It should be noted that dynamic binding using rebinding strategies may also require service mediation, since the interfaces of the two services may differ. The VRESCo service mediation approach is described in Section 3.7.

Strategy	**Description**
Fixed	The proxy never updates its binding (i.e., always invoke service X).
Periodic	The proxy reconsiders its binding periodically (e.g., once every minute).
OnDemand	The proxy reconsiders its binding on client requests.
OnInvocation	The proxy reconsiders its binding prior to every service invocation.

Table 3.4.: Rebinding Strategies

Table 3.4 summarizes all rebinding strategies provided in VRESCo. *Fixed* proxies are used in scenarios where rebinding is not needed (e.g., because of existing contractual obligations). *Periodic* rebinding causes constant overhead since the proxies verify their binding periodically. Clearly, this is inefficient if invocations happen infrequently. *OnDemand* rebinding results in low overhead but has the drawback that the binding is not always up-to-date. In contrast to this, *OnInvocation* rebinding guarantees accurate bindings but seriously degrades the service invocation time. Thus, all rebinding strategies have their strengths and weaknesses, and it depends on the specific situation which strategy to use. In Section 5.1, we introduce another strategy based on events, which aims at combining the advantages of all other strategies.

3.6.3. Dynamic Invocation

Besides dynamic binding, dynamic invocation of services represents another important goal of service-centric systems. Current Web service frameworks often make use of pre-generated stubs to access services. However, this makes them hard-wired to specific service providers which does not follow the initial idea of dynamic binding in SOA.

In this regard, we aim at dynamic, stubless and protocol-independent service invocation provided by Daios [92]. In contrast to existing dynamic service invocation frameworks such as Apache WSIF [3], Daios is highly message-driven and does not focus on RPC-style communication as provided by distributed object technology [189]. The overall idea of Daios is to abstract from the technical internals of the services, and thus, decouple the clients from the services they consume. Furthermore, the framework supports synchronous (request-response), one-way and asynchronous communication, which is suitable for long-running transactions.

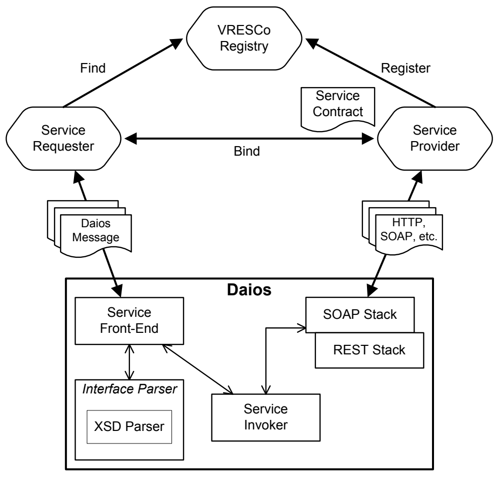


Figure 3.9.: Daios Architecture

The architecture of Daios is shown in Figure 3.9 which is adapted from [92]. It consists of three components: The *Front-End* component contains the framework classes that orchestrate the other components. The *Interface Parser* component is used to preprocess service descriptions (e.g., WSDL and XML schema), while the *Service Invoker* component is used to invoke services using the SOAP [202] or REST [54] stack. Clients (i.e., service requesters) communicate with the framework by sending Daios messages to the Front-End. These messages represent the internal data representation of Daios, which can be seen as unordered lists of name-value pairs. Furthermore, messages can be nested in order to build arbitrary data structures. Thus, Daios messages can encapsulate XML schema complex types.

Invoking services is done in three steps: Firstly, services are discovered using the VRESCo querying mechanism. Secondly, the service is bound during the preprocessing phase that collects all service information from the interface description. Thirdly, the service is dynamically invoked using the corresponding stack (e.g., SOAP, REST). Therefore, the Daios messages are mapped and converted to the actual input expected by the service (e.g., use correct encoding). Finally, the output from the service is converted back into a Daios output message.

```
16 // create service proxy
17 var proxy = querier.CreateRebindingMappingProxy(
18         query, QueryMode.Exact, 100, new PeriodicRebindingStrategy(60000));
19
20 // create Daios message
21 DaiosMessage request = new DaiosMessage();
22 request.SetString("ReceiverNr", "0043-12345678");
23 request.SetString("SenderNr", "0043-98765432");
24 request.SetString("Message", "Phone number has been ported!");
25
26 // invoke service
27 DaiosMessage result = proxy.RequestResponse(request);
28 String confirmationMsg = result.GetString("Confirmation");
```

Listing 3.2: VRESCo Service Invocation

To give an example, Listing 3.2 shows a service invocation from our CPO case study. It should be noted that this listing continues Listing 3.1. Thus, the query in line 18 refers to the former listing and the code is not duplicated for brevity. In line 17, a Daios service proxy is created that is capable of rebinding and service mediation if the interfaces do not match (see Section 3.7). This time, the query is issued using the *exact* strategy (i.e., all query criteria have to be fulfilled). Furthermore, the result set is limited to 100 entries. Line 18 defines that the service proxy should be created using the *periodic* strategy. In this case, the proxy reconsiders its binding every minute (i.e., 60000 ms). In lines 21–24, the input message for the `Notify_Customer` feature is built by specifying data types and name-value pairs. Finally, the corresponding service is executed in line 27 using the request-response pattern. After successful service invocation, the response can be extracted from the result message (line 28).

3.7. Service Mediation

So far, the description of dynamic binding and invocation has not addressed the heterogeneity of service interfaces. Dynamically rebinding and invoking alternative services is rather simple when service interfaces are identically. However, it gets a lot more complex when this assumption does not hold, which is often neglected by existing research approaches.

The VRESCo Mapping Framework (VMF) introduces mechanisms to define the mapping from abstract features to concrete service operations as introduced in Section 3.2.3. VMF follows the notation of a *feature-driven* metadata model. Therefore, a client that wants to invoke a certain service does not provide the input of the concrete service directly, but already in a high-level representation (i.e., the feature input in VRESCo terminology). The VRESCo runtime takes care of *lowering* and *lifting* 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).

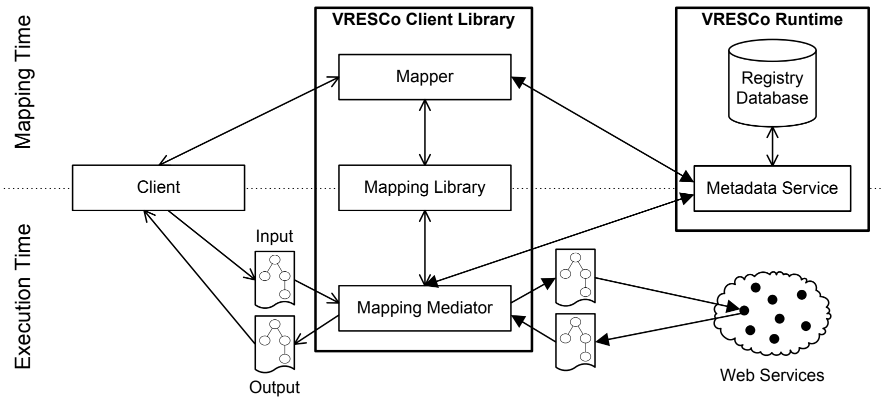


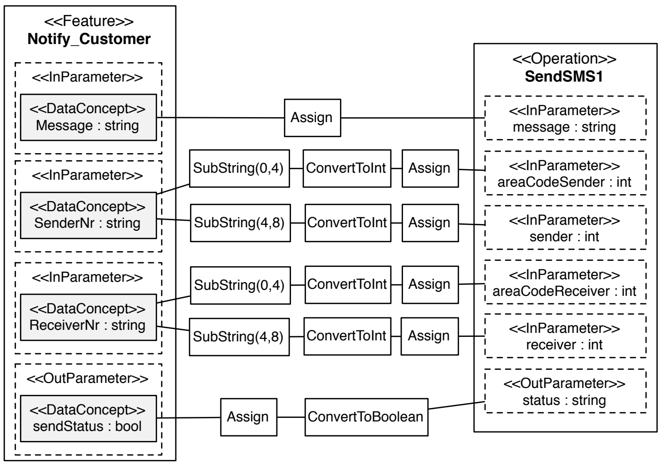
Figure 3.10.: VMF Architecture

Figure 3.10 is adapted from [93] and shows an overview of the VMF architecture. Generally, service mapping and mediation in VMF is done in two steps: Firstly, at *Mapping Time* different services that implement the same feature are mapped using the *Mapper* component. This is done by creating lifting and lowering information (i.e., *Mapping Scripts*) for each service, which is done using the *Mapping Library*. This information is stored in the Registry Database by invoking the *Metadata Service*. Secondly, at *Execution Time* VMF injects a Daios *Mapping Mediator* which is responsible for the mediation process. Mediators are integrated into the Daios chain of mediators which is presented in more detail in [93]. The Mediator retrieves the lifting and lowering scripts from the Metadata Service at runtime, and executes the corresponding mapping. This is done by applying all mapping functions sequentially (i.e., in the order they have been specified). In that sense, VMF implements an imperative and interpreted domain-specific language (DSL).

Functions	Description
`Assign`	Link one parameter to another (source and destination must have the same data type)
`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 (e.g., `substring`, `concat`, etc.)
`Math`	Basic mathematical and logical operations (e.g., `addition`, `round`, `and`, `or` etc.)
`CSScript`	Define complex mappings directly in C#

Table 3.5.: VMF Mapping Functions

Mapping scripts are defined using the *Mapping Library* which includes a number of *Mapping Functions*. Mapping functions are the atomic building blocks from which all mapping scripts are constructed. We have summarized the provided mapping functions in Table 3.5 (grouped into 7 categories). Probably the most important function is `Assign`, which is used to map one input parameter or intermediary result to an output parameter (i.e., a Web service operation parameter in case of a lowering script, a feature output parameter in case of a lifting script). Functions from the `Constants` group are used to create new data directly in the mapping. All remaining mapping functions are used to transform parameters in various ways (e.g., from one data type to another, using string manipulation, or using mathematical and logical operations). Furthermore, more complex mappings can be defined in the CS-Script language [170]. Essentially, this allows to deploy custom mapping functions, which can use the full power of the C# programming language. For instance, this can be used to invoke external Web services at mediation time.



<<Feature>>
Notify_Customer
<<InParameter>>
<<DataConcept>> Message : string
<<InParameter>>
<<DataConcept>> SenderNr : string
<<InParameter>>
<<DataConcept>> ReceiverNr : string
<<OutParameter>>
<<DataConcept>> sendStatus : bool
Assign
SubString(0,4)
SubString(4,8)
ConvertToInt
Assign
ConvertToBoolean
<<Operation>>
SendSMS1
<<InParameter>> message : string
<<InParameter>> areaCodeSender : int
<<InParameter>> sender : int
<<InParameter>> areaCodeReceiver : int
<<InParameter>> receiver : int
<<OutParameter>> status : string

Figure 3.11.: VMF Mapping Example

We give a concrete mapping example in Figure 3.11. In this example, the abstract feature `Notify_Customer` from the CPO case study, which needs three input parameters and produces one output parameter, is mapped to the concrete Web service operation `SendSMS1`. The parameter `Message` is identical in both interfaces, and can therefore be mapped directly (using only an `Assign` function). Note that for the `Assign` function to work both sides need to be represented using the same data concept (in this case string). The parameter `SenderNr` is split into the area code and the actual number. This is done using the string operation `SubString` which takes the start index of the string and the length of the substring as parameters. Afterwards, both substrings are converted to integers using the `ConvertToInt` function. This is necessary since assigning a string to an integer is not possible. The `ReceiverNr` is handled similarly. So far, only input parameters have been mapped (i.e., all information given so far forms the lowering script for this service). The lifting script, which defines how the service output is mapped to the feature output, consists only of a `ConvertToBoolean` and another `Assign` function.

```
// create mapper for feature and operation
Mapper mapper = metadataService.CreateMapper(NotifyCustomer, SendSMS1);

// map feature message to operation message
mapper.AddMappingFunction(
  new Assign(mapper.FeatInParams[0].GetChild("Message"), mapper.OpInParams[0]));

// get AreaCode, convert to int and map it to operation
Substring acSenderStr = new Substring(mapper.FeatInParams[0].GetChild("SenderNr"), 0, 4);
acSenderStr = mapper.AddMappingFunction(acSenderStr);
ConvertToInt acSenderInt = new ConvertToInt(acSenderStr.Result);
acSenderInt = mapper.AddMappingFunction(acSenderInt);

mapper.AddMappingFunction(
  new Assign(acSenderInt.Result, mapper.OpInParams[1]));

// get SenderNr, convert to int and map it to operation
Substring senderNrStr = new Substring(mapper.FeatInParams[0].GetChild("SenderNr"), 4, 8);
senderNrStr = mapper.AddMappingFunction(senderNrStr);
ConvertToInt senderNrInt = new ConvertToInt(senderNrStr.Result);
senderNrInt = mapper.AddMappingFunction(senderNrInt);

mapper.AddMappingFunction(
  new Assign(senderNrInt.Result, mapper.OpInParams[2]));
```

Listing 3.3: VMF Mapping Example Code

Listing 3.3 illustrates how the first two mappings of this example (i.e., `Message` and `SenderNr`) are specified in C# code. The feature `Notify_Customer` requires the data concepts `Message`, `SenderNr` and `ReceiverNr` as input (data type *string*). The `SendSMS1` operation also requires the parameter `Message` (*string*), but sender and receiver number are split into area code and number (*integer*). In this example, phone numbers contain an area code with four digits, followed by a number with eight digits.

Line 2 shows how the Mapper is created for feature `Notify_Customer` and operation `SendSMS1`. Clearly, both objects have to be queried from the registry using VQL before the Mapper can be created (this is not shown in Listing 3.3 for brevity). The `Assign` function used in line 6 links the `Message` of the feature (`Notify_Customer`) to the `Message` of the operation (`SendSMS1`), whereas `mapper.AddMappingFunction()` adds this link to the mapping. Lines 9–15 get the area code from the feature's `SenderNr` as substring. Then it is converted using the `ConvertToInt` function to an integer, which is finally assigned to operation's input parameter `AreaCodeSender`. In lines 18–24 the same is done to map the sender number from the feature's input to the operation's input. Similar mapping functions must be added to map the receiver number from feature and operation, which is not shown in the figure for brevity.

More detailed information about service mediation and the VMF mapping framework can be found in [93] and [70]. It should be noted that doing the mapping programmatically using the API provided by the *Mapper* component gives great flexibility to the developer. However, it remains a tedious task for complex mappings and large numbers of features and operations. Therefore, we envision to support graphical mapping of services as part of the VRESCo GUI, as provided by other mapping tools such as Altova Mapforce [2].

3.8. Security Mechanisms

The services and associated service metadata stored in the Registry Database represent sensitive information that should usually not be available for the public. This raises the need for appropriate security mechanisms that guarantee integrity and confidentiality of this information. Such security mechanisms have been implemented in VRESCo, which is discussed in the following section.

3.8.1. Authentication

Authentication mechanisms generally aim at confirming the identity of users or objects. The VRESCo runtime is not targeted to public Web services but focuses on enterprise scenarios. In these settings, security issues often play a crucial role since only specific clients should be able to access internal services and resources. Therefore, it is important to first authenticate these clients before authorization mechanisms can be applied successfully. For this reason, a dedicated User Management Service has been implemented that is responsible for maintaining all users known to the runtime. In this service, users are assigned to specific user groups that allow fine-grained access control policies. For every user, the runtime maintains several properties (e.g., first name, last name, company, etc.) and the needed user credentials such as username and password.

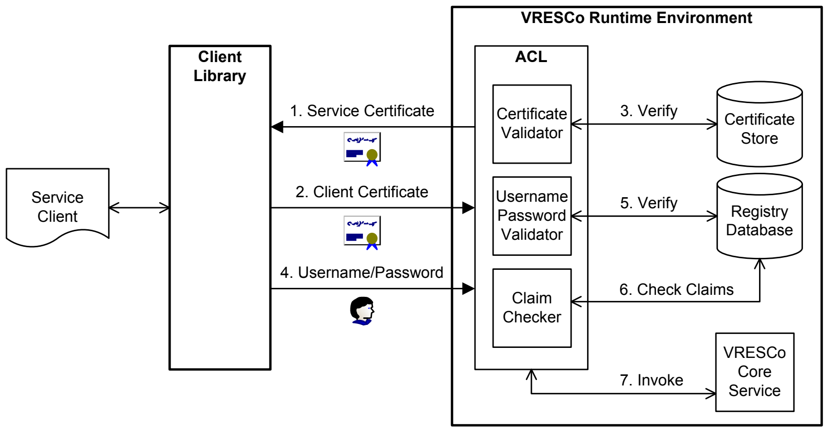


Figure 3.12.: Authentication and Authorization in VRESCo

Figure 3.12 shows the VRESCo authentication mechanism by using a typical invocation of some core service (e.g., Publishing or Metadata Service). As shown before, all client invocations of VRESCo core services must pass the Access Control Layer (ACL). Basically, authentication is then done twofold: using certificates and username/password credentials. Before any core service can be invoked, a secure communication channel between service requester and VRESCo host must be established. This is done using X.509 certificates [67] and HTTPS (i.e., X.509 certificates are associated with every port where VRESCo core services are running). However, the channel can only be established if both communication parties trust each other's certificates. Therefore, in step 1–3 the certificates of client and service are verified by the other side (we assume that the certificates have been exchanged before the first invocation). The client has to trust the service certificate, while the *Certificate Validator* verifies if the client's certificate is in the certificate store (step 3). If this is not the case, an exception is returned to the requester and the requested core service is not executed. It should be noted that the use of certificates additionally enables to encrypt all messages which is provided as built-in functionality by the WCF platform [101].

In addition to authentication based on certificates, the VRESCo runtime supports username-password credentials, which follows the WS-Security specification [139]. For every invocation, these credentials are attached to the SOAP message, which is done transparently by the Client Library (step 4). The *Username/Password Validator* then verifies if these credentials match to the one's stored in the Registry Database (step 5). As before, if they do not match an exception is returned to the requester and the requested core service is not executed. As a result, after executing steps 1 to 5 client and service are authenticated and both sides know the identity of the communicating party. In the next section, we show how claim-based authorization is then applied in VRESCo.

3.8.2. Claim-based Authorization

Authentication and authorization for Web services have been addressed by various research efforts (e.g., [18, 19, 49]) and specifications such as WS-Security [139]. In general, access control is often done role-based where different roles are assigned to users, while security privileges are directly granted to these roles. In our work, we follow the concept of claim-based access control that goes one step further: Claims can be defined on different resources (e.g., following the CRUD operations *Create*, *Read*, *Update* & *Delete*) for users and groups. Users are allowed to access resources if they provide the needed claim in their credentials. This includes all claims that belong to a specific user, while users also inherit the claims assigned to their user group.

Resource	Resource-level	Instance-level
Category	✔	
Service	✔	✔
User	✔	✔
User Group	✔	
Claim	✔	✔
QoS	✔	

Table 3.6.: Basic Claims

Table 3.6 shows resources and their claims that have been implemented in VRESCo. We distinguish between resource- and instance-level claims: Resource-level claims apply to all instances of a resource (e.g., *Read* on all *services*), while instance-level claims refer only to a specific instance of a resource (e.g., *Update* on user *U1*). Besides having claims for the core resources *Service*, *Category*, *User* and *User Group*, the resource *Claim* defines who is allowed to create, modify and delete custom claims. Therefore, users can dynamically add claims for other resources (e.g., regarding the service metadata model). Finally, claims on *QoS* can be used to restrict access to QoS information. In addition, the *PermissionManager* claim enables to assign service instance-level claims to other users or groups. This is of particular interest when service owners want to pass claims for their services to others. Besides assigning claims manually, some claims are generated automatically when users and resources are created.

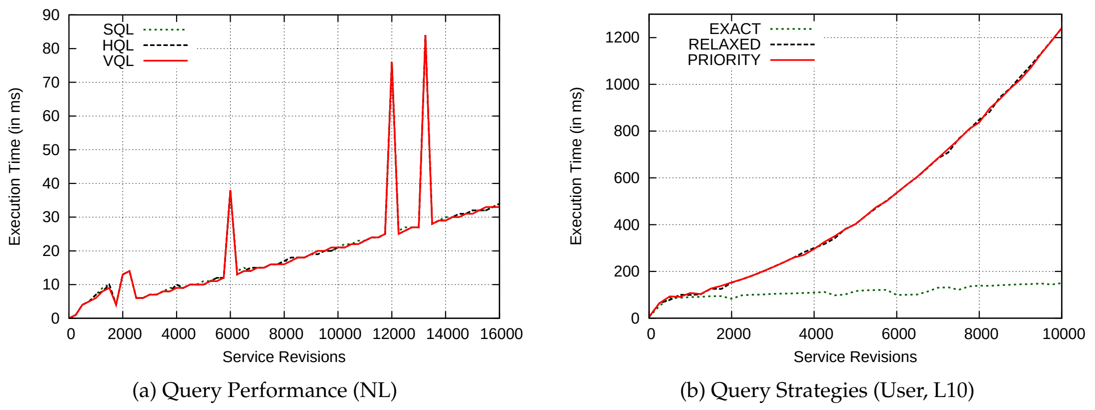
Similar to users and user groups, claims are also managed by the User Management Service and stored in the Registry Database. The VRESCo core services use the Access Control Layer (see Figure 3.12) to verify if the client has the required claims to invoke the current operation. After clients are authenticated, their identity is known and the *Claim Checker* can verify the claims in the database (step 6). If the claims are present the operation is executed (step 7), otherwise an appropriate exception is returned to the requester. To give a concrete example for such claims, the Publishing Service requires the *Create* claim on resource *Service*, while the Query Engine requires the *Read* claim on the queried resources (i.e., either the resource-level *Read* claim or the instance-level *Read* claim on instances returned by the Query Engine).

3.9. Evaluation

In this section, we give an evaluation of the VRESCo runtime focusing on the topics presented so far. The purpose of this evaluation is twofold: Firstly, we show the runtime performance regarding service querying, rebinding and mediation using synthetic data. The goal is to analyze the impact of each aspect in isolation. Secondly, we combine them into a coherent end-to-end evaluation by using an order processing workflow. The aim is to understand the influence of each aspect with regard to the overall process duration in a realistic setting. Additionally, we show how the individual results of the first part interrelate in an end-to-end setting. All experiments have been executed on an Intel Xeon Dual CPU X5450 with 3.0 GHz and 32GB RAM running under Windows Server 2007 SP1. Furthermore, we use .NET v3.5 and MySQL Server v5.1. For mediation, rebinding and end-to-end evaluation we have created different sets of test services and QoS configurations (with varying response times) using the Web service testbed GENESIS [80]. These are described in the corresponding subsections.

3.9.1. Querying Performance

First of all, we show the performance of the VQL Engine in Figure 3.13, which has been measured using the query in Listing 3.1. The test data are generated automatically: In every step, 5 categories are inserted, each having 5 alternative services with 10 revisions, while every revision has 1 tag and 11 QoS attributes with random values. In every step 20% of all services match the queried feature `Notify_Customer` and service owner *CompanyX*, while only 2% of all service revisions match all query criteria. To eliminate outliers, the results represent the median of 10 runs, while the database and Hibernate session cache are cleared after each run.



(a) Query Performance (NL) (b) Query Strategies (User, L10)

Figure 3.13.: Querying Performance

Figure 3.13a compares the performance of the queries generated by SQL, HQL and VQL. Therefore, the query from Listing 3.1 was manually translated into HQL and SQL, while the VQL query is executed on *core objects* using the *exact* strategy without result limit (NL). The queries return only the ID of the matching revisions. Therefore, this table shows the performance of the native queries and does not include the time needed for converting the results back into `ServiceRevision` objects. The results indicate that the queries generated by all three approaches perform equally. Please note that all approaches exhibit the same peaks which are due to internal processing of the database.

Figure 3.13b compares the querying strategies using the same query on *user objects* and limited to 10 results (L10). The limit was chosen since *relaxed* and *priority* return more revisions than *exact* (which influences the results). It can be seen that *exact* is much faster than *relaxed*, while *relaxed* and *priority* have similar performance. The reason for the significant difference is that *relaxed* and *priority* use different table joins and need to sum up and order by the total sum of priority values, while the query in *exact* mode can be optimized by the database.

Finally, Table 3.7 depicts the duration of the individual steps during VQL query processing. Therefore, the previous query is executed on both *core* and *user objects* using the *exact* strategy. *Generation* indicates how long the *Preprocessor* component needs to analyze and generate the query. *Execution* depicts the actual query execution time, while *Conversion* represents the time needed by the *ResultBuilder* to convert the query results.

Revisions	**User Model**			**Core Model**		
	Generation	**Execution**	**Conversion**	**Generation**	**Execution**	**Conversion**
1000	4,8	3,8	84,7	3,1	3,6	7,1
2000	4,8	14,6	87,5	3,2	14,4	6,8
3000	4,8	7,7	87,0	3,2	7,6	6,5
4000	4,8	9,8	77,5	3,2	9,7	6,5
5000	4,8	12,0	81,4	3,2	11,7	6,4
6000	4,8	13,5	83,7	3,1	13,5	7,0
7000	4,8	15,9	86,9	3,2	15,5	6,8
8000	4,8	17,9	86,3	3,2	17,6	7,3
9000	4,8	19,8	82,4	3,2	19,8	7,2
10000	4,8	22,2	86,6	3,1	20,5	6,8

Table 3.7.: VQL Query Processing (in ms, User/Core, L10)

The results show that *Generation* is almost constant for *core/user objects*, while the latter is slightly slower since queries have to be translated to refer to *core objects*. Obviously, *Execution* is almost equal for both approaches. Finally, the table indicates that *Conversion* is fast for *core objects*, while it takes some time for *user objects*. The main reason is that queries actually return IDs, while the corresponding entities are loaded from the NHibernate session context. Furthermore, revision objects have a number of collections (e.g., tags, QoS, etc.) that have to be converted by the

ResultBuilder using reflection, which internally leads to a number of additional queries (since most collections are lazy-loaded [56]). In this setting, *Conversion* is constant for all revisions within both models due to the result limit of 10.

3.9.2. Rebinding Performance

In the following subsection, we give an evaluation of the different rebinding strategies. The evaluation is done using the Web service testbed Genesis [80]. This testbed provides a mechanism to automatically deploy JAX-WS Web services which can be configured using plug-ins that simulate changing QoS attributes (e.g., response time, availability, etc.).

For measuring the rebinding performance, we used Genesis to simulate 10 services that implement the same feature. Then, we leveraged the QoS plug-in to continuously modify the response time of all services using a Gaussian distribution, and we additionally increased the variance after each step in order to simulate an environment where the QoS is subject to significant change. Finally, we implemented one client for each rebinding strategy and measured the average response time when invoking the service. As a result, we can see the impact of the different rebinding strategies for each client.

The results of this experiment are depicted in Figure 3.14. It should be noted that the response time of the best service is decreasing since we increase the variance with every step. All services start with a (server-side) execution time of 2000 ms. The (client-side) response time differs about 400 ms which is caused by the network latency and the time needed for wrapping SOAP messages. Therefore, the actual (server-side) execution time of the best service reaches 0 ms after step 30.

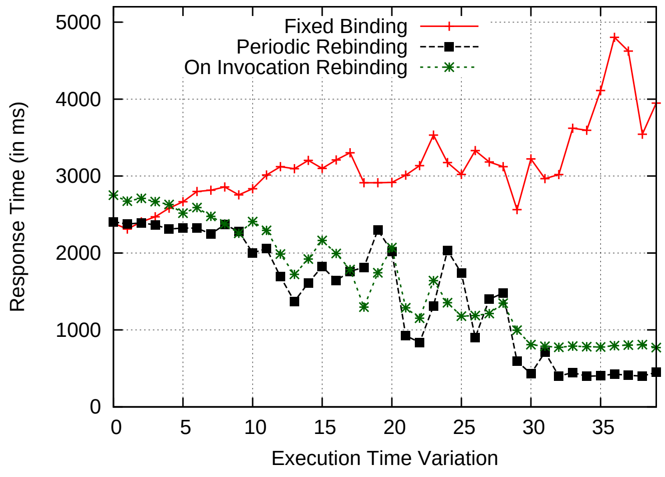


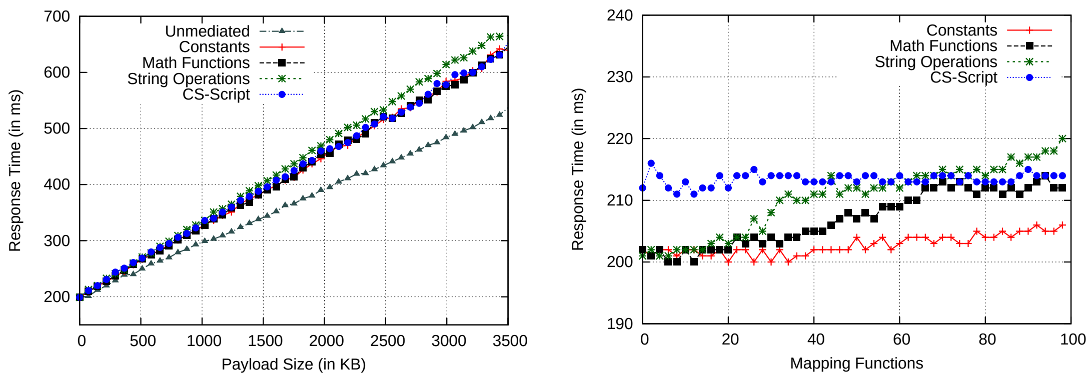
Figure 3.14.: Rebinding Strategy Performance

Obviously, clients with *Fixed* binding perceive the worst response time because the binding does not change. Clients using *Periodic* rebinding mostly use services with good response time. However, since rebinding is done in pre-defined intervals the binding is not always up-to-date (e.g., steps 17–18, 24–25 and 27–28 represent such situations). In contrast to that, clients with *OnInvocation* rebinding always invoke the best service since the rebinding is re-considered just before service invocation. However, this leads to a constant overhead of about 400 ms which is needed to check the binding and update if necessary. In this figure, the *OnDemand* strategy has been omitted since it heavily depends on when the client requests the rebinding.

3.9.3. Mediation Performance

Moreover, we have evaluated the overhead introduced by the VRESCo mediation mechanisms, which is shown in Figure 3.15. This evaluation has again been done using the GENESIS testbed.

Figure 3.15a depicts the response time of a single Web service invocation depending on the size of the message sent to the service. We have evaluated five different scenarios: (1) no mediation, (2) mediation using only constant mapping functions (i.e., replacing an input parameter with a constant string), (3) using mathematical functions (replacing a parameter with a calculated value), (4) using string modification functions (adding a constant string to a string parameter), and finally (5) using CS-Script (a simple script which exchanges the order of two parameters). Unsurprisingly, unmediated invocations are generally faster than any type of mediation. The performance of mediated invocations is similar no matter what type of mapping functions have been applied. However, in our experiments mediation using string operations introduces slightly more overhead than the other types. This is due to the fact that string operations naturally become more expensive when the strings become bigger.



(a) Message Size

(b) Mediation Steps

Figure 3.15.: Mediation Performance

In Figure 3.15b we have studied the overhead introduced by different mapping functions in more detail. We have evaluated how the overhead introduced by mediation depends on the amount of mediation necessary (measured in the number of mapping functions applied). We have evaluated the same scenarios as before, but omitted the tests using unmediated invocations. Generally, the additional overhead introduced by a larger number of mapping functions is rather small: the difference between 1 and 100 mapping functions varies between 5 and 20 ms. As before, the overhead introduced by string operations heavily depends on the size of the string. Our experimentation string was rather sizable at 73 kByte, which explains the comparatively big overhead incurred by this type of mapping function. Note that the overhead of CS-Script mappings is constantly about 10 ms. The reason is that the main overhead is the initialization of the scripting engine, while the execution of the script is usually negligible (as long as the script has no expensive computations, which is not typical for mapping scenarios).

This result differs from what we have reported earlier in [89]. In this work, we have compared various Daios mediators including one based on SAWSDL [201] 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. It should be noted, however, that the results must be compared with caution because they represent two different approaches. While VMF is integrated into VRESCo, the SAWSDL-based mediator has been implemented in the Java branch of Daios. Detailed information about the performance of VMF can be found in [70].

3.9.4. Security Performance

Furthermore, we want to briefly evaluate the security features by showing the overhead of authentication and authorization when invoking VRESCo core services. For our experiments we have used the Publishing Service to activate and deactivate service revisions in the registry. To show the effects of authentication and authorization, we have utilized different users for the service invocation (i.e., user *admin* has all access rights, while authorization has to be verified for other users). In each round, we have added an additional revision. Furthermore, we show the difference of using the resource-level *Update* claim on *Revision*, compared to instance-level claims on specific revisions (we assign two claims per revision to the user).

The results are depicted in Figure 3.16. The red line depicts the time needed for 500 activations/-deactivations without security. The black line indicates the overhead involved in authentication and transport-level message protection based on certificates. In this setting, the average overhead is about 11%. Finally, for authorization we distinguish between resource-level claims (RLC, green line) and instance level claims (ILC, blue line). In this setting, the overhead for authorization plus authentication in relation to authentication only is about 20% for RLC and about 27% for ILC. The results indicate a slight overhead when enabling security. Additionally, RLC should be preferred over ILC since performance is decreasing with the number of claims.

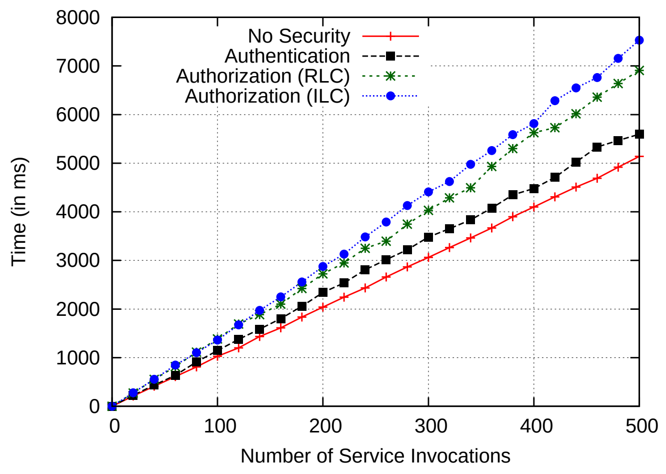


Figure 3.16.: Authentication and Authorization Performance

3.9.5. End-to-End Evaluation and Discussion

The final part of the evaluation consists of an end-to-end scenario, that combines the most important aspects introduced above (i.e., querying, rebinding, mediation and invocation). Therefore, we use a larger order processing workflow taken from the CPO case study introduced in Section 1.1. The use case represents online ordering of new cell phone contracts including mobile phone and SIM card. The workflow has been implemented in C# and consists of 19 activities which are split into 5 sequential subprocesses.

Basically, the workflow starts upon receiving customer orders via the company Web site. Firstly, the internal stock is checked for the availability of phone and SIM card. If one of those components is missing, it is ordered by using one of the internal or external supplier services. Secondly, the contracting subprocess creates a new contract and, if necessary, adds the customer to the CRM system. If the customer wants to transfer her old number, the number porting subprocess as depicted in Figure 1.2b is executed. Then, the new cell phone number is activated in the GSM network. Finally, the payment and shipping subprocesses are executed.

The services used in this case study have been deployed on a different host using Genesis [80]. For each internal service (e.g., CRM, contracting, etc.) we have deployed only one alternative, whereas, for each external service (e.g., Credit Card Service, etc.) multiple alternatives are available. More precisely, we have provided between 60–250 alternatives per service. Additionally, there are 30 alternatives for the internal Notification Service, which is used to notify customers of their order status (e.g., using SMS, email, mail, etc.). This is the only service in our experiment that requires significant mediation. Finally, we use the Genesis testbed to simulate a response time of 30–100 ms for each service.

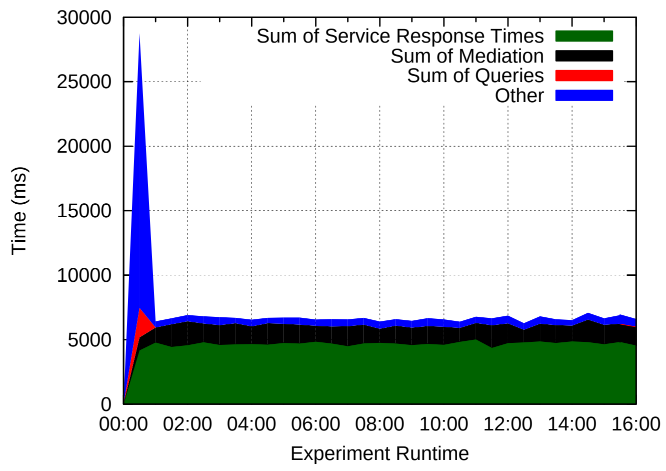


Figure 3.17.: End-to-End Performance

In Figure 3.17, we show the average process duration in this case study based on 40 concurrent clients running on one host that is also hosting the VRESCo environment. Each client continuously executes the process over an experiment time of 16 min. We have chosen the *Periodic* rebinding strategy for this scenario, to accommodate for our highly dynamic scenario with many alternatives for each external service and changing QoS properties. In order to get a big number of re-bindings during our experiment time we have chosen a rebinding interval of 5 seconds. The x-axis of the figure shows the experiment time (in minutes) and the y-axis depicts the averaged process durations of the currently executing process instances. Right after bootstrapping the system, there is a steep incline in the overall duration because each client performs some initialization. This includes querying the available services (red part), as well as creating proxies and binding to one service candidate (blue part). Additionally, the services are invoked (green part) and a certain amount of mediation occurs (black part). After the initialization phase, the system stabilizes and the response times and mediation time are constant. The mediation overhead reflects our detailed mediation results from Figure 3.15a. Together, service response times and mediation account for about 92% of the average process duration after the initialization phase. The remaining 8% (blue part) represent other factors such as thread handling or the workflow business logic. Please note that querying and occasional rebinding still happens after the initialization phase, but it is no longer part of the average process execution time (on the y-axis). This is because the rebinding clients perform querying and rebinding asynchronously in a separate thread.

Therefore, it solely depends on the rebinding strategy whether querying and rebinding is part of the process execution time or just part of the initialization phase. In case of *OnInvocation*, there would be querying and rebinding overhead in the overall process execution time, whereas for *OnDemand* the behavior would be similar as shown above.

Generally, the decision which rebinding strategy to use depends on the particular domain and the requirements. For example, for the number porting service *fixed* binding is not a reasonable choice because even simple changes of the partner CPO's services (e.g., a different endpoint) would break the process. *OnDemand* is only reasonable if changes happen infrequently, and adaptation to changes is not time-critical. *Periodic* rebinding, on the other hand, is only adequate when services change frequently enough to warrant permanent polling for updates. Since number porting is not time-critical, we could have also used the *OnInvocation* rebinding strategy which has a constant invocation overhead but always finds the best available service.

3.10. Related Work

In this section, we want to give an overview of related work, with an emphasis on service metadata and querying, versioning and dynamic binding/invocation of Web services.

- **Service Metadata and Service Querying:** Besides UDDI and ebXML, there are other standards for describing service metadata [23]. Some of them are used by semantic Web service approaches [105], such as OWL-S [193], WSMO [196] and SAWSDL [201]. It should be noted, however, that the VRESCo service metadata model introduced in Section 3.2 is not intended to compete with these approaches. We aim at enterprise development where metadata is an important business asset which should not be accessible for everyone, as opposed to the semantic Web service community where domain ontologies should be public to facilitate integration among different providers and consumers.

  In general, several standards and research approaches have emerged that address the complexities of managing and deploying Web services [209]. In these approaches, service querying and selection play a crucial role, especially regarding service composition (e.g., [64, 159, 212]). However, the query models of current registries and Web service search engines [149] mainly focus on keyword-based matching of service properties which often do not cover the rich semantics of service metadata.

  Yu and Bouguettaya [208] introduce a Web service query algebra and optimization framework. This framework is based on a formal model using service and operation graphs that define a high-level abstraction of Web services. Besides functional service descriptions, they present a QoS model that distinguishes between *runtime quality* (latency, reliability and availability) and *business quality* (fee and reputation). Service queries are specified as algebraic operators on functionality, quality and composition of services, and finally result in service execution plans. Optimization techniques are then applied to select the best service execution plan according to user-defined QoS properties. This work is complementary to ours: While the authors focus on their formal service model and introduce a query algebra for this model, we present a service runtime that provides end-to-end support for service management and querying functionality.

Furthermore, there is plenty of research in the area of service discovery (e.g., [86,176]). The authors distinguish between three types of service discovery. *Early service discovery* occurs in the requirements engineering phase and is driven by the requirements specification. *Architecture-driven service discovery* is done during the design phase and is driven by the specification of functionality, quality attributes and constraints. Finally, *runtime service discovery* deals with the discovery and replacement of services at runtime. In contrast to our work, the approach presented in [176] focuses on runtime monitoring the compliance of service-centric systems to requirements and discovering alternative services at runtime, whereas dynamic binding is not explicitly addressed.

- **Versioning:** The evolution of Web services is subject to a wide ranging debate. The W3C recently paid attention to versioning of Web services [200, 203]. Currently, a common workaround to deal with the lack of versioning support is to use separate namespaces for each version of a service. The general problem of versioning of distributed software systems is sketched in [188] where the author distinguishes between *Distributed-Object Versioning* and *Messaging Versioning*. Even more generally, Web service evolution can be considered a special case of the Software Configuration Management (SCM) problem [33]. Therefore, we have adopted many notions from SCM (e.g., revisions, branching and merging, revision tagging) in our approach.

  Current registry standards provide only little support for evolving Web services. The ebXML versioning approach [134] is based upon the versioning extensions of WebDAV [30], but provides only a small subset of this functionality. If the ebXML registry supports versioning, all registry items are implicitly under version-control. However, it should be noted that ebXML mainly focuses on versioning of registry data but it remains unclear how clients can access specific service revisions. In contrast to ebXML, the UDDI specification [136] does not mention versioning at all. One common approach for versioning in UDDI is that a given version of a `wsdl:portType` is represented by a unique `tModel`. Current best practices for service versioning using UDDI are described in [25].

  One approach that addresses Web service evolution has been introduced by Kaminski et al. [81]. The authors outline various requirements for versioning, and demonstrate why common versioning strategies are inappropriate in the context of Web services. Instead they propose to use the *Chain of Adapters* pattern [57] for developing evolving Web services. Ponnekanti et al. [151] address the interoperability among independently evolving Web services by specifying four types of incompatibilities that may arise: *structural*, *value*, *encoding*, *semantic*. However, the paper mainly focuses on structural and value incompatibilities. The authors introduce static and dynamic analysis algorithms to identify compatibility between applications and non-native services, and present tools that implement these algorithms. Moreover, they introduce so called *cross stubs* for resolving incompatibilities. These cross stubs are generated semi-automatically and mediate the interaction between application and target service at runtime.

- **Dynamic Binding, Invocation and Mediation:** Pautasso and Alonso [145] discuss binding models for (Web) services, as well as different points in time when the bindings are evaluated. The motivation of their work is the shortcoming of current composition languages such as WS-BPEL [140]. In WS-BPEL, dynamic binding is supported by reassigning endpoints using the `partnerLink` construct. Endpoints represent specific ports of a service interface at runtime which are usually identified using WS-Addressing [194]. However, dynamic binding in WS-BPEL can only be achieved if the interfaces of the different services are identical, which limits the flexibility of this approach. The authors present a flexible binding model using their JOpera system. In this approach, binding is done using reflection and therefore does not require a specific language construct.

  Di Penta et al. [40] present the WS-Binder framework for enabling dynamic binding within BPEL processes. They distinguish between three different types of binding mechanisms: *Pre-execution workflow global binding* occurs prior to the execution of a composition and is done using genetic algorithms. *Run-time local binding* allows to select service bindings while the composition is already running. Finally, *run-time workflow slice re-binding* stops the execution of the composition in case of an error (e.g., service is not available or QoS values are not as desired), and determines the workflow slice still to be executed using global binding. This approach is built on top of WS-BPEL and uses proxies to separate the abstract services with the concrete service instances.

  Apache Web Service Invocation Framework (WSIF) [3] represents the first Java-based dynamic invocation framework for Web services. However, WSIF has some drawbacks compared to other frameworks. For instance, it provides only weak support for using complex XML schema types in parameters (input or output) since they have to be mapped to existing Java classes before invocation. Furthermore, the runtime performance compared to other frameworks such as Daios, Apache CXF [7], Apache Axis2 [6], or Codehaus XFire [31] is significantly worse (see [92]). Finally, WSIF has not been under active development since 2003. In contrast to Daios, the frameworks above rely on static components for accessing Web services, and provide little dynamic invocation support.

  There are several research efforts addressing service mediation. Similar to our client-side approach, others use adaptors to resolve interface incompatibility (e.g., [16, 99]). These adaptors are conceptually similar to our mediators, but are more decoupled from the clients. Furthermore, service mediation is also addressed in semantic Web services. For instance, the Web Service Execution Environment (WSMX) [195] provides a mediation architecture based on WSMO. However, semantic approaches usually rely on shared ontologies and explicit semantic information. Others have addressed mediation on the service composition level. For instance, Moser et al. [123] adapt BPEL processes by exchanging service bindings at runtime, while compatibility of services is realized using XSLT transformations. Finally, industry solutions often address mediation at the ESB level [167] by using XSLT-based transformation on SOAP messages.

3.11. Conclusion

In this chapter, we have introduced the VRESCo runtime environment that aims at addressing some of the current challenges in SOC. Among others, this includes service metadata and service querying, QoS-based dynamic binding and invocation, as well as service mediation. Furthermore, we also addressed service versioning and security mechanisms to provide authentication and authorization.

One of the main goals of VRESCo is to provide an environment for transparent binding of services that perform the same task but have different technical interfaces. Therefore, we have introduced a service metadata model following a feature-driven service abstraction. Furthermore, a querying framework has been proposed to query services and associated metadata using different querying strategies. Finally, service mappings are defined to map services with different interfaces, while the mediation framework is used to execute these mappings. As a result, clients can seamlessly access alternative services by automatically rebinding to these alternatives. For instance, this can be desired if new service revisions are published or old services are deleted, or if alternative services provide better QoS.

The evaluation has shown that the performance of the different components (i.e., querying, mediation and security) is adequate for the expected number of services. Furthermore, we also depicted the effects of using different rebinding strategies. Finally, the end-to-end evaluation has discussed the effects of the different components using a scenario from the motivating example introduced in Section 1.1.

Chapter 4.

Service Notification Engine

This chapter describes the VRESCo event notification support in detail. A brief overview of the necessary background was given in Section 2.2. After highlighting the motivation of this work in Section 4.1, the architectural overview of the eventing mechanism is shown in Section 4.2. The details of this eventing mechanism are described in the following sections. This includes event types (Section 4.3), event participants (Section 4.4), subscription and notification mechanisms (Section 4.5), event search (Section 4.6), event ranking (Section 4.7) and event correlation (Section 4.8). Furthermore, event access control using event visibility is introduced in Section 4.9. Section 4.10 evaluates the eventing support regarding the expressiveness of the subscription language and the performance of the Event Engine, as well as a brief software demonstration. Finally, Section 4.11 describes related work in this area and Section 4.12 concludes the chapter.

Contents

4.1. Motivation

In Chapter 1, we have introduced several SOC challenges such as metadata, querying and versioning, as well as dynamic binding, invocation and mediation of services. One reason for these issues is represented by the fact that services, associated service metadata and QoS attributes change regularly. However, service consumers are not aware of these changes, and cannot automatically react to service and environment changes (e.g., by adapting their service-based applications). This raises the need for appropriate event notification mechanisms.

Notifying subscribers when events of interest occur has been addressed by the Publish/Subscribe paradigm [47] in general, and event-based systems [124] in particular. Cugola and Di Nitto [34] give a detailed overview of approaches that combine Publish/Subscribe and SOA. The most popular examples are WS-Notification [138] and WS-Eventing [199]. Additionally, UDDI [136] and ebXML [134] introduce limited support for event notifications in registries.

As discussed in Chapter 2, we see three main challenges in existing approaches. Firstly, the notifications provided by service registries mainly focus on basic service management events. In contrast to this, additional runtime information concerning service invocations and QoS should also be taken into consideration. We argue that such notifications are equally important and should, therefore, be provided by service runtime environments. Secondly, most existing approaches have in common that subscriptions can only address single events (e.g., new service is published). In practice, however, it is often desired to detect more complex events (e.g., average QoS within some timeframe is beyond some threshold). Therefore, complex event processing mechanisms [102] such as sliding window operators and event patterns are needed. Thirdly, some approaches do not store the event history but discard events as soon as all subscribers have been notified. However, we think that users should be enabled to query for historical events since this can be of great interest (e.g., during service selection).

Considering our motivating example from Section 1.1, there are several use cases for simple and complex subscriptions. On the one hand, CPOs want to know if new services are published or existing services are deleted by partners and competitors. On the other hand, the QoS of external services is also of interest. For instance, if the average response time of some service goes beyond a certain threshold, rebinding to an alternative service may become necessary.

4.2. Architectural Overview

This section gives a high-level overview of the VRESCo event notification support, while the details are described in the remainder of this chapter. The basic idea can be summarized as follows: Notifications are published within the runtime if certain events occur (e.g., service is added, user is deleted, etc.). Service consumers are then enabled to subscribe for notifications about the occurrence of these events.

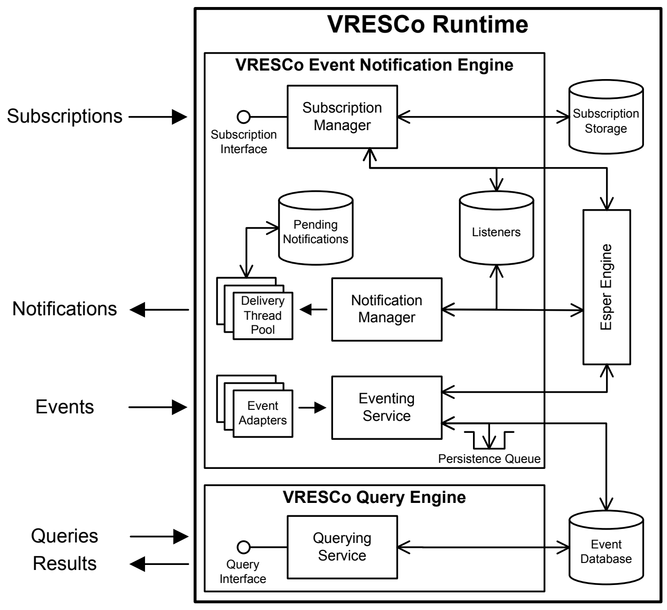


Figure 4.1.: VRESCo Eventing Architecture

Figure 4.1 depicts the architecture of the Notification Engine which represents one component of the VRESCo runtime shown in Figure 3.1. Therefore, it is also implemented in C# on the .NET platform. The event processing functionality is based on NEsper, which is a port of the event processing engine Esper [45]. Within the Notification Engine, events are published using the Eventing Service. Most events are produced by the corresponding VRESCo core services (e.g., user management events are fired by the User Management Service while metadata events are fired by the Metadata Service). Therefore, these services send their events directly to the Eventing Service for publication, which is illustrated in Figure 3.1 using vertical lines between VRESCo core services and Notification Engine. In contrast to this, some events (e.g., related to QoS) are produced by external components (e.g., QoS Monitor), which use the external event interface for publication. Event adapters are thereby used to transform incoming events into the internal event format for further processing. The Eventing Service then forwards these events to the Persistence Queue, which is responsible for persisting events in the Event Database. This is done using the ORM Layer introduced in the previous chapter. Finally, the Eventing Service feeds incoming events into the Esper Engine, which is responsible for event processing.

The Subscription Interface is used for subscribing to events of interest according to the interfaces proposed in the WS-Eventing specification [199]. The Subscription Manager is then responsible for managing subscriptions, which are put into the Subscription Storage. In addition, subscriptions are translated for further processing, which is done by converting the WS-Eventing subscriptions into listeners that can be attached to the Esper Engine.

Esper performs the actual event processing and is, therefore, responsible for matching incoming events received from the Eventing Service to listeners attached by the Subscription Manager. On a successful match, the registered listener informs the Notification Manager, which is responsible for notifying interested subscribers using a thread pool. Depending on the listener type, the Notification Manager knows which mechanism to use (e.g., email, Web service, etc.).

Finally, the Query Interface on the bottom is used to search for historical events using the VQL Engine (see Section 3.5), which returns all events that match a given VQL query. The Event Database is implemented using a relational database and accessed via the ORM Layer.

4.3. Event Types

The first step in developing such an eventing mechanism is to define all event types which are supported by the engine. In the context of our work several events can be captured at runtime. We have identified various event types and show the most important types in Figure 4.2.

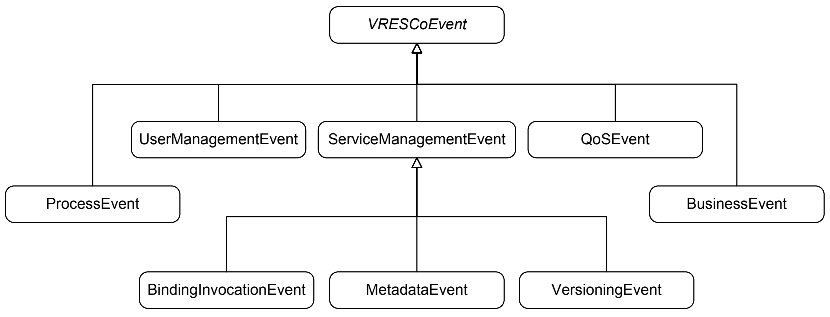


Figure 4.2.: Event Type Hierarchy

In general, the event types form a type hierarchy following the concept of class hierarchies (i.e., events inherit the properties of their parent type). As shown in the figure, all events inherit from the base type *VRESCoEvent*, which provides a unique event sequence number and a timestamp measured during event publication. For efficient processing of events, VRESCo uses its own event format implemented as C# classes. These event classes consist of name-value pairs which are often used in event-based systems since they support efficient content-based filtering of events. According to the type-based approach the event classes are part of an event hierarchy as described above. Furthermore, in addition to simple name-value pairs the event classes may also include non-primitive data types.

In the following, we describe all event types in more detail. It should be noted, that this list is not intended to be exhaustive, but rather contains all events currently provided by VRESCo. However, additional event types can be easily integrated into the runtime environment, which makes the eventing infrastructure extensible.

4.3.1. Service Events

Since VRESCo represents a service runtime environment, the most important events are obviously those related to service management. These events are summarized in Table 4.1 where events are grouped according to their type, while the event condition in the right column describes the situations when the event occurs.

Event Type	Event Name	Event Condition
ServiceManagementEvent	ServicePublishedEvent	Service is published to the runtime
	ServiceModifiedEvent	Service is updated in the runtime
	ServiceDeletedEvent	Service is deleted from the runtime
	ServiceActivatedEvent	Service is activated in the runtime
	ServiceDeactivatedEvent	Service is deactivated in the runtime
VersioningEvent	RevisionPublishedEvent	Revision is published to the runtime
	RevisionActivatedEvent	Revision is activated in the runtime
	RevisionDeactivatedEvent	Revision is deactivated in the runtime
	RevisionTagAddedEvent	Revision tag is added by the owner
	RevisionTagRemovedEvent	Revision tag is removed by the owner
MetadataEvent	ServiceCategoryAddedEvent	Category is added to the runtime
	ServiceCategoryModifiedEvent	Category is modified
	ServiceCategoryDeletedEvent	Category is deleted from the runtime
	FeatureAddedEvent	Feature is added to a category
	FeatureModifiedEvent	Feature is modified
	FeatureDeletedEvent	Feature is deleted from a category
	OperationAddedEvent	Operation is added to the runtime
	OperationModifiedEvent	Operation is modified
	OperationDeletedEvent	Operation is deleted from the runtime
	MappingAddedEvent	Operation is mapped to a feature
	MappingDeletedEvent	Operation-Feature mapping is deleted
BindingInvocationEvent	ServiceInvokedEvent	Service has been invoked
	ServiceInvocationFailedEvent	Service invocation has failed
	ProxyRebindingEvent	Service proxy is (re-)bound to a service
QueryingEvent	RegistryQueriedEvent	Registry is queried using query string
	ServiceFoundEvent	Specific service is found by a query
	NoServiceFoundEvent	No services are found by a query

Table 4.1.: Service Events

To start with, basic service management events record if services are published, modified, activated, deactivated or deleted in the Registry Database. Furthermore, the VRESCo versioning mechanism introduced in Section 3.4 is also captured by events. This includes the creation, activation and deactivation of service revisions. Furthermore, adding and removing of revision tags also triggers corresponding events.

Besides service management, service metadata is another important topic covered by the eventing mechanism. This mainly includes events that record if entities in the VRESCo metadata model are created, updated or deleted (e.g., service categories, features, operations, etc.). In addition, events also record if mappings between operations and features (as described in Section 3.2.3) are created and deleted. Furthermore, events regarding binding and invocation of services are also of particular interest. In VRESCo these events contain both the username and IP address of the service requester, as well as the returned exception message, if the service invocation has not been successful. Additionally, events are raised if service proxies successfully rebind from one service to another. Finally, events regarding querying are listed for historic reasons. They have been disabled by default in the meanwhile, since they happen very frequently but there are only few scenarios where these events are of interest.

4.3.2. QoS Events

Quality of Service represents an important issue in service-oriented systems, especially during service discovery and dynamic rebinding. If there are multiple services candidates, consumers often want to invoke the service having the "best" quality. In VRESCo, QoS events (see Table 4.2) are used to record the current quality attributes of services. Thereby, we distinguish between QoS values of services (*QoSEvent*), service revisions (*QoSRevisionEvent*) and service operations (*QoSOperationEvent*). For instance, some QoS properties are defined on the operation-level (e.g., response time), while others are defined on the revision- or service-level (e.g., security).

Event Type	Event Name	Event Condition
QoSEvent	QoSEvent	QoS value of service is published
	QoSRevisionEvent	QoS value of service revision is published
	QoSOperationEvent	QoS value of service operation is published
	RevisionGetsUnavailableEvent	Service revision gets unavailable
	RevisionGetsAvailableEvent	Service revision gets available again

Table 4.2.: QoS Events

The QoS events reflect the QoS model described in Section 3.3. In Section 5.3 we will show how QoS values are actually measured in VRESCo, how QoS events are used to combine different monitoring techniques, and how this can be further used to monitor the compliance to SLAs.

4.3.3. Process Events

According to the SOA model, services are often invoked as part of business processes which can be represented as composition of services. One common standard for SOA-based business

Event Type	Event Name	Event Condition
WorkflowTrackingEvent	WorkflowAbortedEvent	Workflow has been aborted
	WorkflowChangedEvent	Workflow has been changed
	WorkflowCompletedEvent	Workflow has been completed
	WorkflowCreatedEvent	Workflow has been created
	WorkflowExceptionEvent	Workflow execution has raised an exception
	WorkflowIdleEvent	Workflow is idle
	WorkflowLoadedEvent	Workflow has been loaded
	WorkflowPersistedEvent	Workflow has been persisted
	WorkflowResumedEvent	Workflow has been resumed
	WorkflowStartedEvent	Workflow has been started
	WorkflowSuspendedEvent	Workflow has been suspended
	WorkflowTerminatedEvent	Workflow has been terminated
	WorkflowUnloadedEvent	Workflow has been unloaded
ActivityTrackingEvent	ActivityInitializedEvent	Activity has been initialized
	ActivityExecutingEvent	Activity has been executed
	ActivityCancelingEvent	Activity has been canceled
	ActivityFaultingEvent	Activity has been faulted
	ActivityClosedEvent	Activity has been closed
	ActivityCompensatingEvent	Compensating activity has been executed
UserTrackingEvent	UserTrackingEvent	Track business- or workflow-specific data

Table 4.3.: Process Events

processes is WS-BPEL [140]. Such processes can raise events which are often called process or workflow events (e.g., some activity has been invoked, the process has completed, etc.).

The process events shown in Table 4.3 are based on the events provided by the Windows Workflow Foundation (WF) [171]. Basically, these events record the lifecycle of workflows (e.g., *Created*, *Started*, *Completed*, etc.) and workflow activities (e.g., *Executing*, *Compensating*, etc.). Furthermore, business- or workflow-specific data can also be tracked using *UserTrackingEvents*. Section 4.4 describes in more detail how these events are handled in VRESCo.

4.3.4. User Events

Another type of events concern the management of VRESCo users, which mainly consists of the three basic operations *create*, *update* and *delete*. Furthermore, events are raised when users login to or logout from the Runtime Manager GUI.

Table 4.4 summarizes all user events currently available in VRESCo. These events seem to be of minor importance at first glance but there are some scenarios where they are useful. For instance, some users may be interested if user details (e.g., postal address) of partners change.

Event Type	Event Name	Event Condition
UserManagementEvent	UserAddedEvent	User is added to the runtime
	UserModifiedEvent	User is modified
	UserDeletedEvent	User is deleted from the runtime
	UserLoginEvent	User login using the GUI
	UserLogoutEvent	User logout using the GUI

Table 4.4.: User Events

4.3.5. Business Events

The final group of events is represented by business events. Similar to *UserTrackingEvents*, such events can be used to publish user- or business-specific data which is not covered by the existing event types. In general, business events leverage the extensibility of the VRESCo eventing mechanism.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <esper-configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xmlns="http://www.espertech.com/schema/esper"
4     xsi:noNamespaceSchemaLocation="esper-configuration-3-0.xsd">
5   <!-- ... -->
6   <event-type name="VRESCoEvent" class="VRESCo.Contracts.Eventing.VRESCoEvent"/>
7   <!-- ... -->
8 </esper-configuration>
```

Listing 4.1: Event Type Configuration

To be more concrete, these events have to be defined as C# classes and configured in XML configuration files, which is currently done at design-time. As described in Section 2.2, the C# classes representing the events must provide *getter* and *setter* methods for the event properties. Furthermore, the event classes must be part of the event type hierarchy (e.g., inherit from the base class *VRESCoEvent*). Finally, the XML configuration file defines the mapping between name of the event and event class as shown in Listing 4.1 (line 6).

4.4. Event Participants

Event-based systems usually consist of two types of participants which pose different requirements to the system, namely event producers and event consumers (also referred to as event source and sink). The following section describes the event participants in more detail.

4.4.1. Event Producers

In general, events are produced by various VRESCo components. However, different components are responsible for firing different kinds of events. In this section, we describe these components which mainly differ in their location. In this regard, we distinguish between *internal events* which are produced within the VRESCo runtime (i.e., by the core services) and *external events* which are published from components outside the runtime (e.g., service proxies).

- **VRESCo Services:** Most events are directly produced by the corresponding VRESCo core services as indicated in Figure 3.1 using vertical arrows. To give an example, service management events (e.g., *ServicePublishedEvent*) are fired by the Publishing Service. The same is true for versioning events, while metadata events are raised by the Metadata Service. Consequently, user management events are published by the User Management Service. All these event types have in common that they are produced as part of the VRESCo core services and thus represent internal events. As described in Section 4.2, the events are then forwarded to the Eventing Service which triggers the event processing mechanisms. To accomplish this, the Eventing Service first adds a unique sequence number and the timestamp of publication to the event payload, before publishing them into the Esper Engine that performs the actual matching.
- **Service Proxies:** The application logic inherent to binding and invocation of services is located in the service proxies that are provided by the Client Library (see Figure 3.1). As a result, events concerning binding and invocation (e.g., *ServiceInvokedEvent*, *ServiceInvocationFailedEvent*, etc.) are fired by this component. Therefore, VRESCo provides an external event interface (as part of the Management Service) in order to allow clients to feed binding and invocation events into the runtime. These client events represent external events which are then transformed into the internal event format.
- **QoS Monitor:** QoS events are published by the QoS Monitor, which regularly measures the QoS values of services. These events contain the id of the service/revision/operation (depending on the type of *QoSEvent*), the name of the QoS property (e.g., response time), and the measured value (e.g., 100 ms). Similar to the Client Library, the QoS Monitor uses the external event interface to feed external events into the runtime. In Section 5.3 we present two different QoS monitoring techniques in more detail. We also show how QoS events are used to publish current QoS values and monitor SLA violations.
- **Workflow Engine:** Process events are based on the VRESCo Composition Engine introduced in [158–160]. Compositions in VRESCo are defined in a domain-specific language called Vienna Composition Language (VCL). The overall idea is to define functional and QoS constraints which can make use of constraint hierarchies by leveraging hard (i.e., required) and soft (i.e., optional) constraints. The Composition Engine then tries to find an optimal solution for these constraints semi-automatically. Finally, the optimized compositions are executed using Windows Workflow Foundation (WF) [171].

The basic idea of process events is to trace workflow executions. Therefore, WF provides a powerful tool called Tracking Service [78], which enables to hook into the workflow engine to receive certain events (see Table 4.3). Moreover, tracking profiles are used to define event filters, while matching events are sent to the users using tracking channels.

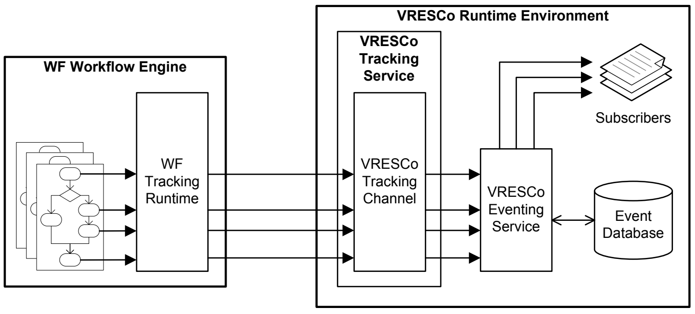


Figure 4.3.: Process Event Tracking

To integrate these workflow events into VRESCo, we have implemented our own Tracking Service and Tracking Channel as shown in Figure 4.3. The Tracking Service reads the tracking profile from a configuration file and listens to all events that match this profile (in our case all WF workflow events). When such events are published by the Workflow Engine, they are sent to the Tracking Channel and finally forwarded to the Eventing Service that feeds them into Esper. In addition to notifying interested subscribers, the events are persisted into the Event Database. As a result, users can subscribe to and search for workflow events in the same way as for all other VRESCo events.

Please note that event producers may use different mechanisms for event publication. More precisely, internal events are published by components within the runtime (e.g., VRESCo core services, Workflow Engine) by directly interacting with the Eventing Service. The latter triggers event publication using Esper and event persistence using the Persistence Queue. In contrast to this, events of external components (e.g., QoS Monitor, service proxies) are published using the external event interface and event adapters. They are transformed into internal events and forwarded to the Eventing Service to perform the same processing steps.

4.4.2. Event Consumers

Similar to event producers, we distinguish between *internal* and *external consumers*. Internal consumers reside within the VRESCo runtime and register listeners at the Esper Engine which are invoked when subscriptions match incoming events. External consumers outside the runtime are notified depending on the notification delivery mode defined in the subscription request. External consumers can be further divided into *humans* and *services*. Clearly, notification delivery and notification payload differ for these two groups.

- **Humans:** Humans are mainly interested in notifications sent per email, SMS or other technologies such as news feeds (e.g., RSS [165], Atom [166]). In some scenarios, it may also be suitable to log the occurrence of events in log files which are regularly checked by the system administrator. In any case, notifications for humans may be less explicit since humans can interpret incomplete information. Our current VRESCo prototype supports human notifications using email. Furthermore, basic state changes of the registry content can be published using Atom.
- **Services:** In contrast to this, service notifications are mainly sent using the specifications WS-Eventing and WS-Notification (clearly, emails/feeds can also be used to some extent). For our current prototype, we have enhanced the WS-Eventing specification since it represents a lightweight approach supporting content-based subscriptions. The integration and enhancements of WS-Eventing are described later.

The notification payload may differ for humans and services. While services need exact information about the event type or the context in which an event occurred, the notification payload for humans may be less verbose. In addition, notifications for humans do not necessarily have to adhere to standardized formats or rules.

Finally, another distinction can be made between service providers and service consumers since they may be interested in different types of events. For instance, service consumers may not be interested in user management events or may even not be allowed to receive them. Section 4.9 discusses access control mechanisms for events in more detail.

4.5. Subscription and Notification Mechanisms

After discussing event types and participants, this section goes into the details of the subscription and notification mechanisms. The former is needed to declare interest in events and define how notifications should be sent, while the latter shows how event notifications are finally sent to the subscribers.

4.5.1. Subscription Mechanism

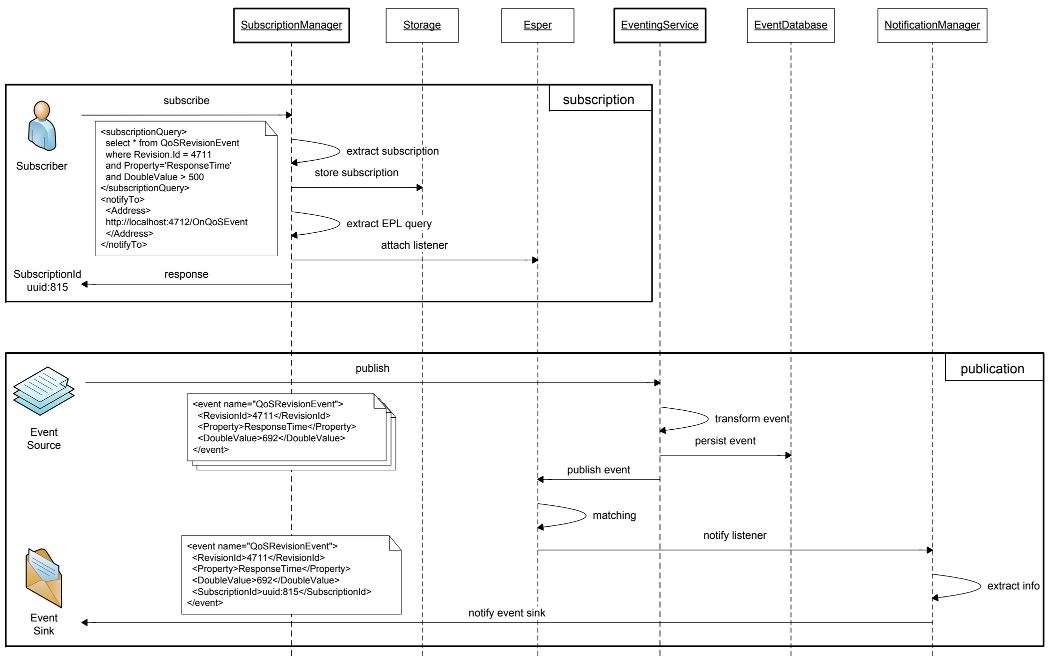
In general, event consumers can be enabled to subscribe to their events of interest in several ways [47]. The most basic way is following the topic-based style which uses topics to classify events. Event consumers subscribe to receive notifications about that topic. Similar to topic-based subscriptions, the type-based style uses event types for classification. Even though these two styles are simple, they do not provide fine-grained control over the events of interest. Therefore, the content-based style can be used to express subscriptions based on the actual notification payload.

Since the VRESCo runtime is provided using Web service interfaces, the Subscription Interface should also be exposed as Web service. WS-Eventing [199] represents a lightweight specification that defines such an interface by providing five operations: *Subscribe* and *Unsubscribe* are used for subscribing and unsubscribing. The *GetStatus* operation returns the current status of a subscription, while *Renew* is used to renew existing subscriptions. Each subscription has a given duration specified by the *Expires* attribute. Finally, *Subscription End* is used if an event source terminates a subscription unexpectedly. More information on WS-Eventing and alternative specifications can be found in Section 2.

For the implementation of the VRESCo event processing mechanism, we build on an existing WS-Eventing implementation [85] which was extended for our purpose. WS-Eventing normally uses XPath [192] message filters as subscription language which are used for matching incoming XML messages to stored subscriptions. However, these message filters only refer to single event messages and cannot describe filters on event sequences as provided by our approach. Fortunately, the specification defines an extension point to use other filter dialects which we have used to introduce the *EPLDialect* for using Esper EPL queries as powerful subscription language. Using this mechanism, the actual EPL query is attached to the subscription message by introducing a new message attribute *subscriptionQuery*.

The WS-Eventing specification distinguishes between subscriber (the entity that defines a subscription) and event sink (the entity that receives the notification), which are both implemented using Web services. VRESCo additionally supports notifications sent per email and written to log files. Therefore, in addition to the default delivery mode *PushDeliveryMode* using Web services, we have introduced *EmailDeliveryMode* and *LogDeliveryMode* which are also attached to the subscription messages.

The subscription process is illustrated in Figure 4.4, where the VRESCo eventing components with public interfaces are depicted bold. When the Subscription Manager receives a *Subscribe* request, it first extracts the subscription and puts it into the Subscription Storage (to be able to retrieve it later). Then it extracts the EPL subscription query and the delivery mode from the request and creates a corresponding Esper listener. This listener is finally attached to the Esper Engine to be matched against incoming events. Furthermore, the Subscription Manager is responsible for keeping the subscriptions in the storage and the listeners attached to Esper synchronized. That means, when subscriptions are renewed or expire, the Subscription Manager re-attaches the corresponding listener or removes them, respectively. Finally, the Subscription Manager sends the response to the *Subscribe* message to the requester. This message contains the subscription identifier implemented as UUID [88] and the expiration date of the subscription. The identifier can be used to get the status or renew subscriptions, or to unsubscribe. Please note that for brevity the figure shows only simplified versions of the request and response message, while the complete XML sample messages can be found in Listing B.1 and B.2 in Appendix B.



```
<subscriptionQuery>
  select * from QoSRevisionEvent
  where Revision.Id = 4711
  and Property='ResponseTime'
  and DoubleValue > 500
</subscriptionQuery>
<notifyTo>
  <Address>
  http://localhost:4712/OnQoSEvent
  </Address>
</notifyTo>
```

```
<event name="QoSRevisionEvent">
  <RevisionId>4711</RevisionId>
  <Property>ResponseTime</Property>
  <DoubleValue>692</DoubleValue>
</event>
```

```
<event name="QoSRevisionEvent">
  <RevisionId>4711</RevisionId>
  <Property>ResponseTime</Property>
  <DoubleValue>692</DoubleValue>
  <SubscriptionId>uuid:815</SubscriptionId>
</event>
```

Figure 4.4.: VRESCo Subscription and Event Publication

4.5.2. Notification Mechanism

Sending notifications can be done in several ways: In the best-effort model, notifications are lost in case of communication errors. To prevent such loss, subscribers may send acknowledgements on receiving notifications. Besides pushing notifications towards the interested subscribers, pull-style notifications enable subscribers to retrieve pending notifications.

VRESCo notifications are generally sent push-style using email or listener Web services. As shown in Figure 4.4, the Notification Manager knows which notification delivery mode to use depending on the listener attached to the Esper Engine. On a successful match the Notification Manager first extracts this information from the listener. In addition, the subscription identifier is added to the notification message so that the event sink can correlate subscriptions and notifications. If the event sink prefers email notifications, the Notification Manager connects to an SMTP server. In case of Web service notifications, the Notification Manager invokes the corresponding listener Web service provided by the event sink. In both cases, a dedicated delivery thread pool is used to inform the interested subscribers. Additionally, if the event sink cannot be notified (e.g., the listener Web service is offline), pending notifications are stored in the Event Database and can be retrieved by the subscribers in pull-style. The figure again shows only a simplified version of the notification message, while the complete XML sample notification message can be found in Listing C.1 in Appendix C.

4.6. Event Persistence and Event Search

Event notifications are often used when subscribers want to quickly react on state changes. Additionally, in many situations it is also important to search in historical event data. For instance, users may want to get notified if a new service revision is published into the registry, while they also want to search for the QoS history of services. To support such functionality, the Notification Engine stores all events in the Event Database and provides an appropriate Query Interface for it. As illustrated in Figure 4.4, when events are published by an event source, the Eventing Service first transforms them into the internal event format and persists them. Please note, however, that event persistence can be disabled if not desired.

Events are queried using a dedicated Query Interface. Initially, this interface was part of the simple keyword-based Querying Service, which was used to search for services in the first prototype implementation of VRESCo [119]. Since data access is done via the ORM Layer using NHibernate [157], the first version of the event query was built on the Hibernate Query Language (HQL). More precisely, search strings consisting of name-value pairs using simple operators were translated into corresponding HQL queries.

However, this first querying approach had a number of disadvantages, which have encouraged the design and development of the VQL querying framework introduced in Section 3.5. Since VQL represents a generic and type-safe querying approach, it can be used to search for historical events in the same way as users search for services or service metadata.

```
1 // create query object
2 var query = new VQuery(typeof(QoSRevisionEvent));
3 query.Add(Expression.Eq("Revision.Id", 13));
4 query.Add(Expression.Gt("Timestamp", new DateTime(2010, 1, 1)));
5 query.Add(Expression.Eq("Property", Constants.QOS_AVAILABILITY));
6
7 // execute query
8 IVRESCoQuerier querier = VRESCoClientFactory.CreateQuerier("admin", "secret");
9 var results = querier.FindByQuery(query, 100, QueryMode.Exact) as IList<QoSRevisionEvent>;
```

Listing 4.2: Event Query

Listing 4.2 gives an example for a simple event query in VQL. The query in this example searches for all *QoSRevisionEvents* (line 2) concerning service revision 13 (line 3), but it should return only events that have occurred after 1.1.2010 (line 4). Moreover, the query only addresses events regarding availability (line 5). Finally, the query is executed using the *exact* strategy and the top 100 results are returned to the client (lines 8–9).

Since event-based systems often have to deal with vast numbers of events, in some situations using relational databases may not be efficient enough. In such cases, building highly targeted and efficient index structures may be preferred instead. In this regard, we envision to use the Vector space engine described in [149,150] in addition to a traditional relational event database.

Following the Vector space model, documents (events) are represented by n-dimensional vectors where each dimension represents one keyword. The similarity of two vectors then indicates the similarity of the two corresponding documents (events) using these keywords. The advantage of the Vector space model compared to traditional database search is that the search returns a list of fuzzy matches together with a similarity rating. Furthermore, the search queries can be easily executed on multiple distributed vector spaces.

4.7. Event Ranking

In general, event notifications range from critical alerts to minor status updates. The importance and relevance of different events can be estimated by ranking them according to some fitness function. This is of particular interest when dealing with vast numbers of events. The following list describes several ways we have identified for event ranking.

- **Priority-based:** Event priority properties (e.g., 1 to 10 or 'high' to 'low') can be predefined according to the event model, or defined by the event producer when publishing the event. In the latter case, one problem may be that event producers cannot estimate the importance of particular events compared to other events.
- **Hierarchically:** Events are ordered in a tree structure where the root represents the most important event while the leaves are less important.
- **Type-based:** All events are ranked based on their type. That means, each event has a specific type (possibly supporting type inheritance) which is used to define the ranking. However, the importance of some events may not only depend on its type – sometimes the event properties will make the difference.
- **Content-based:** Events can be ranked based on keywords in the notification payload (e.g., if the payload contains the keyword 'exception' it may be more important than events with keyword 'warning' or 'info').
- **Probability-based:** In general, the frequency of different events depends on environmental factors. In this regard, one can assume that frequent events (e.g., *RegistryQueriedEvent*) may be less important than infrequent ones (e.g., *RevisionGetsUnavailableEvent*).
- **Event Patterns:** Finally, some events often occur as part of event patterns (e.g., service proxy is bound to a specific service, followed by service is invoked using this proxy). The ranking mechanism could consider such event patterns.

VRESCo currently supports hierarchical and typed-based ranking through the event type hierarchy, while priority- and content-based ranking can be realized using special event attributes. Probability-based ranking has been integrated by continuously counting the number of occurrences for each event type. This number is then used to calculate the frequency which is attached to the event during publication.

In general, however, it should be noted that event ranking has one inherent problem: While one specific event is critical for one subscriber, it might be only minor for others (e.g., QoS value changes, service revision is deleted, etc.). Yet, introducing event ranking mechanisms provides different ways to express the importance of events. This information can also be useful during event processing. For instance, when huge amounts of events have to be processed at peak loads, critical and important events could be handled first. Event consumers, on the other side, could use different display options for different event priorities (e.g., pop-up windows for critical events versus log entries for minor events).

4.8. Event Correlation

Event-based systems usually deal with vast numbers of events which have to be managed accordingly. Event correlation techniques are used to avoid losing track of all events and their relationship. For instance, Rozsnyai et al. [164] describe the *Event Cloud* that provides different correlation mechanisms. Basically, the idea is to use event properties which have the same value as correlation identifier. For instance, two events (e.g., *ServicePublishedEvent* and *ServiceDeletedEvent*) having the same event attribute *ServiceId* are correlated since they both refer to the same service.

Correlation Set	**Events**	**Identifier**
User Management	Create, update & delete users	UserId
Service Lifecycle	Create, update, delete, bind, invoke & query services	ServiceId
Service Revision Lifecycle	Create, update, delete, bind, invoke, query & tag revisions	RevisionId
QoS	Correlate all QoS measurements of one service revision	RevisionId
Service Category	Correlate all events of services within one service category	CategoryId
Feature	Correlate all events of services that implement one feature	FeatureId

Table 4.5.: VRESCo Event Correlation Sets

In the context of our work, we have identified the correlation sets summarized in Table 4.5, which shows the name of the correlation set, the events which are subsumed and the correlation identifier. The correlation sets mainly cover three different aspects: user management using the *UserId* as correlation identifier, service (and service revision) lifecycle and QoS using *ServiceId* (and *ServiceRevisionId*) and service category information using the *ServiceCategoryId*.

The difference between event correlation sets and event types can be summarized as follows: While event types represent groups of events that occur in the same situations or indicate the same state change (e.g., service is published), event correlation sets group all events that are related due to some event attribute (e.g., service revision *X* is published, deactivated, invoked, or QoS values change, etc.). Therefore, correlation sets enable users to track all important events which are related.

4.9. Event Visibility

In our first prototype, events were visible to all users within the runtime, which can be problematic in business scenarios. For instance, considering our CPO case study, CPO1 might agree that PARTNER1 can see events concerning service management and versioning, but might restrict that events related to binding and invocation are only visible for its own employees.

Mühl et al. [124] discuss security issues in event-based systems by introducing different access control techniques such as access control lists (ACL), capabilities and role-based access control (RBAC) [15]. ACLs represent a simple way to define the permissions of different users (principals) for a specific security object. In contrast, capabilities define the permissions of a specific user for different security objects. The difference is that ACLs are stored for every security object while capabilities are stored for every user. Finally, RBAC extends capabilities by allowing users to have several roles which are abstractions between users and permissions. Users can have one or more roles while permissions are directly granted to the different roles.

In the VRESCo Notification Engine, we have integrated an access control mechanism following RBAC which is similar to the idea of *scopes* [52]. Therefore, users are divided into different user groups. The event visibility can then be defined according to the event visibilities shown in Table 4.6. In our work, event publishers are enabled to define the visibility of their events. While one publisher may not want that other users can see her events (*PUBLISHER*), another may not define any restrictions on events (*ALL*). Furthermore, it is possible to grant only specific users access to events (e.g., *joe*). RBAC is then introduced by either granting access to all users of a specific group which is indicated using a leading colon (e.g., *:admins*), or all users within the same group as the publisher (*GROUP*).

Event Visibility	**Description**
ALL	Events are visible to all users
GROUP	Events are visible to all users within the publisher's group
PUBLISHER	Events are visible to the publisher only
:GroupName	Events are visible to all users within a specific group
Username	Events are visible to a specific user only

Table 4.6.: Event Visibilities

Besides defining event visibilities for different users and groups, more fine-grained access control is provided by allowing users to specify event visibilities for different event types. Clearly, these definitions take the event type hierarchy into consideration: If no event visibility is defined for a specific event type, the engine takes the visibility of the parent type. If there is no visibility for any type the default visibility is chosen (i.e., *ALL* for type *VRESCoEvent*).

The access control mechanism is enforced by the Eventing Service and the Notification Manager (see Figure 4.4). On the one side, the Eventing Service attaches both event visibility and name of the publisher to the event before feeding it into Esper (see Listing C.1 in the Appendix for an example). While the name of the publisher can be directly extracted from the request message of the invoked VRESCo core service (e.g., Publishing Service), the event visibility of the publisher is queried from the Registry Database.

On the other side, when events match subscriptions, the Notification Manager gets name and user group of the subscriber from the Subscription Storage and extracts publisher name and event visibility from the notification payload. Based on this information, the Notification Manager can verify if the current event is visible to the subscriber or not. If the event is visible the subscriber is notified, otherwise the notification is discarded. It should be noted that the VQL querying mechanism follows the same principle: if events returned by a query are not visible to the query requester, they are removed from the result set.

In our approach, publishers are able to specify which subscribers can see which events by using event visibilities. Therefore, event access is mainly controlled by the publishers. Apart from that, however, subscribers are able to specify which event producers they are interested in. This is done by specifying the event attribute *publisher* in the EPL subscription.

4.10. Evaluation

In this section we evaluate the VRESCo event notification support threefold: Firstly, we show the expressiveness of the subscription language by using scenarios from the motivating example in Section 1.1. Secondly, we show concrete code examples how such subscriptions can be requested using the VRESCo Client Library. Finally, we present various performance measurements of the VRESCo Event Notification Engine. Further application scenarios enabled by this work are discussed in Chapter 5.

4.10.1. Subscription Expressiveness

Considering our CPO case study, assume the system administrator of CPO1 wants to get notified as soon as some of their service revisions get deactivated. This can be easily expressed using the following subscription.

```
select * from RevisionDeactivatedEvent
where Service.Owner.Company = 'CPO1'
```

Another example is to notify about new services. Consider that a service consumer wants to get notified if a new revision of service 11 is published. This can be written as

```
select * from RevisionPublishedEvent
where Service.Id = 11
```

The first two examples are intentionally basic. Besides the fact that UDDI does not provide versioning support, these examples could also be implemented using existing registries.

Furthermore, the VRESCo runtime also considers QoS attributes of services which are measured by the QoS Monitor. Although not natively supported by UDDI, this could be implemented by storing QoS attributes in corresponding *tModels* of the UDDI registry as for example illustrated in [212]. To give a concrete example, assume a service consumer wants to get notified, as soon as the response time of service revision 17 is greater than 500 milliseconds, which is expressed by the following subscription.

```
select * from QoSRevisionEvent
where Revision.Id = 17
and Property = 'ResponseTime'
and DoubleValue > 500
```

The notification features of current Web service registry standards mainly provide support for subscribing to static registry data. The VRESCo Event Engine goes one step further and also includes runtime information such as binding and invocation of services. In that way, service providers are enabled to get notified if some service has been invoked. Furthermore, subscribers are interested in events within a given period of time which is supported by the sliding window operator. For instance, getting univariate statistics (e.g., sum, average, variance, etc.) of property *Priority* of the last ten subsequent *ServiceInvokedEvents* concerning service 9 can be expressed as easily as follows.

```
select * from ServiceInvokedEvent(Service.Id=9).win:time(10).stat:uni('Priority')
```

Similar to this, the sliding window can also be defined on the actual time when the events occur. Additionally, the *where* clause can be used to express constraints on the statistical function. For example, the following subscription fires if the average *Availability* of *QoSRevisionEvents* concerning service revision 47 that occurred within the last 24 hours is greater than 95%.

```
select * from QoSRevisionEvent(Revision.Id=47 and Property='Availability')\\
.win:time(24 hours).stat:uni('DoubleValue')
where average > 0.95
```

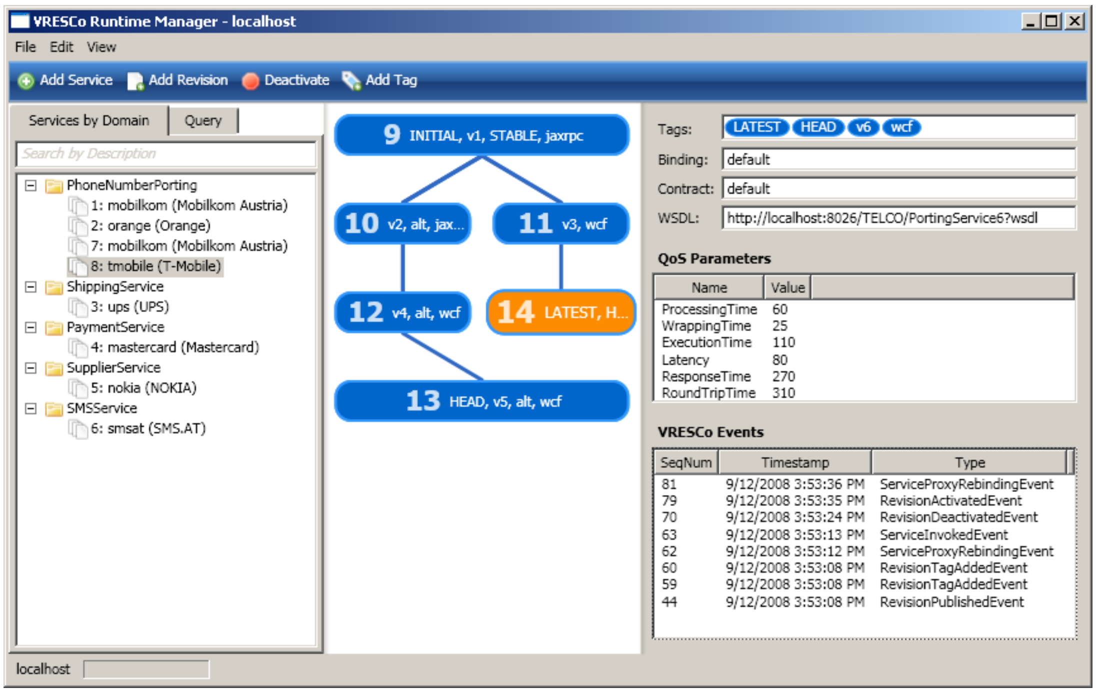
Finally, event patterns may be considered which enables subscribers to define temporal relations between events. For instance, the following subscription fires, if a new service revision is invoked within ten days after its publication. `A -> B` means that event *A* happens before event *B*, the *every* operator defines which events trigger the pattern to be fired.

```
select * from pattern
[every publish=RevisionPublishedEvent -> every invoke=ServiceInvokedEvent\\
     (publish.Revision.Id=invoke.Revision.Id)
where timer:within(10 days)]
```

To summarize, the subscription language of our approach enables to define complex subscriptions using event patterns, sliding window operators and statistical functions on event streams, which cannot be defined using traditional service notification mechanisms.

4.10.2. Software Demonstration

In this section, we give a brief demonstration of the VRESCo Runtime Manager GUI [112, 113] using the CPO case study. Figure 4.5 shows some services of this case study in the Runtime Manager. Service categories and their services are illustrated in the left part of the GUI, which also provides an interface for querying services within the Registry Database. The service revision graph of the selected service is depicted in the middle, showing identifier and tags (e.g., INITIAL, STABLE, etc.) of all service revisions (depicted as blue boxes). The initial revision is always placed on the top of the graph while the edges define the predecessor-successor relationship. The details of the selected service revision (depicted as orange box) are shown in the right part including revision tags, URL of the WSDL document, binding information, current QoS attributes and all events related to this revision (showing sequence number, timestamp and event type). We envision to extend this GUI with additional service metadata information and mechanisms to graphically define service mediation.



Tags: LATEST HEAD v6 wcf
Binding: default
Contract: default
WSDL: http://localhost:8026/TELCO/PortingService6?wsdl

QoS Parameters

Name	Value
ProcessingTime	60
WrappingTime	25
ExecutionTime	110
Latency	80
ResponseTime	270
RoundTripTime	310

VRESCo Events

SeqNum	Timestamp	Type
81	9/12/2008 3:53:36 PM	ServiceProxyRebindingEvent
79	9/12/2008 3:53:35 PM	RevisionActivatedEvent
70	9/12/2008 3:53:24 PM	RevisionDeactivatedEvent
63	9/12/2008 3:53:13 PM	ServiceInvokedEvent
62	9/12/2008 3:53:12 PM	ServiceProxyRebindingEvent
60	9/12/2008 3:53:08 PM	RevisionTagAddedEvent
59	9/12/2008 3:53:08 PM	RevisionTagAddedEvent
44	9/12/2008 3:53:08 PM	RevisionPublishedEvent

Figure 4.5.: VRESCo Runtime Manager

```
1  IVRESCoSubscriber subscriber =
2    VRESCoClientFactory.CreateSubscriber("admin", "secret");
3
4  Identifier sid = subscriber.SubscribePerEmail(
5    "select * from RevisionPublishedEvent where Service.Id = 11",
6    "anton@infosys.tuwien.ac.at", 10*60);
7
8  sid = subscriber.SubscribePerWS(
9    "select * from QoSRevisionEvent"+
10   "where Revision.Id = 17 and Property = 'ResponseTime' and DoubleValue > 500",
11   "net.tcp://localhost:8005/SubscriptionEndTo",
12   "net.tcp://localhost:8006/OnVRESCoEvents", new DateTime(2011,11,11) );
13
14 sid = subscriber.SubscribePerEmail(
15   "select * from QoSRevisionEvent(Revision.Id=47 and Property='Availability')."+
16     "win:time(24 hours).stat:uni('DoubleValue') where average > 0.95",
17   "anton@infosys.tuwien.ac.at", 30*24*60*60);
```

Listing 4.3: Subscription Example Listing

The CPO case study has several scenarios where notifications are useful. Listing 4.3 shows how the examples from the previous section are specified using the Client Library. First of all, a subscriber proxy is generated by the Client Factory which takes the username and password as parameters (lines 1–2). The name of the VRESCo host and the port number of the Subscription Manager Service are read from configuration files.

The first example shown in lines 4–6 represent an email notification. The first parameter defines the subscription in EPL which in this example means that notifications should be sent every time a new revision of service 11 is published (line 5). The second parameter specifies the email address the Notification Manager should use for the notifications, while the third parameter defines the duration of the subscription in seconds (i.e., 10 minutes in this example). The Subscription Manager returns a unique subscription identifier *sid* which can be used to unsubscribe or renew the subscription.

The second example declares interest in *QoSRevisionEvents* where the *ResponseTime* of revision 17 is greater than 500 ms (lines 8–12) since this might violate some SLAs. This time notifications should be sent using Web service notifications following the WS-Eventing specification. The second parameter defines where the *subscriptionEnd* messages should be sent, while the third parameter specifies the destination of the actual notification messages (line 12). The subscription should be valid until 11.11.2011.

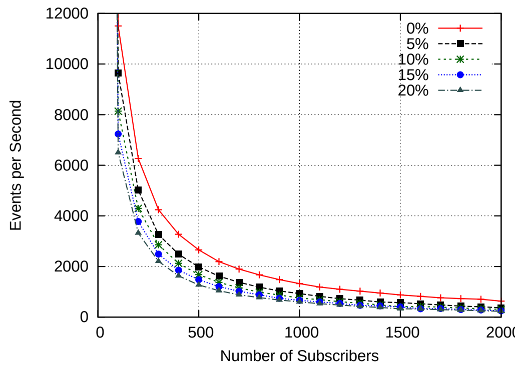
Finally, the third example demonstrates the use of sliding windows and statistical functions on multiple events. More precisely, it defines that notifications should be sent per email if the average *Availability* of *QoSRevisionEvents* concerning service revision 47 that occurred within the last 24 hours is greater than 0.95 (lines 14–17). In addition, the subscription should be valid for 30 days.

4.10.3. Performance Results

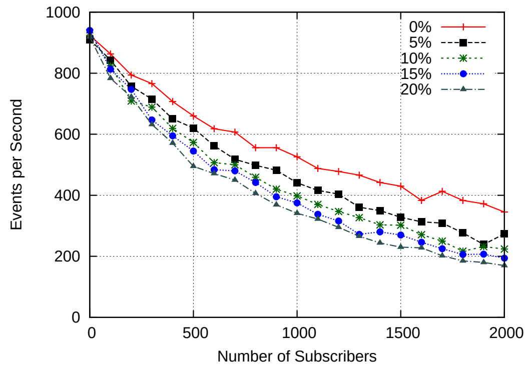
In this section, we give a performance evaluation of the VRESCo Event Notification Engine that includes throughput, processing and overhead. The following experiments have been executed on an Intel Xeon Dual CPU X5450 with 3.0 GHz and 32GB RAM running on Windows Server 2007 SP1 and .NET v3.5, while MySQL v5.1 has been used as database. Furthermore, all results represent the average of 10 repetitive runs.

First, we have evaluated the performance of the Event Notification Engine by using a simulation of QoS events to measure the throughput of the actual matching between events and subscriptions. These events were continuously published, while we increased the number of subscribers and varied the percentage of matching subscriptions (we have chosen values between 0% and 20% since higher values are unusual in typical settings). Finally, we measured how many events can be processed per second. It should be noted that we do not consider the time needed to actually notify external subscribers, since this is done by a dedicated delivery thread pool and varies significantly depending on the notification mechanism (such as email or Web services).

The results of the internal event throughput of our first prototype are shown in Figure 4.6. In this case, the events are raised internally (i.e., by components that are within the VRESCo runtime). Figure 4.6a depicts the throughput of internal events when event persistence is disabled. The graph illustrates that the throughput is high for small numbers of matching subscriptions (e.g., about 115.000 events per second without subscribers which is not shown in the figure) and decreases with the number of matching subscriptions. In this setting, it converges to 250–350 events per second for 2000 subscriptions (depending on the number of matching subscriptions).



(a) Without Persistence



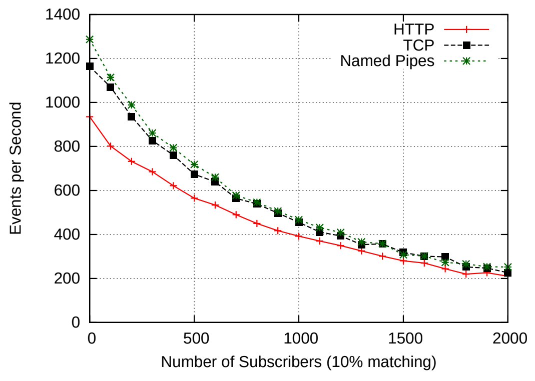
(b) With Persistence

Figure 4.6.: Internal Event Throughput

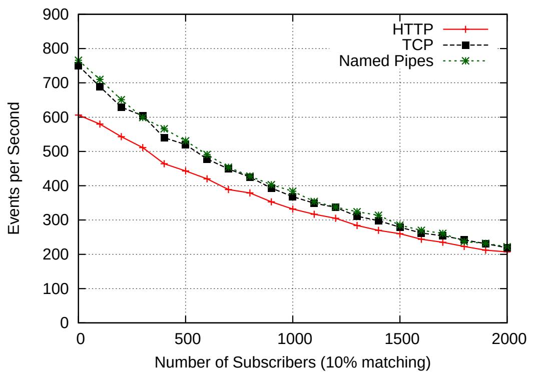
Figure 4.6b shows that providing event persistence significantly reduces the throughput, especially for low numbers of subscribers. This is mainly caused by sequentially persisting events instead of batch processing. However, the performance of the internal events is adequate for our system since the typical setting consists of a small to medium number of services, which minimizes the number of produced events. Furthermore, most events are triggered by invocations of the VRESCo core services, which are provided as Web services. Therefore, the maximum throughput of internal events is usually not reached in practical deployments. Besides integrating batch processing, the performance decrease could be alleviated by refraining from persisting frequent and minor events (e.g., *ServiceFoundEvent*, etc.), or by removing event persistence as integral part of the engine and using external components instead.

Figure 4.7a depicts the throughput of external events (e.g., fired by the QoS Monitor) that are published using the Management Service. Besides HTTP (red line), we have used additional bindings such as TCP (black line) and named pipes (green line) for the external event interface. The results are different to [114], since the VRESCo runtime has evolved in the meanwhile and we have used a more powerful machine for this evaluation. Figure 4.7b shows the throughput decrease when events are persisted (we have used 10% matching subscriptions in this example). Both figures indicate that the external event throughput using HTTP is only slightly lower than TCP and named pipes. Therefore, we currently use HTTP for this interface since integration with external components is most comfortable when providing Web service invocations over HTTP. However, JMS [177] or message queuing (MSMQ) could also be integrated.

In general, the throughput performance is satisfying but there is still room for improvement. Therefore, we have integrated batch processing and notification thread pools in a second step [118]. We have again evaluated the internal throughput of the Event Notification Engine. The experiment setting is the same as used in Figure 4.6 (with event persistence).



(a) Without Persistence



(b) With Persistence

Figure 4.7.: External Event Throughput

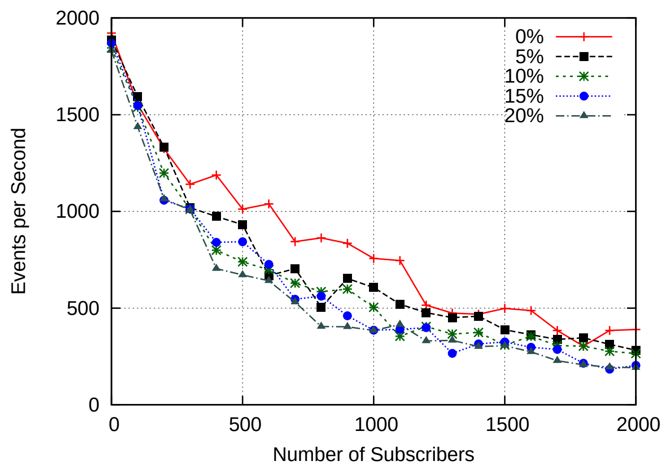


Figure 4.8.: Internal Eventing Throughput with Persistence and Batch Processing

The results are depicted in Figure 4.8. Obviously, the throughput decreases with the number of subscribers and the percentage of matching subscriptions. However, this time it starts about 2000 events per second without subscriptions (compared to about 900 events without batch processing) and again converges to 250–350 events per second for 2000 subscriptions. As stated above, the measured throughput is still higher than the expected number of events in typical VRESCo settings.

Another interesting point is the effect of the different processing steps that are involved in event processing. For this experiment, we have simulated 1000 event publications using different event types. Then, we have measured the performance overhead of each processing step. To keep it simple, we have used only one subscription that matches all events. To eliminate outliers, the results represent the median of the measured values.

Table 4.7 summarizes the results which are divided into four processing steps (please note that the values represent microseconds). The column *Generation* specifies how long it takes to generate the event and append the necessary information. This includes name of the publisher, event visibility, timestamp and sequence number. *Ranking* shows the overhead of probability-based ranking as introduced in Section 4.7. *Persistence* defines the time needed to store events in the Event Database, which is done prior to the actual publication (since we want to measure the values for single events, we have disabled batch processing for this experiment). Finally, *Processing* specifies how long it takes to process events. *Total* represents the overall time needed to publish the event into Esper and process the listeners. This includes to check the event visibility (*EV*) and convert Esper events back into the VRESCo core model (*Conversion*). The time needed for notifying external subscribers is ignored here, since it heavily depends on the used notification mechanism.

Event Type	Generation	Ranking	Persistence	Processing		
				EV	Conversion	Total
QoSEvent	1550,1	0,9	1143,2	1,2	9,0	738,7
ServiceManagementEvent	1443,8	0,9	1107,8	1,2	7,9	737,2
VersioningEvent	1524,4	0,9	1122,4	1,3	9,2	734,6
MetadataEvent	1469,5	0,9	1105,9	1,2	8,7	735,5
BindingInvocationEvent	1512,7	0,9	1133,9	1,3	8,6	737,0
ProcessEvent	1464,5	0,9	1094,6	1,2	8,7	729,3
UserManagementEvent	2311,2	0,9	1053,7	1,2	8,5	713,0

Table 4.7.: Event Processing Performance (in μs)

The results indicate that the measured values are almost identical for all event types, except for *UserManagementEvents* which are slightly more expensive to generate. The time needed for *Generation* could be reduced by caching visibility of frequent events, since this is currently read from the database. The overhead of *Ranking* is below 1 microsecond, however, it can still be disabled if not desired. *Persistence* is an expensive operation that can either be reduced when using batch processing (as shown in the throughput evaluation above) or completely disabled in the configuration. Finally, *Conversion* slightly differs for the different event types, while checking event visibility has almost constant overhead here, since we have used the same event visibility for all events.

For the sake of completeness, we did a re-run of the last experiment using *QoSEvents* with different event visibilities to measure the performance overhead. Table 4.8 shows how long it takes to verify the event visibility depending on the visibility type. It can be seen that verifying specific users or specific user groups is twice expensive than using the pre-defined visibilities *ALL*, *GROUP* and *PUBLISHER*. However, the evaluation has shown that event visibility checking is very efficient for all visibility types.

Type	Check EV
ALL	1,1
GROUP	1,3
PUBLISHER	1,3
:groupname	2,5
username	2,3

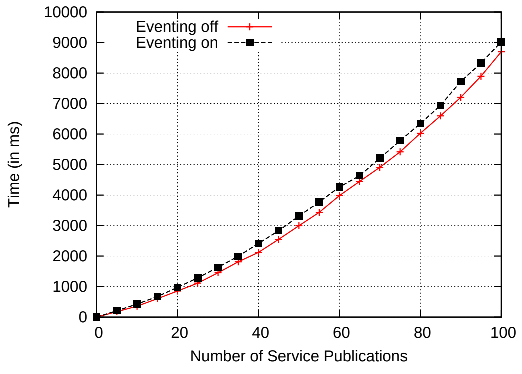
Table 4.8.: Check Event Visibility (in μs)

Furthermore, we have also measured the time needed for the different notification mechanisms provided by VRESCo, since this was not part of the results shown in Table 4.7. We have used 10 event publications for each event type. However, since there was no significant performance deviation between the different event types, we did not distinguish between them any further.

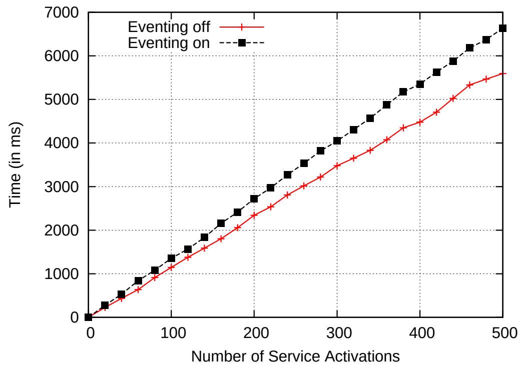
	Console	**WS-Eventing**		**Email**	
		Successful	**Failure**	**Async. Send**	**ACK**
First	1,0	245,3	1118,1	43,8	181,1
Average	0,1	37,6	1008,2	0,7	7,7

Table 4.9.: Notification Duration (in ms)

Table 4.9 shows the three main notification mechanisms *Console*, *WS-Eventing* and *email*. Furthermore, we depict both average results and the results of the first run, which are significantly higher due to initialization (e.g., proxy generation for listener Web service or connection setup to the mail server). It can be seen that console listeners are clearly fastest. For WS-Eventing, the first notification needs about 240 ms while the following notifications are much faster. In contrast, if the listener Web service is not available the notification process takes about one second (i.e., socket timeout plus time for persisting the pending notification). Finally, emails are sent asynchronously which needs about 40 ms for the first message and goes down to about 1 ms in average, while the acknowledgement from the mail server needs about 8 ms in average.



(a) Service Publication



(b) Service Activation

Figure 4.9.: Eventing Performance

Finally, the overhead of the eventing support is also considered. Therefore, we have measured the overhead for two invocations of the Publishing Service, which is shown in Figure 4.9. More precisely, we have simulated a certain number of sequential Web service publications (i.e., one service with one initial service revision) both with and without eventing support, and measured how long both operations take. The results in Figure 4.9a show that the average overhead is less than 10% in this setting. In Figure 4.9b we have repeated the experiment of deactivating/activating service revisions (compare Figure 3.16). The results indicate that the average overhead increases to 18% in this setting. The reason is that activation/deactivation of revisions represents a more basic operation than service publication. Therefore, the almost constant overhead of eventing has more impact on the overall time in this experiment. In general, we argue that the eventing overhead is acceptable, especially when considering the possibilities opened up by the VRESCo eventing mechanism. However, it should be noted that eventing support can be completely disabled in the configuration if not desired.

4.11. Related Work

Event-based systems [124] and the Publish/Subscribe paradigm [47] have been researched in the past years, which led to event-based architecture definition languages (e.g., Rapide [103]) and QoS-aware event-dissemination middleware [104]. Moreover, data and event stream processing has also been addressed in different prototypes (e.g., STREAM [11], Esper [45], etc.).

Approaches to integrate Publish/Subscribe and the SOA model led to WS-Notification [138] and WS-Eventing [199]. Furthermore, combining event-driven architectures and SOA is also addressed by Enterprise Service Bus implementations (e.g., Servicemix [9], Mule [125], etc.). In contrast to our work, ESBs mainly focus on connecting various legacy applications by using a common bus that performs message routing, transformation and correlation.

Ostrowski and Birman [142] have studied the combination of Web service notifications and IP multicast, in order to provide a scalable and reliable Publish/Subscribe platform. Cugola and Di Nitto [34] give a detailed overview of other research approaches combining SOA and Publish/Subscribe. Furthermore, they introduce a system that aims at adopting content-based routing (CBR) in SOA by extending the work presented in [14]. Their approach is built on the CBR middleware REDS [37] and provides notifications following WS-Notification. Service discovery is implemented according to the query-advertise style using UDDI inquiry messages. The aim of this work is to use CBR to perform service discovery, while we focus on event processing and notifications in service runtime environments. Besides service discovery, we also provide support for dynamic binding and QoS attributes.

Medjahed [106] presents several patterns for event-driven SOA by categorizing using the two taxonomies *interaction* and *filtering*. According to the author, interaction has three dimensions: *mode* (push vs. pull), *cardinality* (1:1 vs. 1:N) and *strategy* (implicit vs. explicit dissemination). In contrast, filtering has the two dimensions *event* (topic-based, content-based, type-based, semantic and generic filtering) and *service* (specific, category-based, policy-based and semantic filtering). The remainder of this work focuses on various protocols for the peer-to-peer Push 1:N-Implicit pattern using topic-category-based filtering, which the author refers to as *implicit notifications*. According to this categorization, the VRESCo Event Engine follows the centralized 1:N Explicit pattern, and provides support for both push- and pull-style. Filtering is currently done content- and type-based on events, while the service dimension is supported both specific and category-based. Our access control mechanism using event visibility can be seen as a special form of policy-based filtering.

Service registries (e.g., UDDI [136], ebXML [134]) represent one part of the SOA triangle that is responsible for maintaining a service repository including publishing and querying functionality. Both UDDI and ebXML provide subscription mechanisms to get notified if certain events occur within the service registry. However, these notifications are limited to the service data stored in the registry and do not include service runtime information. Notifications are sent per email or by invoking listener Web services. Other registry approaches such as AWSR [184] and XMethods [207] use news feeds (e.g., RSS [165] and Atom [166]) for dissemination of changing service repository content. News feeds enable to seamlessly federate multiple registries, yet, in contrast to our approach, do not provide fine-grained control on the received notifications since they follow the topic-based subscription style. Furthermore, similar to UDDI and ebXML, these approaches do not include service runtime information.

There are several approaches that address search in historical events [79, 95, 164]. Rozsnyai et al. [164] introduce the *Event Cloud* system, which aims at searching for business events. Their approach uses indexing and correlation of events by using different ranking algorithms, and is based on the open source text search engine Apache Lucene [8]. In contrast to our approach, the focus of this work is on building an efficient index for searching in vast numbers of events whereas subscribing to events and getting notified about their occurrence is not addressed.

Li et al. [95] present a data access method which is integrated into the distributed content-based Publish/Subscribe system PADRES. The system enables to subscribe to events published in both the future and the past. In contrast to our work, the focus is on building a large-scale distributed Publish/Subscribe system that provides routing of subscriptions and queries.

Jobst and Preissler [79] present an approach for business process management and business activity monitoring using event processing. The authors distinguish between SOA events regarding violation of QoS parameters and service lifecycle, as well as business/process events building upon BPEL. These events are fired by `receive` and `invoke` activities within a BPEL process. In contrast to our work, the focus is on search and visualization of business events whereas subscribing to events is not addressed. Furthermore, the different SOA events and how they are handled is not described in detail.

Finally, there is some work on access control of event-based systems [15, 53, 206]. Belokosztolszki et al. [15] discuss general requirements regarding access control in Publish/Subscribe systems. Furthermore, the authors integrate role-based access control into the Hermes middleware [147]. Conceptually, this work is similar to ours. Furthermore, they also address how trust between brokers can be achieved in P2P Publish/Subscribe networks. Fiege et al. [52, 53] introduce the notion of scopes in event-based systems, which was adopted in our work. In their approach, scopes are used to group components in a hierarchical manner. Events between two components are sent only if they are visible to each other (i.e., share a common superscope). In contrast to that, in our work visibility is defined by the event publisher. Therefore, the event type hierarchy and the user groups are taken into consideration.

4.12. Conclusion

Since services change regularly, service consumers want to get notified about such changes. Event notifications have been addressed in service registry standards (e.g., UDDI and ebXML) and other service repository approaches as introduced in Section 2.4. However, most approaches do not support complex event processing and consider only changes in registry data, which does not include runtime information concerning binding and invocation of services.

In this chapter, we have presented an approach for event notifications in service runtime environments that is capable of including such runtime information together with QoS attributes. Additionally, complex event processing is provided by temporal relations between events using the sliding window operator, event patterns and statistical functions on events. Furthermore, we have presented how historical events can be accessed and how event visibility provides fine-grained access control on events. Finally, we have shown how our approach has been integrated into the VRESCo runtime environment.

The evaluation has been done threefold: Firstly, we have shown the expressiveness of the subscription language using some usage examples. Secondly, we have demonstrated how concrete subscriptions are implemented in VRESCo. Thirdly, the performance results have indicated that the event notification support can deal with the expected number of events and subscribers. Additionally, the overhead of the Eventing Engine and the performance improvements with respect to the first prototype implementation have been investigated. Finally, we have mentioned related work in this area.

Chapter 5.

Service Notification Applications

After describing the VRESCo runtime environment and its event notification support in detail, this chapter presents three application scenarios that have been implemented based on this work. Firstly, Section 5.1 describes how events can help in combining the advantages of the different rebinding strategies provided by VRESCo. Secondly, the events stored in the Event Database can be used to maintain the provenance of services, which is presented in Section 5.2. Thirdly, Section 5.3 shows how events are leveraged to combine client- and server-side QoS monitoring and SLA violation detection. Furthermore, Section 5.4 and 5.5 briefly sketch other use cases that are enabled by VRESCo eventing, but have been left for future work. Finally, Section 5.6 concludes this chapter.

Contents

5.1. Notification-based Rebinding

The first use case for the VRESCo eventing support represents one of the core challenges that have driven the development of the VRESCo runtime environment, namely dynamic binding and invocation of services. In contrast to many existing SOC solutions, VRESCo enables clients to dynamically bind and invoke different services from a pool of service candidates, even if the technical service interfaces are different. In this regard, the VRESCo mediation mechanism is responsible for applying the necessary mapping functions to hide this interface mismatch.

5.1.1. Rebinding Strategies Revisited

In Section 3.6.2, we have introduced different rebinding strategies. Basically, these strategies are proactive in the sense that service requesters have to define when the rebinding should be done (e.g., *OnInvocation* and *OnDemand*), how often it should be done (e.g., *Periodic*), or if the service proxies should not rebind at all (e.g., *Fixed*).

In the evaluation in Section 3.9, we have discussed that it always depends on the specific situation which rebinding strategy to use, since all strategies have their strengths and weaknesses. For instance, the *OnInvocation* strategy obviously always invokes the best available service but introduces constant overhead for every service invocation. Therefore, this is not a good choice if the services do not change often. In contrast, the *Periodic* strategy does not influence the actual service invocation since the service proxy considers its binding periodically in the background. However, it is not guaranteed if the proxy always invokes the best service. Clearly, decreasing the rebinding interval can mitigate this problem, but this finally leads to processing overhead for both clients and the VRESCo runtime (which has to be queried when considering the binding). As a result, when choosing a rebinding strategy one usually has to make a tradeoff between invocation/rebinding overhead and optimal service binding.

5.1.2. OnEvent Rebinding Strategy

The VRESCo eventing support bridges the gap between the different rebinding strategies by providing manners to combine their advantages in a reactive way. Basically, the idea is that the rebinding of service proxies is done when critical events occur (e.g., the service response time goes beyond a given threshold, the service is deactivated, etc.). Therefore, clients use subscriptions to specify in which situations the current binding should be verified, instead of using periodic rebinding intervals or rebinding on every service invocation. The actual rebinding is finally triggered by event notifications that match these subscriptions.

Listing 5.1 shows how service proxies use the *OnEvent* strategy. As before, a VQL query is used to define the desired service revision (lines 2–8). We have used the same query as shown in Listing 3.1, except that it uses *Relaxed* instead of *Priority* querying. The actual subscription is

```
1  // define query
2  var query = new VQuery(typeof(ServiceRevision));
3  query.Add(Expression.Eq("IsActive", true));
4  query.Add(Expression.Eq("Service.Category.Features.Name", "NotifyCustomer"));
5  query.Match(Expression.Eq("Service.Owner.Company", "CompanyX"));
6  query.Match(
7      Expression.Eq("QoS.Property.Name", "ResponseTime") &
8      Expression.Lt("QoS.DoubleValue", 1000.0));
9
10 // define subscription
11 string epl = "select * from RevisionDeactivatedEvent";
12 string notifyTo = "net.tcp://vresco.vitalab.tuwien.ac.at:33333/OnVRESCoEvents";
13 DateTime expires = new DateTime(2011, 11, 11);
14
15 // create proxy using onEvent strategy
16 var querier = VRESCoClientFactory.CreateQuerier("username", "password");
17 DaiosProxy proxy = querier.CreateRebindingMappingProxy(query, QueryMode.Relaxed, 10,
18   new OnEventRebindingStrategy(epl, notifyTo, expires));
```

Listing 5.1: OnEvent Proxy Generation

then defined in lines 11–13 using EPL. In this example, the rebinding should be verified every time a service revision is deactivated (which is signaled by `RevisionDeactivatedEvents`). Furthermore, *notifyTo* and *expires* define the notification endpoint and how long the subscription is valid. This subscription information, together with the VQL query, is finally used to generate the service proxy in lines 17–18. The actual service invocation is the same as shown in Listing 3.2, and has therefore been omitted for brevity.

Listing 5.2 depicts the corresponding event handler method `Notify` of the `IEventNotification` interface. Service proxies listen for WS-Eventing notifications defined by the *notifyTo* endpoint and invoke this method when notifications arrive. Besides writing tracing information for debugging, the rebinding of the service proxy is finally triggered in line 10.

```
1  class MyServiceProxy : IEventNotification {
2    // ...
3    public void Notify(VRESCoEvent[] newEvents, VRESCoEvent[] oldEvents, string subscriptionId)
4    {
5      Trace.WriteLine("I got the following rebinding events for subscription: " + subscriptionId);
6      for (int i = 0; i < newEvents.Length; i++ )
7        Trace.WriteLine(newEvents[i].ToString());
8
9      // force proxy to rebind
10     proxy.ForceRebinding();
11     Trace.WriteLine("Rebinding done...");
12   }
13 }
```

Listing 5.2: OnEvent Handler

Finally, Listing 5.3 shows a code snippet of the configuration file, which is needed to host the

```
<services>
  <service name="MyServiceProxy">
    <endpoint
      address="net.tcp://vresco.vitalab.tuwien.ac.at:33333/OnVRESCoEvents"
      binding="customBinding"
      bindingConfiguration="binding1"
      contract="RKiss.WSEventing.IEventNotification"/>
  </service>
</services>
```

Listing 5.3: OnEvent Handler Configuration

WS-Eventing endpoint of the event handler. It defines the contract (i.e., interface) of the handler (`IEventNotification`), the name of the service (`MyServiceProxy`), and the endpoint address (`net.tcp://vresco.vitalab.tuwien.ac.at:33333/OnVRESCoEvents`) plus binding information (TCP). Currently, these settings have to be manually configured by the client. However, we envision to automate this step so that the WS-Eventing endpoints are hosted autonomously.

It should be noted, that in this basic example the rebinding is done every time service revisions are deactivated (since the subscription declares interest in all `RevisionDeactivatedEvents`). During rebinding, the service proxy first queries the VRESCo runtime using the query specified above, and then binds to the best result. However, it is also possible to attach the new service revision (i.e., the revision that should be bound) to the notification, so that the service proxy can extract this information from the notification payload and directly bind to it without querying the VRESCo runtime again.

5.1.3. Evaluation

For evaluating the *OnEvent* strategy, we have re-run the evaluation shown in Figure 3.14. Briefly summarized, we have used the Web service testbed Genesis [80] to simulate 10 services that implement the same feature, but exhibit different response times (using a Gaussian distribution and by increasing the variance). Finally, we have implemented one client for each rebinding strategy and measured the average response time when invoking the service.

The figure shows that *OnEvent* (blue line) removes the constant overhead of 300–400ms which is introduced by *OnInvocation* (green line). The reason for this is that rebinding is done using background threads and does not influence the service invocation time. The same behavior can be seen for the *Periodic* strategy (black line). However, in contrast to periodic rebinding *OnEvent* invokes the best service since the binding is always up-to-date (due to the fact that rebinding is triggered by events). Furthermore, it usually leads to less overhead on the client since the rebinding is not checked periodically, but only when certain events of interest occur. However, this mainly depends on the concrete subscription and rebinding interval.

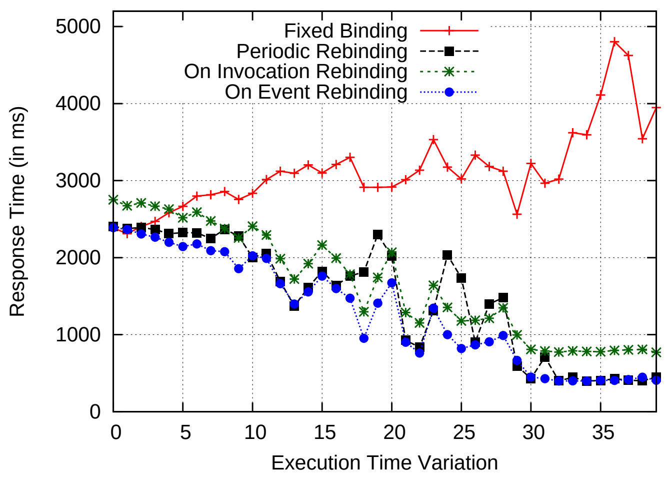


Figure 5.1.: Rebinding Strategy Performance (with *OnEvent* strategy)

One obvious drawback of the *OnEvent* strategy is that it requires the VRESCo eventing support, which is optional and can be turned off if not desired. Furthermore, clients must be able to host WS-Eventing endpoints and must allow the VRESCo Notification Manager to invoke these endpoints. This may lead to problems, for instance if firewalls drop network connections to non-standard ports. This is the main reason why we currently let clients manually define which ports to use for notifications. Finally, if subscriptions are not chosen properly (e.g., rebind on every event) it may seriously degrade the overall system performance, which also happens if clients choose very short intervals for the *Periodic* strategy (e.g., rebind every second).

5.1.4. Conclusion

This section has introduced the *OnEvent* strategy as first use case which shows the usefulness of the Event Notification Engine. The aim of this strategy is to combine the strengths and reduce the weaknesses of the rebinding strategies initially introduced in VRESCo. Thereby, event notifications are used to trigger the rebinding of service proxies. More precisely, subscriptions are used to define when rebinding should be done automatically. For instance, clients can now define that rebinding is triggered when service revisions are deactivated, or when certain QoS attributes change (or any other situation which is recorded by events).

We have described how clients can use this strategy using illustrative code snippets from the CPO case study. Furthermore, the evaluation was done using a Web service testbed. The results have shown that the *OnEvent* strategy indeed binds to the best service without introducing any invocation overhead (as done by the *OnInvocation* strategy).

5.2. Service Provenance

Another interesting use case for VRESCo eventing goes beyond just notifying subscribers if certain events of interest occur. Since all events are also persisted in the Event Database, this can be used as source for historical event information. This information, together with service metadata stored in the Registry Database, represents the provenance of services. In the following section, we present a novel service provenance approach [116, 118] that was implemented within the scope of this thesis.

5.2.1. Introduction

The term *provenance* is commonly used to describe the origin and well-documented history of some object and is used in various areas such as fine arts, archeology or wines. Provenance information is often used to prove the authenticity and estimate the value of objects. For instance, the price of wine depends on origin, vintage and how the wine was stored. In information systems, the notion of provenance was adopted to refer to the origin and history of electronic data [121], which has been applied in different research areas such as e-Science [172].

Provenance is an important issue that enables (especially in service-oriented systems) assertions on who did what in applications or business processes (possibly including human interaction). Based on the availability of event data, provenance information can be gathered and used to proof compliance with certain regulations (e.g., laws, standardized processes, etc.).

SOA and Web services represent well-known paradigms for developing flexible and cross-organizational enterprise applications. Data provenance in such applications and the provenance of business processes as realized in Business Activity Monitoring are important issues that have been addressed by several research projects [38, 154, 185]. These approaches mainly focus on the provenance of data which is produced, transformed or routed through an SOA system. In contrast to that, service provenance also plays a central role, for instance during service selection. If there are multiple alternative services available, service consumers may be interested in the history of candidates. This includes creation date, ownership or modifications, and QoS information such as failure rate or response time. Additionally, service providers are also interested in service provenance (e.g., to identify services that do not perform well).

In this section, we introduce a service provenance approach that has been integrated into the VRESCo runtime environment. In most current approaches, provenance information is captured at runtime and usually managed in a dedicated provenance store. In our approach, we have enhanced the existing event processing mechanism in order to capture and maintain provenance information. Events are thereby published and correlated when certain situations occur (e.g., new service is created, service revision is added, QoS changes, service operation is invoked, etc.).

5.2.2. Motivation

Existing work on provenance mainly focuses on data provenance (i.e., how and when data was created, transformed or accessed in business processes or scientific workflows). This is important, for instance, to validate the results of scientific simulation runs. In contrast to that, however, we aim at addressing service provenance, which is the origin and well-documented history of services. Although conceptually similar at first glance, these two paradigms differ since our work introduces a different view on provenance in service-oriented systems.

The CPO case study from Section 1.1 highlights the motivation of our work. As stated above, service selection represents an illustrative application for service provenance. If there are multiple alternative services, service consumers might want to take the origin and history of the alternatives into consideration. For instance, one service consumer may not trust specific service providers (due to bad experience in the past). Other service consumers may pay special attention to QoS values of alternative services. If one service has performed well over the last months, it might be preferred over recently published services without documented QoS history. Once selected, the services may change which also includes their behavior regarding QoS. In such cases, service consumers may want to automatically rebind to alternative services, which can be triggered based on existing provenance information. Besides service selection, another motivation for service provenance derives from the fact that service consumers and service providers continuously query and monitor current provenance information (e.g., changing QoS attributes, new service revisions, etc.). For instance, service providers can verify if their services perform as expected and take corrective actions otherwise.

Current service registry standards, such as UDDI [136] and ebXML [134] provide only limited support for service provenance. In UDDI, the *businessEntity* construct can be used to store information about the owner of a service, but this construct is fixed and there is no further support for more complex structures regarding the history of a service. In ebXML, every *RegistryObject* can be associated to persons or organizations that have either submitted this information or are responsible for it. In addition, ebXML provides full versioning of registry information. Therefore, the provenance model of ebXML is clearly advanced compared to UDDI, but there is still no support to further collect and process service provenance information.

5.2.3. Provenance Approach

The following section describes how the service provenance approach is realized in VRESCo. This includes how to collect, retrieve and visualize provenance information. Furthermore, we also aim at addressing security issues that typically occur in provenance systems. Finally, our approach should be integrated into an existing service runtime environment instead of introducing a dedicated and stand-alone provenance system.

Collecting Service Provenance Information

In VRESCo, service provenance information is collected at runtime. This consists of various aspects, such as basic service information, service metadata and service runtime events. While the former two are mostly published by service providers, events are raised automatically by the runtime. These aspects are discussed in more detail next.

- **Basic Service Information:** The first part of service provenance information is represented by what we call basic service information, which is kept in the Registry Database. This consists of required information to invoke services (e.g., service endpoint, binding, WSDL document, etc.). Furthermore, every service can be associated with service owner information (e.g., name, address, etc.). As described in Section 3.4 services can have multiple service revisions that are represented in service revision graphs. According to this, service versioning information and revision tags (i.e., every revision can be tagged by the service provider) are also part of provenance information.
- **Service Metadata:** Besides basic service information, another important source for provenance information is represented by service metadata as described in Section 3.2. Briefly summarized, service metadata in VRESCo is used to describe the functionality and semantics of services that cannot be seen in the WSDL descriptions. To accomplish this, we have defined a mapping between our service model and service metadata model, which is shown in Figure 3.3.
- **Service Runtime Events:** The third and most important source of provenance information is provided by the VRESCo Event Engine, which has been introduced in Chapter 4. Basically, the idea is to publish events when certain situations occur, such as new services being published or existing services being modified. Subscribers are then enabled to receive notifications using different mechanisms (e.g., email, WS-Eventing, etc.). In addition, all events are persisted in the Event Database, and can later be retrieved. This opens new possibilities for our provenance approach, such as provenance subscriptions and provenance queries which are described below.

Security Considerations

Before going into the details of our provenance approach, we want to briefly mention security issues, which are often neglected in provenance approaches. Provenance data represents sensitive information that should not be publicly available. Moreover, fine-grained access control policies can be used to gain access to this information only for specific users.

The VRESCo Access Control Layer introduced in Section 3.8 provides the required mechanisms regarding authentication and authorization. Firstly, all clients need to be authenticated when interacting with the VRESCo core services. Therefore, only authenticated users can invoke the core services (e.g., to enter provenance information into the system). Secondly, the claim-based access control mechanism provides authorization features in order to define which users are allowed to create, read, update or delete which resources. Finally, event visibility (as introduced in Section 4.9) can be used to define which users can access which events (either through notifications or by using queries in historical event data). Since the service provenance approach builds on these events, event visibility is of particular importance.

Provenance Queries

Once provenance information is collected at runtime, the next issue is how to access and query this information. This ranges from simple queries like "Who has created service *X*?" to more complex queries like "What is the average response time of service *X*?" or "How often has service *X* been invoked in the last 24 hours?".

```
1  IVRESCoQuerier querier = VRESCoClientFactory.CreateQuerier("username", "password");
2
3  // build provenance query regarding QoS
4  var query1 = new VQuery(typeof(QoSRevisionEvent));
5  query1.Add(Expression.Eq("Revision.Id", 815));
6  query1.Add(Expression.Eq("Property", Constants.QOS_RESPONSE_TIME));
7  query1.Add(Expression.Gt("DoubleValue", 500));
8
9  // build provenance query regarding invocations
10 var query2 = new VQuery(typeof(ServiceInvokedEvent));
11 query2.Add(Expression.Eq("Revision.Id", 4711));
12 query2.Add(Expression.Eq("Publisher", "telco1"));
13 query2.Add(Expression.Gt("Timestamp", new DateTime(2010, 1, 1)));
14 query2.Add(Expression.Lt("Timestamp", new DateTime(2010, 1, 31)));
15
16 // execute provenance queries
17 var results1 = querier.FindByQuery(query1, QueryMode.Exact) as IList<QoSRevisionEvent>;
18 var results2 = querier.FindByQuery(query2, QueryMode.Exact) as IList<ServiceInvokedEvent>;
```

Listing 5.4: Provenance Queries

Listing 5.4 gives two examples for provenance queries. Initially, the *querier* (i.e., the proxy to the Query Engine) is created using the Client Library that takes username and password as input (line 1). The certificates are attached by the Client Library transparently. The first query (lines 4–7) returns all measuring points (*QoSRevisionEvents*) where the response time of service revision 815 was greater than 500 milliseconds. The second query (lines 10–14) returns all service invocations (*ServiceInvokedEvents*) of service revision 4711 from user *telco*1 that happened between 1.1.2010 and 31.1.2010. After the queries are built, they are executed using the *querier* in lines 17–18. The Query Engine returns all events considering their visibility. For instance, if the event visibility of *ServiceInvokedEvents* is set to user *sue*, the query will return no results for user *joe*. Internally, the Query Engine first builds the result set of the query, then iterates through the results to check the visibility and finally returns only the visible events.

Provenance Subscriptions

Besides using queries on the historic provenance information stored in the runtime, the Event Notification Engine enables users to subscribe to certain events of interest. Subscriptions for events or event patterns are specified in the Esper Event Processing Language (EPL) [45], which has been introduced in Section 2.2. If such events or event patterns occur, notifications are sent to the interested subscribers using email or WS-Eventing notifications. This mechanism can now be leveraged to receive notifications if events of interest occur that refer to provenance information.

```
1 IVRESCoSubscriber subscriber = VRESCoClientFactory.CreateSubscriber("username", "password");
2
3 // subscribe per email
4 int sid = subscriber.SubscribePerEmail(
5   "select * from QoSRevisionEvent where " +
6   "Revision.Id = 815 and " +
7   "Property = 'ResponseTime' and DoubleValue > 500",
8   "joe@foo.bar",
9   new DateTime(2010, 1, 1));
```

Listing 5.5: Provenance Subscription

Listing 5.5 gives an example subscription. It should be noted that this subscription is semantically equal to the first query shown in Listing 5.4. The actual subscription is defined in EPL in lines 5–7. If the response time of revision 815 is greater than 500 milliseconds, a notification email should be sent to the given email address (line 8). The date in line 9 specifies that the subscription is valid until 1.1.2010. Furthermore, the identifier *sid* returned in line 4 can be used to cancel or renew the subscription.

Provenance Graphs

Besides querying provenance information, another useful feature is to illustrate this information using provenance graphs. The aim of these graphs is to give an overview of relevant provenance information, such as service versioning information, service ownership and service history regarding binding and invocation, as well as QoS attributes. The input of provenance graphs can either be services/revisions or provenance queries. In the first case, the graph is built with all provenance information that is available for the requested service or revision. In the second case, the result of a provenance query (which is a list of events as shown in Listing 5.4) is displayed in a graph. This is done using pre-defined templates that control the graph generation. These templates are based on the event type returned by the provenance query (i.e., only the relevant parts of the provenance graph are shown). Currently, we provide such templates for QoS events and service invocation events.

Due to the vast amount of information stored in the runtime, the provenance graphs tend to get overloaded quickly. Therefore, the information inherent to the events is divided into several groups such as core service details, versioning graph, invocations, QoS attributes, revision tags and operations. Each group summarizes the information of the corresponding events.

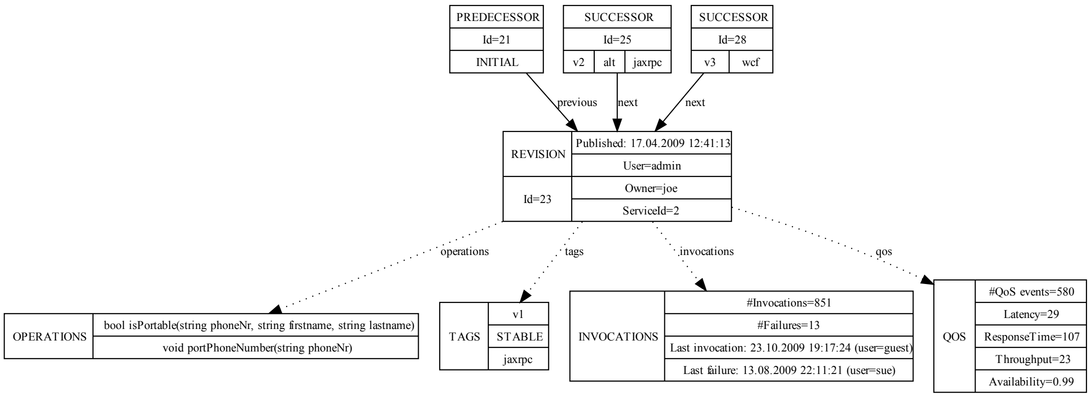


Figure 5.2.: Provenance Graph

Figure 5.2 depicts a provenance graph example which was generated by our approach. This graph shows provenance information of a specific service revision. First of all, parts of the versioning graph are shown on the top of the graph. This includes both predecessors (edge *previous*) and successors (edge *next*) of the current revision. In this case, the revision has one predecessor and two successors which represent the beginning of a branch. The revision itself is positioned in the center and gives information about the corresponding service, owner, creation date and the user that created this revision. While the first two elements are read from the service metadata, the last two elements are stored in the `RevisionPublishedEvent`. The bottom part illustrates the groups *Invocations* (i.e., number of successful and failed service invocations, last successful and failed service invocation, etc.), *QoS* (i.e., number of QoS events and aggregated QoS information), *Tags* (i.e., all revision tags such as “v1”) and *Operations* (i.e., all operations of this revision including their input and output).

The service provenance graphs in VRESCo are built using the open source graph drawing libraries QuickGraph [32] and GraphViz [58]. Before such a graph is built, all relevant provenance information (i.e., events and service metadata) is retrieved using the Query Engine. The corresponding graph is then generated using this information while the graph libraries are used to render the resulting graph according to the user’s preferences (e.g., PDF, PNG, etc.). The graph image (or a graph representation as GraphViz’ DOT file [58]) is finally returned to the user. The overall approach of building provenance graphs is implemented and provided as part of the VRESCo Query Service.

Selective Service Provenance

As described above, VRESCo provides fine-grained access control for service provenance information stored in the runtime. This guarantees that only authenticated and authorized user are able to access this information. It should be noted that this mechanism also has an interesting side effect: Different users can come to different conclusions regarding the provenance of services. In other words, two users (with different claims and event visibilities) may have different views on the same service.

To give a concrete example, reconsider the provenance graph shown in Figure 5.2. This graph was generated for some user that had access to all provenance information (e.g., user *admin*). However, claims and event visibility may restrict the visible information for specific users. For instance, users without the resource-level *Read* claim on *Service* or without the instance-level *Read* claim on Service 2 clearly would not receive any provenance information about this service. To give an example for event visibility, if the visibility of *BindingInvocationEvent*s is set to *telco1*, then other users might not see the *INVOCATIONS* node in the graph. This can be further refined, by granting user *telco1* access to `ServiceInvokedEvents` but restrict access to `ServiceInvocationFailedEvents` only to users within the user group *admins*. In that case, only information about successful invocations would be shown in the graph.

Process Provenance

So far, we have focused on the provenance of services since, to the best of our knowledge, this has not been addressed by other research projects. However, data and process provenance is also of particular importance.

In this regard, the tracking of workflow events in VRESCo (as introduced in Section 4.4) can be leveraged. Consequently, process provenance could be provided since the workflow events are persisted in the Event Database. Therefore, it is easily possible to query the information inherent to these events (e.g., which workflow instances have been started or completed, which services have been executed as part of the workflow activities, etc.). Furthermore, the user tracking events could be leveraged to implement data provenance (i.e., which data item has been produced or consumed by the different workflow activities). This is one potential extension to the presented provenance approach, which has been left for future work.

5.2.4. Evaluation

The evaluation of the presented provenance approach is done twofold: Firstly, we discuss the usefulness and advantages of our work based on the motivating example. Secondly, we show some performance results of our current prototype implementation.

First of all, we want to highlight that there are several use cases for service provenance, which are of interest for both service consumers and service providers. On the one hand, service providers often want to know if their services perform as expected regarding QoS (e.g., response time, failure rate, etc.) or the expected number of service invocations. Otherwise, corrective actions may be taken in order to achieve the expected values. Due to the combination of service management and runtime events, provenance information can also be used to investigate which service changes had negative impact on QoS. On the other hand, service provenance information is also of particular importance for service consumers, especially when it comes to service selection. As described in the motivation, if there are multiple candidate services, service consumers may want to take a look at the history of these alternatives. If a service had good performance during the last year it might be more trustworthy than services which have been recently published. In this regard, the VRESCo service mediation approach can be used to dynamically rebind to alternative services if the current service is removed from the runtime or does not fulfill the requirements any longer.

Furthermore, security issues are often neglected in current provenance approaches. Therefore, one goal of our approach represents the integrity of provenance information, as well as appropriate access control mechanisms. Firstly, we want to ensure that all provenance information is accurate, which requires that all users within VRESCo are authenticated. Secondly, and this is even more important, only authorized users must be able to access services and metadata stored in the Registry Database. This has been implemented by the claim-based access control mechanism. Finally, considering provenance information we find it crucial that producers of provenance information are able to define who is authorized to see which piece of information (i.e., different clients may have different views of the same service). Therefore, we have leveraged event visibility to provide fine-grained access control to events.

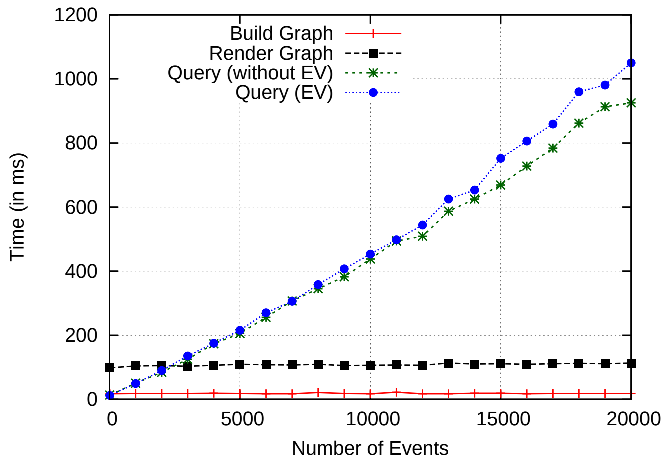


Figure 5.3.: Provenance Performance

Secondly, we describe the performance of our approach which is depicted in Figure 5.3. It illustrates how long it takes to generate the provenance graph shown in Figure 5.2 depending on the number of events that have to be considered. Again, all test results represent the average of 10 repetitive runs.

The red and black lines illustrate how long it takes to build the graph (i.e., generate the corresponding GraphViz DOT file) and render it (i.e., transform the DOT file into the desired format such as PNG) once all necessary information has been queried from the Registry Database. The lines are almost constant, which is due to the grouping of similar events in the graph. The green and blue lines depict the query performance by distinguishing whether event visibility must be evaluated. This is done by using two query issuers with different event visibility: The first user *admin* can access all events, while only 25% of all events are visible to the second user (i.e., the remaining 75% have to be sorted out). The graph shows that our approach scales linearly for several thousands of events and that the provenance queries perform well (e.g., about 1s for 20000 events). Furthermore, it can be seen that the overhead introduced when considering event visibility is acceptable (e.g., 13% for 20000 events, and 25% for 40000 events which is not shown in the figure). All results were measured on the server-side since the client's SOAP request to the Query Engine heavily depends on the network latency.

5.2.5. Related Work

After describing our provenance approach, we want to give a brief overview of related work. The provenance of electronic data has been addressed in various research efforts [121]. The focus of this research has often been on provenance in e-Science and scientific workflows [172], which led to different research prototypes such as Chimera [55]. Over the years, research on data provenance resulted in the Open Provenance Model [122] and reference architectures for provenance systems [62]. Additionally, there is existing work in the area of data provenance in service-based systems [29,154,173,185] which is discussed in more detail below. In general, these approaches address data provenance which aims at capturing the history of some piece of data generated by some process. In contrast to that, our work focuses on service provenance by maintaining the origin and history of services and associated metadata.

There are several issues when designing provenance in service-oriented systems. Tsai et al. [185] discuss the issues of data provenance in SOA systems compared to traditional data provenance techniques. Their main focus is on security, reliability and integrity of data routed through such a system. Tan et al. [181] also address security issues in SOA-based provenance systems. They use p-assertions [121], which represent specific items that document parts of a process, as foundation for their considerations. Similar to our work, they argue that access control, trust and accountability of provenance information are crucial points. In addition to that, we also address security issues in service runtime environments, which is realized using authentication based on certificates and claim-based authorization.

Rajbhandari and Walker [154] present a system that incorporates provenance into scientific workflows to capture the history of the produced data items. This history is captured by the workflow engine and recorded into a provenance database, which is structured using RDF schema. Furthermore, a provenance query service is used to query the provenance information stored in the database. Heinis and Alonso [66] present another approach to provenance of scientific data. In their approach, they focus on how provenance data can be efficiently stored and queried in the provenance database.

Another interesting work in this area is described by Simmhan et al. [173] who introduce the Karma2 system. The goal of this work is to provide provenance in data-driven workflows. The authors describe their provenance model including different provenance activities (e.g., *ServiceInvoked*, *DataProduced*, etc.). The idea is to trace workflow executions for both process provenance (i.e., which services are invoked by a process) and data provenance (i.e., which data items are produced and consumed). The architecture of Karma2 uses a Publish/Subscribe infrastructure to publish provenance activities to interested subscribers. In addition, provenance queries are provided to display provenance information using graphs. Although our notion of provenance is different, there are some similarities to our work. Both approaches use provenance queries and provenance graphs. Additionally, provenance information is sent using a Publish/Subscribe infrastructure based on WS-Eventing [199]. However, while our approach provides content-based subscriptions and complex event processing, Karma2 supports only topic-based subscriptions (i.e., subscribers can either receive all or none events for one workflow). Furthermore, Karma2 uses a modified SOAP library for collecting provenance information while our work is integrated into the VRESCo runtime. Finally, our definition of service provenance also includes service metadata and QoS attributes.

Chen et al. [29] introduce what they call augmented provenance, which is based on the idea of semantic Web services (SWS). They address process provenance in scientific workflows with a special emphasis on Grid environments. Their approach applies ontologies to model metadata at various levels of abstraction while SWS are used for capturing execution-independent metadata. Similar to our work, the authors use metadata as source for provenance. However, they focus on traditional SWS technologies while our work builds on the VRESCo metadata model. Furthermore, they do not provide provenance graphs and subscriptions.

Curbera et al. [38] present a slightly different view on provenance. They introduce the notion of business provenance in order to achieve compliance violation monitoring. The basic idea is to trace end-to-end business operations by capturing various business events, correlate these events into a provenance store, and monitor if some compliance goals are violated. The authors introduce a generic provenance data model, which can be represented in provenance graphs. These graphs are built based on the event information in the provenance store and can be queried for root cause analysis. This work is complementary to ours since the authors address business provenance using business events, while we focus on service provenance based on events raised on the service management level.

5.2.6. Conclusion

In this section, we have introduced a second interesting use case for VRESCo eventing, which makes use of the historical information stored in the Event Database. This information can be used as a means to document the origin and history of services within the runtime. Since such documented history of objects is often referred to as provenance, we introduced the notion of service provenance.

Provenance information is collected at runtime and can be retrieved using provenance queries, subscriptions or graphs. Besides describing how the three types have been realized in VRESCo, we also given concrete examples. Based on this description, the evaluation discusses the usefulness and applicability of our approach, and gives performance results for generating provenance graphs. Furthermore, we position our work among related research efforts, which have mainly focused on the provenance of data in business processes and scientific workflows.

5.3. QoS Monitoring and SLA Violation Detection

Besides notification-based rebinding and service provenance, the VRESCo Event Notification Engine also supports monitoring QoS attributes by combining two conceptually different approaches. The QoS information collected at runtime using events can then be used to detect SLA violations. These two topics represent the focus of the following section.

5.3.1. Introduction

QoS plays a crucial role in service-oriented systems. For instance, reconsider the motivating example in Section 1.1. When CPO1 integrates external services into internal business processes (e.g., Number Porting Service of CPO2), it is important to consider the quality guarantees of the service provider. In this regard, SLAs [82] are used to define the expected QoS between service consumers and service providers.

In general, QoS attributes can be classified as deterministic or non-deterministic. The former indicates that the QoS attribute is known before a service is invoked (e.g., price, security, etc.), while the latter includes all attributes that are unknown at service invocation time (e.g., response time, availability, etc.). For non-deterministic QoS attributes, monitoring approaches can be used to continuously measure current QoS values.

Conceptually, there are two main approaches for QoS monitoring: Server-side monitoring is usually accurate but requires access to the actual service implementation which is not always possible. In contrast, client-side monitoring is independent of the service implementation but the measured values may not always be up-to-date since client-side monitoring is usually done by sending probe requests (i.e., test requests that are similar to real requests).

We aim at combining the advantages of both approaches which has been realized in the VRESCo runtime environment. Therefore, we have linked an existing client-side QoS monitoring approach [163] together with server-side monitoring based on Windows Performance Counters [204]. Furthermore, event processing is used to integrate both approaches and provide means to monitor SLAs. If SLA obligations are violated, notifications are sent to interested subscribers using email or Web service notifications.

5.3.2. QoS Monitoring

In this section, we present two conceptually different monitoring approaches for Web services, which we have integrated into VRESCo. The QoS model used for this work has been introduced in Section 3.3.

Client-side Monitoring

The first approach to address QoS monitoring is done client-side using the QUATSCH tool which was introduced in [163]. The overall idea is to send probe requests to the services that should be measured. The service invocation is thereby divided into the time periods shown in Figure 3.6 that correspond to the introduced QoS attributes.

The actual monitoring is done in three phases. In the preprocessing phase, the WSDL files of the services are parsed and stubs are generated. The performance measurement code is thereby weaved into the stubs using aspect-oriented programming (AOP) [84] or interceptors (depending on the used Web service framework). In the evaluation phase, the services are executed by probing arbitrary values as input parameters. Moreover, templates can be used to provide user-defined input. Finally, the result analysis phase stores the results of the evaluation phase in a database. Interestingly, the client-side monitoring approach is indeed able to accurately measure server-side attributes such as execution time (as we will show below). In QUATSCH, this is done using low-level TCP packet sniffing and analysis of SOAP message traces. For instance, the network latency can be measured by leveraging the TCP handshake (i.e., SYN and ACK packets). More details on the trace analysis algorithm can be found in [158].

Server-side Monitoring

The main drawback of the client-side approach is the fact that monitoring is done by regularly sending probe requests (e.g., every 5 minutes). Single results should be handled with caution since they represent only snapshots (e.g., the service may be under heavy load when the probe request is sent). Decreasing the monitoring interval may mitigate this problem to some extent but this must be done carefully since short monitoring intervals (e.g., once every second) may finally affect the actual performance of the service.

Server-side monitoring addresses this problem by continuously measuring QoS attributes. Since no probe requests are needed anymore, the measured values represent "real" service invocations. However, as said above, this technique requires access to the actual Web service implementation which is not always possible in practice.

Windows Performance Counters (WPC), especially the counters regarding the Windows Communication Foundation (WCF) [146], are part of the .NET framework and provide such server-side QoS monitoring for Web services [204]. WPC supports a rich set of counters that can be measured at runtime. For our work, we focus on the following counters: *Call Duration* indicates the service invocation time which resembles *Execution Time* in the client-side approach. *Calls Per Second* defines how often a service has been invoked, while *Calls Failed Per Second* represents a similar counter for unsuccessful service invocations. Other performance counters (e.g., *Transactions Flowed Per Second*, *Security Validation and Authentication Failures*, etc.) could also be integrated easily.

As before, monitoring is done in user-defined intervals. The different performance counter values are thereby aggregated and averaged within an interval, and finally re-set at the beginning of the next interval. For instance, if a service has been invoked 3 times, the average response time of these three invocations is returned by the counter.

5.3.3. QoS/SLA Integration in VRESCo

In this section, we show how the presented client- and server-side approaches are integrated in VRESCo, and how this enables event-based SLA violation detection [115].

QoS Integration

The overall architecture of our monitoring approach is shown in Figure 5.4. The client-side monitor QUATSCH was first integrated into VRESCo. Users can specify QoS monitoring schedules following the CRON time notation to define which service (or service operation) should be monitored in which time intervals. Since this is a client-side approach, the monitor runs on a dedicated QUATSCH host as shown in the middle of the figure. The actual monitoring is then done using AOP and TCP packet analysis as described in Section 5.3.2. Once the current QoS values have been measured, they are published into VRESCo. This is done via the QoS Manager that receives the values and, in turn, publishes them as corresponding QoS events into the Event Notification Engine.

Monitoring is done regularly based on the user-defined monitoring schedules. The set of resulting QoS events represents the history of QoS information as collection of single QoS snapshots. To make these values easily accessible, they are aggregated by a QoS aggregation scheduler task on a regular basis. Finally, they are attached to the corresponding service revision (or service operation in case of *QoSOperationEvent*s).

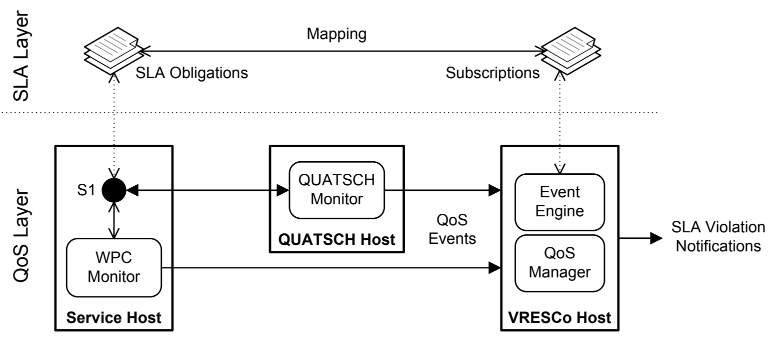


Figure 5.4.: QoS/SLA Monitoring Approach

As already discussed, the client-side approach has both strengths and weaknesses. Therefore, we decided to additionally integrate a server-side approach using WPC which is an integral part of the .NET framework. Consequently, the WPC-based approach is restricted to services implemented in .NET. The WPC monitor runs on the same host as the service (see Figure 5.4) and continuously monitors its QoS attributes. The measured values are published into the VRESCo runtime the same way as described before. For the WPC-based approach the monitoring schedules are defined in configuration files as shown in Listing 5.6. It defines which service/operation should be monitored, together with the monitoring and availability checking interval (in ms). The former describes how often the counters are retrieved while the latter is required since availability should be checked more often than other QoS attributes to get meaningful results.

```
<vresco.qosmonitoring monitoringinterval="60000" availabilitycheckinterval="5000">
 <webservices>
  <webservice wsdl="http://localhost:8013/s?wsdl">
   <operations>
    <add name="TestOperation"/>
   </operations>
  </webservice>
 </webservices>
</vresco.qosmonitoring>
```

Listing 5.6: WPC Monitoring Configuration

In general, the two approaches are independent. However, some attributes can only be measured by one of the approaches (e.g., latency and response time have to be measured from the client-side). Table 5.1 shows the QoS attributes currently measured in VRESCo and depicts which approach has been taken for which attribute. *Throughput* and *Calls Per Second* seem to refer to the same QoS attribute. However, the first represents the maximum number of requests that can be processed, while the latter indicates the number of invocations that really occurred.

QoS Attributes	Monitored by
Execution Time	QUATSCH & WPC
Response Time	QUATSCH
Latency	QUATSCH
Availability	QUATSCH & WPC
Throughput	QUATSCH
Calls Per Second	WPC
Calls Failed Per Second	WPC

Table 5.1.: QoS Attributes

It can be seen that two attributes are measured by both approaches. For both *Availability* and *Execution Time*, the WPC-based approach is usually more accurate since it does not need to send probe requests, but represents the values of real invocations. However, we still monitor using both approaches since the measured values may be different. In Section 5.3.4, we briefly discuss why this combination is useful and give some concrete examples.

SLA Monitoring

QoS monitoring approaches, as introduced in the last section, represent an essential foundation for SLAs, which define the expected QoS between service providers and consumers. In this section, we describe the SLA monitoring approach in VRESCo, and how clients can react to SLA violations. This approach is based on the VRESCo Event Engine and is depicted in the top part of Figure 5.4. To give a brief overview, simple SLA obligations can be attached to services. This is done using the Publishing Service that also allows to temporary start and stop SLA monitoring. These obligations are then transformed to subscriptions specified in the Esper Processing Language (EPL) [45] since VRESCo eventing is based on the Esper Engine. This engine does the actual matching between subscriptions and events. Finally, when such matches occur the subscribers are notified about the corresponding SLA violation.

Frameworks such as WSLA [82] have been proposed for defining complex SLAs, but they are rarely used in practice. Therefore, we decided to define simple SLA obligations representing guarantees on the QoS attributes, which are shown in Table 5.2. First of all, obligations can be either attached to service operations or revisions. Every obligation is valid only within a given period of time after which it expires. The property name represents the QoS attribute to monitor (e.g., response time), while logical operator and property value are used to define threshold values (e.g., < 500 ms). Aggregation functions (e.g., sum, max, avg, median, etc.) can further be defined on multiple QoS events. Obligations also define the notification mechanism and the address used for violation notifications (e.g., email or WS-Eventing notifications). Finally, sliding window operators can be used to define the time period to consider for the QoS events (e.g., the last 10 events or events within the last 5 minutes).

Property	**Description**
Id	Identifier of the obligation
RevisionId	Identifier of the service revision
OperationId	Identifier of the service operation
StartDate	Start date of the obligation
EndDate	End date of the obligation
PropertyName	QoS attribute to monitor
Aggregation	Aggregation function on property
LogicalOperator	Logical operator used for comparison
PropertyValue	Threshold value used for comparison
ReactionType	Notification mechanism to use
ReactionAddress	Address of the subscriber
WindowType	Type of the sliding window operator
WindowValue	Value of the sliding window operator

Table 5.2.: SLA Obligations

To give concrete examples, a simple SLA obligation could define that the availability of revision 23 should be greater than 0.99. This obligation is transformed to the following EPL expression (please note that logical operators must be inverted since subscriptions represent violation conditions):

```
select * from QoSRevisionEvent
where Revision.Id=23 and Property='Availability' and DoubleValue<=0,99
```

A more complex SLA obligation could define that operation 47 of service revision 11 should have an average response time less than 500 ms within the last 24 hours. Besides the sliding window operator (`win:time`) this SLA obligation uses univariate statistics on event streams (`.stat:uni` and `average`) which are provided by Esper:

```
select * from QoSOperationEvent\\
(Revision.Id=11 and Operation.Id=47 and Property='ResponseTime')\\
.win:time(24 hours).stat:uni('DoubleValue')
where average>=500
```

Once an SLA violation is detected, notifications are sent using email or Web service notifications to the specified address. The subscribers can then react accordingly, for instance by rebinding to functionally equal services [117]. In this regard, the SLA violation mechanism can also be used by service providers to monitor if services perform as intended. SLA violation notifications could then automatically trigger to start new instances of this service and publish them into VRESCo. Such scenarios and ways to define SLA penalty models are part of our future work.

5.3.4. Evaluation

To evaluate our approach, we compare the accuracy of both monitoring approaches in terms of execution time and availability as two exemplary values that can be measured by both QUATSCH and WPC. Based on these findings, we discuss why a combination of both approaches is useful and highlight some of its advantages and disadvantages.

Our evaluation environment consists of a server hosting VRESCo and a set of C#/.NET dummy services that have a configurable execution time and a variable availability (3 downtimes of 30 min, 2 min and 10 min length, and simulated network problems with short interruptions between 19:00 and 19:20). Additionally, QUATSCH is hosted on a VMWare image running on a different server in the LAN.

Figure 5.5 and 5.6 depict the results of our monitoring experiments where QUATSCH probes every 5 minutes whereas WPC measures every minute. We further use soapUI [48] to simulate clients that continuously invoke the test services. The different measurement intervals are based on the fact that QUATSCH sends real invocations to probe a service, while WPC has lower overhead because it queries performance counters provided by the operating system.

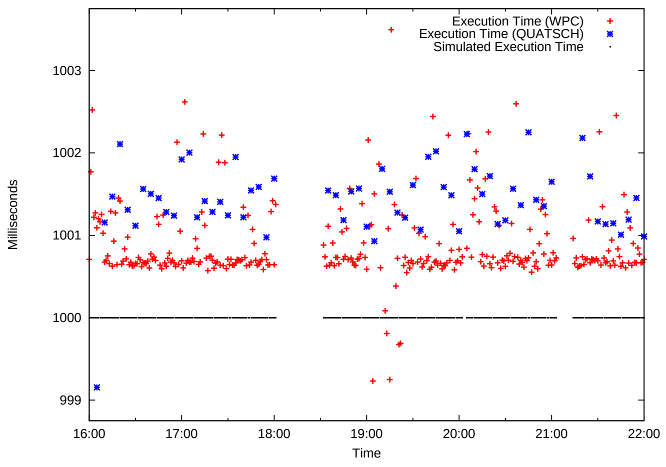


Figure 5.5.: Execution Time Comparison

Figure 5.5 depicts the measured execution time over 6 hours. The results show that both approaches are pretty accurate. The deviation from the simulated execution time (1 sec) is less than 2 ms for most measurements. The values for QUATSCH indicate that execution time can be indeed measured from the client-side. Additionally, it can be seen that WPC is more accurate because it represents the average execution time measured on the server-side. The gap between simulated execution time and average value measured by WPC is 0.88 ms, which is partly caused by internal processing of the test services (e.g., threading, console output, etc.).

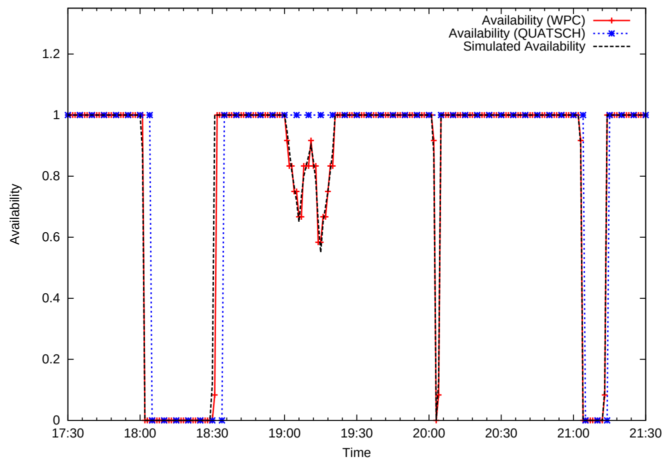


Figure 5.6.: Availability Comparison

Figure 5.6 shows the simulated and measured availability of the test services. It can be observed that WPC detects downtimes faster, which is due to the shorter monitoring interval. WPC further divides this interval into availability checking intervals (5 sec). Therefore, the availability of one interval can also be measured (e.g., at 18:31 and 20:02). In contrast to that, QUATSCH does not have this fine-grained distinction (i.e., availability is either 0 or 1). As a result, QUATSCH does not recognize the short downtime at 20:02. The same is true for the timespan between 19:00 and 19:20 where WPC is quite accurate whereas QUATSCH does not detect this at all. It should be noted that the same behavior could be observed when the execution time in Figure 5.5 is varying.

Nonetheless, combining both approaches is still useful. Firstly, some QoS attributes can only be measured from the client-side (e.g., latency). Secondly, it is possible to distinguish between client- and server-side view on some QoS attributes (e.g., availability). For instance, if there is no network connection on the QUATSCH monitoring host, client-side availability decreases even if the service is running. However, this can be verified by the WPC approach. Thirdly, bogus server-side measurements can be detected by the client-side approach, by comparing measured QoS values over a longer period of time. Fourthly, another dimension to client-side monitoring could be added by integrating actually perceived QoS values on the client-side (in addition to the measured values of the QUATSCH probe requests). However, the combination of both approaches also has some drawbacks. For instance, clients must agree to install monitoring software which may not always be the case.

We have shown that the accuracy of the monitoring approaches makes them suitable for SLA monitoring. As depicted in Figure 4.8, the event throughput is high enough for the expected number of services (e.g., 2000 services with the same monitoring intervals as above). Therefore, it is easily possible to subscribe to SLA violations and react adaptively if needed.

5.3.5. Related Work

Several different QoS models have been proposed in literature (e.g., [107,155,212]). However, most approaches do not discuss how QoS can be monitored. An overview of QoS monitoring approaches for Web services is presented by Thio et al. [182]. The authors discuss various techniques such as low-level sniffing or proxy-based solutions. The prototype presented in their paper adopts an approach where the SOAP library 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.

QoS monitoring has been an active research area for years which is not only focused on Web service technology. For instance, Garg et al. [59] present the WebMon system that aims at monitoring the performance of Web transactions using a sensor-collector architecture. Similar to our work, their approach correlates client- and server-side response times which are measured by different components. In their work, the question is whether to instrument the Web server or the Web browser for doing the performance measurements.

There are many existing approaches for SLA monitoring and violation detection (e.g., [28,100, 153,175] just to name a few). Skene et al. [175] introduce SLAng which is a general SLA language not only focused on Web services, but targeted to distributed systems and applications with reliable QoS characteristics. The language is modeled in UML while the syntax is defined using XML schema. The authors further define a model for all parties and services involved in such agreement. The actual constraints in the SLAs are then defined using the Object Constraint Language (OCL). Finally, UML profiles are used to extend SLAng with a QoS catalog which enables to define QoS characteristics of services.

Raimondi et al. [153] describe an SLA monitoring system that translates timeliness constraints such as latency or availability of SLAs into timed automata. The execution traces of Web services are then verified using these timed automata. Their approach is implemented as handler for the open source engine Axis [6] while they use SLAng for defining SLAs.

Lodi et al. [100] describe a middleware for SLA-driven clustering of QoS-aware application servers. They use XML for defining SLAs, which was inspired by SLAng. The architecture consists of three components: The Configuration Service is responsible for managing the QoS-aware cluster, the Monitoring Service observes the application at runtime to detect violations of SLAs, and the Load Balancing Service intercepts client requests to balance them among different cluster nodes. If the cluster is mainly idle or close to breach the SLA (e.g., the response time converges to the upper bound), it is reconfigured (i.e., add/release nodes).

Chau et al. [28] present a similar approach for modeling and event-based monitoring of SLAs which is part of the eQoSystem project. The SLA model refines the WSLA specification: SLAs consist of multiple SLOs and use various metrics that indicate different measurement aspects of a process. Furthermore, action handlers can be defined to react when SLOs are violated. Similar to our work, the SLA monitoring approach is based on events. These events are assumed to be emitted by the business process and contain a snapshot of the current process state. In contrast to that, our QoS events focus on the service- and operation-level. Furthermore, we additionally address how QoS attributes of Web services can be monitored from client- and server-side.

Moser et al. [123] present the VieDAME system that aims at non-intrusive monitoring of BPEL processes. The main idea is to monitor QoS of processes and provide service adaptation by introducing different selection strategies for services. For instance, this can be used to implement load balancing. Furthermore, if services do not perform well, they can be replaced with alternative services. If the alternatives do not have the same interfaces, SOAP messages are adapted using XSLT transformations. Wetzstein et al. [191] monitor and analyze the QoS of BPEL processes to observe key performance indicators (KPIs) and find factors that influence process performance. Their approach uses process events published by the process engine and QoS information (e.g., service response time) measured by the QoS monitor, while the dependencies of KPIs to process and QoS metrics are depicted in tree structures.

Finally, it should be noted that there are other options for handling QoS besides monitoring. For instance, Bianculli et al. [21,22] propose the ReMan infrastructure for reputation management of composite Web services. The basic idea is to build service rankings based on the client-perceived QoS of external services. Furthermore, notifications can be sent if the rankings change. This work is complementary to the Quality of Experience (QoS) approach introduced in VRESCo [91]. While ReMan focuses on how reputation of composite services can be estimated and how clients are notified if service rankings change, our work uses tagging as mechanism for receiving structured and unstructured user feedback. Furthermore, tag merging and trust relationships are used to aggregate this information.

5.3.6. Conclusion

Monitoring QoS attributes of Web services is an essential aspect to enforce SLAs established among business partners. In this section, we have shown that a combination of client- and server-side QoS monitoring can be beneficial regarding the overall monitoring capabilities since both approaches have strengths and weaknesses. These monitoring capabilities combined with a powerful Web service runtime enable an event-based detection of SLA violations for Web services, while subscribers can react appropriately to such violations. We have evaluated the accuracy of both QoS monitoring approaches, which has confirmed our assumption that server-side monitoring is more accurate than client-side monitoring. However, we have further discussed a number of reasons why combining both approaches can still be useful in practice.

For future work, we plan to automatically react to SLA violations, such as deploying new service instances on-the-fly or dynamically increase virtual machine capabilities (that are often used to host services). Furthermore, we envision to measure and publish the actual response times at the client-side, in addition to the measured QUATSCH probe requests.

5.4. Service Pricing and Penalty Models

The use cases presented so far have been realized within the scope of this thesis. Furthermore, there are several other scenarios that can leverage the VRESCo eventing support. We briefly present two of them in the following two sections.

Service pricing receives increasing attention as more and more services become available. In this regard, service usage can be automatically billed to the user account according to the agreed pricing model. To give a concrete example, reconsider the CPO case study introduced in Section 1.1. In this example, CPO1 provides *Messaging Services* for sending SMS messages, which can only be used by customers. However, such services are typically not free but have to be paid by its users. This raises the need for appropriate pricing models.

Service pricing models [43] can range from simple models with fixed prizes to complex models where prizes can be calculated in a dynamic manner. For instance, the basic *flat-rate* model allows users unlimited access to services for given period of time. In contrast to this, more flexible pricing models include *pay-per-use* where users pay a certain amount of money for every service invocation (or for a certain amount of invocations).

In VRESCo, such billing information can be easily aggregated based on the information stored in the Event Database, since every invocation is represented by *ServiceInvokedEvent*s. Therefore, the Billing Service simply has to query the Event Database for every invocation within a given time period, and apply this information to the agreed service pricing model.

Furthermore, service pricing may also be influenced by the SLAs defined between partners, and possibly result in penalties if providers cannot meet the SLAs. For instance, if the service response time is beyond a given threshold, service providers may be bound by contract to make penalty payments to service consumers. The same is true if service invocations fail, which is recorded by *ServiceInvocationFailedEvent*s.

Similar to pricing models, penalty models can be defined in order to specify the effects if SLAs are violated. Using the information stored in events, this can also include complex penalty models (e.g., pay 10$ if the average response of service X within the last month goes beyond a given threshold). Finally, pricing and penalty models can be combined, which allows flexible derivation of pricing models based on dynamically negotiated SLAs. For instance, users may get 5 free service invocations for every failed invocation.

5.5. Event-based Composition

Another interesting point for future work in VRESCo is to link together the Composition Engine [158] and Eventing Engine in order to provide what we call *event-based composition*. However, it should be noted that this is conceptually different to the work of Hu et al. [68], who describe how Publish/Subscribe can be used for automatic service composition.

Currently, the VRESCo Composition Engine statically builds the composition using the services and metadata stored in the Registry Database. Especially in large-scale settings, the execution time of the query represents an essential part of the overall time [159]. To reduce this overhead, event notifications could be used by subscribing to these queries and perform re-optimization of compositions, instead of running the whole composition process from scratch.

In this regard, the re-optimization process is then triggered by events at runtime. For instance, if the response time of a service within a composition changes significantly, the Composition Engine may be triggered to re-optimize the composition. In this way, not well-performing services could be replaced by services with better QoS, that implement the same feature.

5.6. Conclusion

This section has presented some application scenarios of VRESCo eventing. Some of them have been implemented within the scope of this thesis, while others are proposed for future work.

Firstly, the initial motivation for the VRESCo Event Notification Engine was to use event notifications to trigger dynamic rebinding of service proxies. Using events for this purpose obviously has advantages compared to the other rebinding strategies introduced earlier.

Secondly, the information inherent to historical events stored in the Event Database can be used as source of service provenance. The aim of service provenance is to document the origin and history of services in the Registry Database, which can be visualized using provenance graphs. Furthermore, access control is used to support selective service provenance.

Thirdly, event processing can be used for QoS/SLA monitoring. In this regard, we have presented two different QoS monitoring techniques that were combined using the VRESCo eventing support. Furthermore, the complex event processing mechanisms of the Event Engine also enable event-based SLA violation detection.

Finally, there are several use cases that are enabled by this work, but have not been implemented so far. We have briefly summarized two of them, namely service pricing models and event-based composition of services. To sum up, this section has shown the usefulness and applicability of the VRESCo Event Notification Engine.

Chapter 6.

Conclusion

This chapter concludes the thesis and summarizes what has been presented. Furthermore, the research questions raised in Chapter 1 are revisited, by discussing how this work provides corresponding answers. Finally, we briefly mention future work and research directions that are enabled by this thesis.

6.1. Summary

Service-oriented Architecture has received attention from both academia and industry in the past years. Following this paradigm, loosely-coupled software services are provided and can be composed into business processes to achieve higher-level functionality. Web services represent the most common and widely-used realization of SOA. Within the past years, several standards and specifications have been proposed and a complete Web services stack has emerged.

In this thesis, we have presented several challenges that we see in SOC research and practice. More precisely, we argue that current service-oriented solutions are often not as flexible and loosely-coupled as initially intended by the SOA model. Among others, these challenges include service metadata and service querying, QoS-based dynamic binding and invocation, as well as service mediation. We have introduced the VRESCo service runtime environment that addresses these challenges.

The focus of the second part of this thesis has been put on one challenge, which represents asynchronous event notifications in service-oriented systems. After describing the state of the art regarding event processing and SOC, we have highlighted the problems of current solutions. Based on these findings, we have integrated complex event processing support into VRESCo.

Finally, we have given several application scenarios for event notifications in service-oriented systems. For instance, event notifications can be used for dynamic binding of services, which has clear advantages over other rebinding strategies. Besides describing some application scenarios that have been investigated within the scope of this thesis, we also point to future research directions enabled by this work.

6.2. Research Questions Revisited

After this brief summary, we want to evaluate the research questions raised in Chapter 1. More precisely, we show how the results of this thesis contribute to the three research questions.

- **Q1:** *What are current challenges of flexible and adaptive SOC infrastructures in general, and service registries in particular? How can these challenges be addressed within a coherent system?*

  In Section 1.2 we have highlighted the challenges we see in SOC. Moreover, we have investigated several approaches with regard to these challenges in Section 2.4.

  Based on these results, we argue that many approaches provide only limited support for several challenges. Therefore, we have introduced the novel service runtime environment VRESCo in Chapter 3. Among others, one of the main objectives is QoS-based dynamic binding and invocation of services. This also includes mediation mechanisms if functionally equal services provide different technical interfaces. As a result, clients can dynamically bind between different services that perform the same task. For instance, if the response time of some service goes beyond a given threshold, service proxies can be automatically rebound to an alternative service with better QoS.

  This requires a QoS-aware environment which is provided using QoS monitoring techniques. Furthermore, service metadata are used to map abstract service descriptions to concrete service implementations. This enables to define if two services perform the same task, and to specify the concrete technical service interface, which is needed for service mediation. Besides that, service querying is needed in order to query services and associated service metadata at runtime. Finally, other important topics include support for service versioning in order to maintain multiple revisions of a service, and access control mechanisms to gain access to VRESCo only to authorized parties.

  The evaluation in Section 3.9 has shown that the performance of the VRESCo components is adequate in large-scale settings. Moreover, an end-to-end scenario has been exemplified to highlight the effects of single components on the overall system performance. In contrast to simple keyword-based matching of most service registries, VRESCo provides a powerful querying framework supporting view-based querying, different strategies and optional/weighted constraints. Furthermore, the rebinding, mediation and versioning mechanisms facilitate the implementation of flexible and adaptive service-based applications, as opposed to existing service frameworks (see qualitative comparison in Table 2.1).

- **Q2:** *How can event processing principles be seamlessly integrated in SOC infrastructures? What are specific challenges and approaches?*

  In Chapter 2 we have introduced the state of the art concerning event processing in SOC by describing different standards and specifications, as well as eventing support in Web service registries. Currently, there are various similar specifications which are competing, and it is unclear which specification will succeed.

Most approaches have in common to support only basic events (e.g., new service has been published, old service has been deleted, etc.). However, we argue that more complex events are of particular interest. This includes support for event patterns (e.g., *A* happens before *B*) and sliding window operators (e.g., consider only events within the last 2 hours).

Consequently, one of the main objectives of this thesis is to introduce such complex event processing mechanism, which is described in detail in Chapter 4. The VRESCo eventing support builds on the open source engine Esper, that provides several complex event processing features. We have introduced various event types that can be of interest in service-oriented systems (e.g., service management and QoS, process or business events). Furthermore, events are persisted into the Event Database, while the VQL querying framework can be used to search in historical events. Finally, since events may include sensitive information, appropriate access control mechanisms have been investigated that gain access to events only to authorized parties.

The evaluation in Section 4.10 has shown that the VRESCo Event Engine can deal with several hundreds of events per second. In general, this is adequate for various application scenarios. Furthermore, the overhead of the Event Engine depicted in Figure 4.9 seems acceptable when considering the new possibilities opened up by the eventing support. Finally, concrete usage examples have emphasized the expressiveness of the subscription language. In this regard, most of the complex event processing mechanisms are not supported by current service infrastructures that usually provide only basic events.

- **Q3:** *Which application scenarios are enabled by this work? What would be more difficult or less efficient to realize without event processing support?*

  The previous question has addressed the general event processing mechanisms of service runtime environments, without pointing to actual application scenarios. We argue that these mechanisms can be used to address several SOC challenges more efficiently, which highlights the usefulness and applicability of our approach.

  First of all, event notifications can be used during dynamic binding of services. In the first prototype, rebinding was done proactively (e.g., check the binding periodically). Using notifications, subscriptions can define when the rebinding should be triggered (e.g., if the response time of service *X* goes beyond 500 ms, etc.). Together with the VRESCo mediation approach this increases the flexibility of service-oriented solutions, which can automatically react and adapt to changing environments. To support this claim, Figure 5.1 has shown that the *OnEvent* rebinding strategy performs better than other strategies.

  Even though *OnEvent* rebinding was the main motivation for the Event Engine in the first place, there are several other use cases that have been investigated. For instance, the events stored in the Event Database can be used as source for provenance information since they represent the origin and history of services. This information can be retrieved using provenance queries, graphs or subscriptions (e.g., during service selection).

In contrast to existing approaches that address data and process provenance, VRESCo focuses on the provenance of services. Furthermore, our approach was integrated into an existing infrastructure instead of providing a dedicated provenance framework. Besides giving concrete provenance examples, Figure 5.3 has shown that generating provenance graphs in VRESCo can be done efficiently for several thousands of events.

Besides that, QoS/SLA monitoring represents another interesting use case for event processing. The VRESCo eventing mechanism has been leveraged to combine client- and server-side QoS monitoring techniques, as well as event-based SLA violation detection. These violations may be used to automatically trigger reactions (e.g., rebind to another service). In Section 5.3 we have compared client- and server-side monitoring techniques. Even though server-side monitoring is usually more accurate, we have shown that combining both techniques into a comprehensive monitoring framework can still be beneficial.

6.3. Future Research Directions

The work presented in this thesis opens up new possibilities for future work. To conclude, we give some examples and future research directions that we envision based on this thesis.

- **Real-Life Case Study:** The evaluation of Chapter 3 has depicted the performance of the VRESCo components, together with an end-to-end evaluation using a simple case study.

  In the next step, it would be interesting to apply VRESCo in a large-scale and real-life case study. After conversations with different software companies, our view on the SOC challenges has been encouraged since many of these problems are also present in practice. Therefore, such case study additionally aims at bridging the gap between research and practice. In this regard, we also plan to make VRESCo available as open source in order to allow interested researchers and practitioners to evaluate our work. Finally, a comparative case study would also be worthwhile to highlight the benefits of VRESCo compared to traditional Web service technology.

- **Federation of Registries:** Currently, the VRESCo runtime uses a centralized architecture where all services are stored in one database. In this regard, it would be interesting to provide support for federation of registries using appropriate distribution techniques.

  On the one hand, services and associated metadata could be maintained in multiple registries, and changes in one registry instance must be propagated to the others. Others have already studied the problems that arise when federating registries [14, 42, 174]. Besides that, the distribution of other VRESCo components requires more attention. For instance, it must be investigated how the complex event processing infrastructure can be distributed in order to still guarantee efficient matching. In this regard, Cugola and Margara introduce the RACED middleware [36] that addresses distributed detection of complex events by using an event definition language similar to Esper EPL.

Furthermore, another interesting topic would be to deploy VRESCo into a cloud computing environment [65]. On the one hand, services and associated metadata could be stored in the cloud instead of a registry database [24]. On the other hand, the VRESCo services could also be deployed in the cloud for scalability reasons.

- **Adaptive Rebinding Strategies:** In this thesis, we have introduced different rebinding strategies that can be used to dynamically bind and invoke services. According to our client-side approach, service consumers can define the point in time when rebinding should be reconsidered (e.g., *Periodic*, *OnInvocation*, *OnEvent*, etc.).

  Our evaluation has shown an interesting effect in large-scale settings, which is caused by this client-side approach. If all service consumers want to invoke the best available service, the quality of this service may get worse due to the high number of requests (or may even break down completely). As a result, clients would be rebound to the second best service and the same effect starts over again.

  To prevent such situations, the rebinding strategies could be made adaptive. In other words, the rebinding decision of one service proxy may depend on the binding of other service proxies that use the same feature (or different features on the same host). In this regard, it should be investigated how software load balancers can be used to evenly spread the load among services that perform the same task [100, 152].

- **Further Eventing Applications:** In this thesis, we have presented three usage scenarios that have been implemented based on the VRESCo Event Engine, namely notification-based rebinding, service provenance and event-based QoS/SLA monitoring.

  Furthermore, we have mentioned additional usage examples enabled by VRESCo eventing, which have been left for future work. This includes service pricing and penalty models and event-based composition. These topics have been discussed in more detail in the previous chapter. Another interesting aspect would be to detect recurring event patterns in past event data, in order to predict the behavior of a system in the future.

- **SLA Violation Prediction:** In service-oriented systems, SLAs are often used to define the contractual obligations between service consumers and providers. In general, service providers aim at preventing SLA violations since this usually entails penalty payments and decreases customer satisfaction. Consequently, it is of particular importance to be able to predict SLA violations before they actually happen. Service providers can then try to react in order to avoid the impending violation (e.g., by adapting the business process).

  In this regard, our ongoing and future work includes to investigate how machine learning techniques can be used to predict SLA violations based on historical QoS data [94]. This historical data is based on QoS events that are published by the workflow engine. Once SLA violations are predicted, the question is how the process can be automatically adapted to meet the SLA. However, in some cases it might be preferred to violate the SLA since the penalty payment is less expensive than the necessary adaptation actions.

Bibliography

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

[2] Altova, Inc. *MapForce*, 2009. `http://www.altova.com/mapforce.html` (Last accessed: November 30, 2009).

[3] Apache Software Foundation. *Web Services Invocation Framework*, September 2003. `http://ws.apache.org/wsif` (Last accessed: August 31, 2009).

[4] Apache Software Foundation. *Muse v2.2*, March 2007. `http://ws.apache.org/muse/` (Last accessed: October 3, 2009).

[5] Apache Software Foundation. *Savan/C v0.9*, May 2007. `http://ws.apache.org/savan/c/` (Last accessed: October 3, 2009).

[6] Apache Software Foundation. *Axis2 v1.5.1*, October 2009. `http://ws.apache.org/axis2/` (Last accessed: October 30, 2009).

[7] Apache Software Foundation. *CXF v2.1.7*, October 2009. `http://cxf.apache.org/` (Last accessed: October 30, 2009).

[8] Apache Software Foundation. *Lucene v2.9*, September 2009. `http://lucene.apache.org/` (Last accessed: October 3, 2009).

[9] Apache Software Foundation. *Servicemix v3.3.1*, June 2009. `http://servicemix.apache.org/` (Last accessed: October 3, 2009).

[10] Apache Software Foundation. *Tuscany v2.0*, July 2009. `http://tuscany.apache.org/` (Last accessed: October 3, 2009).

[11] Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. STREAM: The Stanford Data Stream Management System. In *Data Stream Management: Processing High-Speed Data Streams*, M. Garofalakis, J. Gehrke, and R. Rastogi, Eds. Springer, 2010. `http://infolab.stanford.edu/~usriv/papers/streambook.pdf` (forthcoming).

[12] Baresi, L., Ghezzi, C., and Mottola, L. On Accurate Automatic Verification of Publish-Subscribe Architectures. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)* (2007), IEEE Computer Society, pp. 199–208. DOI: 10.1109/ICSE.2007.57.

[13] Baresi, L., Ghezzi, C., and Zanolin, L. Modeling and Validation of Publish/Subscribe Architectures. In *Testing Commercial-off-the-shelf Components And Systems*, S. Beydeda and V. Gruhn, Eds. Springer Verlag, 2005, pp. 273–292. DOI: 10.1007/3-540-27071-X_13.

[14] Baresi, L., and Miraz, M. A Distributed Approach for the Federation of Heterogeneous Registries. In *Proceedings of the 4th International Conference on Service-oriented Computing (ICSOC'06)* (December 2006), Springer, pp. 240–251. DOI: 10.1007/11948148_20.

[15] Belokosztolszki, A., Eyers, D. M., Pietzuch, P. R., Bacon, J., and Moody, K. Role-Based Access Control for Publish/Subscribe Middleware Architectures. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)* (2003), ACM, pp. 1–8. DOI: 10.1145/966618.966622.

[16] Benatallah, B., Casati, F., Grigori, D., Nezhad, H. R. M., and Toumani, F. Developing Adapters for Web Services Integration. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering (CAiSE'05)* (2005), Springer, pp. 415–429. DOI: 10.1007/11431855_29.

[17] Bennett, K. H., and Rajlich, V. T. Software Maintenance and Evolution: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE'00)* (New York, NY, USA, 2000), ACM, pp. 73–87. DOI: 10.1145/336512.336534.

[18] Bhargavan, K., Fournet, C., and Gordon, A. D. A Semantics for Web Services Authentication. *Theoretical Computer Science 340*, 1 (2005), 102–153. DOI: 10.1016/j.tcs.2005.03.005.

[19] Bhargavan, K., Fournet, C., and Gordon, A. D. Verifying Policy-Based Web Services Security. *ACM Transactions on Programming Languages and Systems (TOPLAS) 30*, 6 (2008), 1–59. DOI: 10.1145/1391956.1391957.

[20] Bhola, S., Strom, R. E., Bagchi, S., Zhao, Y., and Auerbach, J. S. Exactly-once Delivery in a Content-based Publish-Subscribe System. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)* (2002), IEEE Computer Society, pp. 7–16. DOI: 10.1109/DSN.2002.1028881.

[21] Bianculli, D., Binder, W., Drago, L., and Ghezzi, C. Transparent Reputation Management for Composite Web Services. In *Proceedings of the 6th International Conference on Web Services (ICWS'08)* (2008), IEEE Computer Society, pp. 621–628. DOI: 10.1109/ICWS.2008.39.

[22] Bianculli, D., Binder, W., Drago, M., and Ghezzi, C. ReMan: A Pro-active Reputation Management Infrastructure for Composite Web Services. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)* (May 2009), IEEE Computer Society, pp. 623–626. DOI: 10.1109/ICSE.2009.5070571.

[23] Bodoff, D., Ben-Menachem, M., and Hung, P. C. Web Metadata Standards: Observations and Prescriptions. *IEEE Software 22*, 1 (2005), 78–85. DOI: 10.1109/MS.2005.25.

[24] Brantner, M., Florescu, D., Graf, D., Kossmann, D., and Kraska, T. Building a Database on S3. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)* (2008), ACM, pp. 251–264. DOI: 10.1145/1376616.1376645.

[25] Brown, K., and Ellis, M. Best Practices for Web Services Versioning, January 2004. `http://www-128.ibm.com/developerworks/webservices/library/ws-version/` (Last accessed: October 3, 2009).

[26] Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems 19*, 3 (Aug. 2001), 332–383. DOI: 10.1145/380749.380767.

[27] Chappell, D. *Enterprise Service Bus*. O'Reilly Media, Inc., 2004.

[28] Chau, T., Muthusamy, V., Jacobsen, H.-A., Litani, E., Chan, A., and Coulthard, P. Automating SLA Modeling. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'08)* (2008), ACM, pp. 126–143. DOI: 10.1145/1463788.1463802.

[29] Chen, L., Yang, X., and Tao, F. A Semantic Web Service Based Approach for Augmented Provenance. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI'06)* (2006), IEEE Computer Society, pp. 594–600. DOI: 10.1109/WI.2006.25.

[30] Clemm, G., Amsden, J., Ellison, T., Kaler, C., and Whitehead, J. RFC3253: Versioning Extensions to WebDAV (Web Distributed Authoring and Versioning), March 2002. `http://tools.ietf.org/html/rfc3253` (Last accessed: December 9, 2009).

[31] Codehaus. *XFire*, 2007. `http://xfire.codehaus.org/` (Last accessed: August 31, 2009).

[32] Codeplex. *Quickgraph v3.1*, Jan. 2009. `http://www.codeplex.com/quickgraph` (Last accessed: November 30, 2009).

[33] Conradi, R., and Westfechtel, B. Version Models for Software Configuration Management. *ACM Computing Surveys 30*, 2 (1998), 232–282. DOI: 10.1145/280277.280280.

[34] Cugola, G., and Di Nitto, E. On Adopting Content-Based Routing in Service-Oriented Architectures. *Information and Software Technology 50*, 1–2 (Jan. 2008), 22–35. DOI: 10.1016/j.infsof.2007.10.004.

[35] Cugola, G., Di Nitto, E., and Fuggetta, A. The JEDI Event-Based Infrastructure and its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering 27*, 9 (2001), 827–850. DOI: 10.1109/32.950318.

[36] Cugola, G., and Margara, A. RACED: An Adaptive Middleware for Complex Event Detection. In *Proceedings of the 8th Workshop on Adaptive and Reflective Middleware (ARM'09)* (2009), ACM, pp. 1–6. DOI: http://dx.doi.org/10.1145/1658185.1658188.

[37] Cugola, G., and Picco, G. P. REDS: A Reconfigurable Dispatching System. In *Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM'06)* (2006), ACM, pp. 9–16. DOI: 10.1145/1210525.1210530.

[38] Curbera, F., Doganata, Y. N., Martens, A., Mukhi, N., and Slominski, A. Business Provenance - A Technology to Increase Traceability of End-to-End Operations. In *Proceedings of the 16th International Conference on Cooperative Information Systems (CoopIS'08)* (Nov. 2008), Springer, pp. 100–119. DOI: 10.1007/978-3-540-88871-0_10.

[39] Decker, G., Kopp, O., Leymann, F., and Weske, M. BPEL4chor: Extending BPEL for Modeling Choreographies. In *Proceedings of the 5th International Conference on Web Services (ICWS'07)* (July 2007), IEEE Computer Society, pp. 296–303. DOI: 10.1109/ICWS.2007.59 (`http://www.bpel4chor.org/`).

[40] Di Penta, M., Esposito, R., Villani, M. L., Codato, R., Colombo, M., and Di Nitto, E. WS Binder: A Framework to Enable Dynamic Binding of Composite Web Services. In *Proceedings of the International Workshop on Service-oriented Software Engineering (SOSE'06)* (2006), ACM Press, pp. 74–80. DOI: 10.1145/1138486.1138502.

[41] Dingel, J., Garlan, D., Jha, S., and Notkin, D. Reasoning About Implicit Invocation. In *SIGSOFT Software Engineering Notes* (November 1998), vol. 23, ACM Press, pp. 209–221. DOI: 10.1145/291252.288312.

[42] Dustdar, S., and Treiber, M. View Based Integration of Heterogeneous Web Service Registries – the Case of VISR. *World Wide Web Journal 9*, 4 (2006), 457–483. DOI: 10.1007/s11280-006-8561-3.

[43] Eckert, J., Ertogrul, D., Papageorgiou, A., Repp, N., and Steinmetz, R. The Impact of Service Pricing Models on Service Selection. In *Proceedings of the 4th International Conference on Internet and Web Applications and Services (ICIW'09)* (2009), IEEE Computer Society, pp. 316–321. DOI: 10.1109/ICIW.2009.53.

[44] Erl, T. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[45] EsperTech. *Esper Reference Documentation*, 2009. `http://esper.codehaus.org/` (Last accessed: October 27, 2009).

[46] Eugster, P. Type-based Publish/Subscribe: Concepts and Experiences. *ACM Transactions on Programming Languages and Systems (TOPLAS) 29*, 1 (2007), 6. DOI: 10.1145/1180475.1180481.

[47] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The Many Faces of Publish/Subscribe. *ACM Computing Survey 35*, 2 (2003), 114–131. DOI: 10.1145/857076.857078.

[48] Eviware. *soapUI*, 2009. `http://www.soapui.org/` (Last accessed: December 3, 2009).

[49] Felix, P., and Ribeiro, C. A Scalable and Flexible Web Services Authentication Model. In *Proceedings of the 2007 ACM Workshop on Secure Web Services (SWS'07)* (2007), ACM, pp. 66–72. DOI: 10.1145/1314418.1314429.

[50] Fiadeiro, J. L., and Lopes, A. A Formal Approach to Event-Based Architectures. In *Proceedings of the 6th International Conference on Fundamental Approaches in Software Engineering (FASE 2006)* (March 2006), Springer-Verlag, pp. 18–32. DOI: 10.1007/11693017_4.

[51] Fidler, E., Jacobsen, H.-A., Li, G., and Mankovski, S. The PADRES Distributed Publish/-Subscribe System. In *Proceedings of the 8th International Conference on Feature Interactions in Telecommunications and Software Systems (FIW'05)* (2005), S. Reiff-Marganiec and M. Ryan, Eds., IOS Press, pp. 12–30.

[52] Fiege, L., Mezini, M., Mühl, G., and Buchmann, A. P. Engineering Event-Based Systems with Scopes. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)* (June 2002), Springer-Verlag, pp. 309–333. DOI: 10.1007/3-540-47993-7_14.

[53] Fiege, L., Zeidler, A., Buchmann, A., Kilian-Kehr, R., and Mühl, G. Security Aspects in Publish/Subscribe Systems. In *Proceedings of the 3rd International Workshop on Distributed Event-Based Systems (DEBS'04)* (May 2004), IET, pp. 44–49. DOI: 10.1049/ic:20040381.

[54] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm` (Last accessed: December 3, 2009).

[55] Foster, I., Vockler, J., Wilde, M., and Zhao, Y. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management* (2002), pp. 37–46. DOI: 10.1109/SSDM.2002.1029704.

[56] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

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

[58] Gansner, E. R., and North, S. C. An Open Graph Visualization System and its Applications to Software Engineering. *Software: Practice and Experience 30*, 11 (2000), 1203–1233. (`http://www.graphviz.org/`) DOI: 10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.3.CO;2-E.

[59] Garg, P. K., Eshghi, K., Gschwind, T., Haverkort, B. R., and Wolter, K. Enabling Network Caching of Dynamic Web Objects. In *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS'02)* (2002), Springer Verlag, pp. 329–338. DOI: 10.1007/3-540-46029-2_24.

[60] Garlan, D., Khersonsky, S., and Kim, J. S. Model Checking Publish-Subscribe Systems. In *Proceedings of the 10th International SPIN Workshop on Model Checking of Software (SPIN 2003)* (May 2003), pp. 166–180.

[61] Globus Alliance. *Globus Toolkit v4.0.8*, August 2008. `http://www.globus.org/toolkit/` (Last accessed: September 23, 2008).

[62] Groth, P., Jiang, S., Miles, S., Munroe, S., Tan, V., Tsasakou, S., and Moreau, L. An Architecture for Provenance Systems. Tech. Rep. 12023, Univ. of Southampton, 2006.

[63] Halfond, W. G. J., and Orso, A. AMNESIA: Analysis and Monitoring for NEutralizing SQL-Injection Attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)* (Nov. 2005), ACM, pp. 174–183. DOI: 10.1145/1101908.1101935.

[64] Harney, J., and Doshi, P. Selective Querying for Adapting Web Service Compositions Using the Value of Changed Information. *IEEE Transactions on Services Computing 1*, 3 (2008), 169–185. DOI: 10.1109/TSC.2008.11.

[65] Hayes, B. Cloud Computing. *Communications of the ACM 51*, 7 (2008), 9–11. DOI: 10.1145/1364782.1364786.

[66] Heinis, T., and Alonso, G. Efficient Lineage Tracking for Scientific Workflows. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (2008), ACM, pp. 1007–1018. DOI: 10.1145/1376616.1376716.

[67] Housley, R., Ford, W., Polk, W., and Solo, D. RFC2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile, January 1999. `http://tools.ietf.org/html/rfc2459` (Last accessed: December 9, 2009).

[68] Hu, S., Muthusamy, V., Li, G., and Jacobsen, H.-A. Distributed Automatic Service Composition in Large-Scale Systems. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)* (2008), ACM, pp. 233–244. DOI: 10.1145/1385989.1386019.

[69] Huang, Y., Slominski, A., Herath, C., and Gannon, D. WS-Messenger: A Web Services-Based Messaging System for Service-Oriented Grid Computing. In *Proceedings of the 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'06)* (2006), IEEE Computer Society, pp. 166–173. DOI: 10.1109/CCGRID.2006.109.

[70] Huber, A. VMF - A Transformation Engine for Resolving Web Service Heterogeneities within the VRESCo Runtime. Master's thesis, Vienna University of Technology, July 2009.

[71] IBM Corporation. *Web Service Level Agreements (WSLA)*, January 2003. `http://www.research.ibm.com/wsla/` (Last accesssed: November 11, 2009).

[72] IBM Corporation. *Common Base Event (CBE)*, August 2004. `http://www.ibm.com/developerworks/library/specification/ws-cbe/` (Last accesssed: October 20, 2009).

[73] IBM Corporation. *Common Event Infrastructure (CEI)*, August 2004. `http://www-01.ibm.com/software/tivoli/features/cei/` (Last accesssed: October 20, 2009).

[74] IBM Corporation. *Web Services Metadata Exchange (WS-MetadataExchange)*, August 2006. `http://www.ibm.com/developerworks/webservices/library/specification/ws-mex/` (Last accessed: October 4, 2009).

[75] IBM Corporation. *WebSphere Application Server v7*, September 2008. `http://www.ibm.com/software/webservers/appserv/was/` (Last accessed: October 4, 2009).

[76] IBM Corporation. *WebSphere MQ*, September 2008. `http://www.ibm.com/software/integration/wmq/` (Last accessed: October 20, 2009).

[77] IBM, Inc. *WebSphere Service Registry and Repository, v6.2*, July 2008. `http://www.ibm.com/software/integration/wsrr` (Last accessed: Novermber 30, 2009).

[78] Jaganathan, R. Windows Workflow Foundation: Tracking Services Deep Dive, January 2007. `http://msdn.microsoft.com/en-us/library/bb264458%28VS.80%29.aspx` (Last accessed: September 2, 2009).

[79] Jobst, D., and Preissler, G. Mapping Clouds of SOA- and Business-Related Events for an Enterprise Cockpit in a Java-Based Environment. In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java (PPPJ'06)* (2006), ACM, pp. 230–236. DOI: 10.1145/1168054.1168089.

[80] Juszczyk, L., Troung, H.-L., and Dustdar, S. GENESIS – A Framework for Automatic Generation and Steering of Testbeds of Complex Web Services. In *Proceedings of the 13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)* (Mar. 2008), pp. 131–140. DOI: 10.1109/ICECCS.2008.27.

[81] Kaminski, P., Müller, H., and Litoiu, M. A Design for Adaptive Web Service Evolution. In *Proceedings of the International workshop on Self-Adaptation and Self-Managing Systems (SEAMS'06)* (2006), ACM Press, pp. 86–92. DOI: 10.1145/1137677.1137694.

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

[83] Kephart, J. O., and Chess, D. M. The Vision of Autonomic Computing. *Computer 36*, 1 (2003), 41–50. DOI: 10.1109/MC.2003.1160055.

[84] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)* (June 2001), pp. 327–354. DOI: 10.1007/3-540-45337-7_18.

[85] Kiss, R. *WS-Eventing for WCF (Indigo)*, June 2007. `http://www.codeproject.com/KB/WCF/WSEventing.aspx` (Last accessed: October 22, 2009).

[86] Kozlenkov, A., Spanoudakis, G., Zisman, A., Fasoulas, V., and Sanchez, F. Architecture-driven Service Discovery for Service Centric Systems. *International Journal of Web Service Research 4*, 2 (2007), 82–113.

[87] Laner, T. VQL Ű A View-based Querying Approach for the VRESCo Runtime. Master's thesis, Vienna University of Technology, July 2009.

[88] Leach, P., Mealling, M., and Salz, R. RFC4122: A Universally Unique Identifier (UUID) URN Namespace, July 2005. `http://tools.ietf.org/html/rfc4122` (Last accessed: December 9, 2009).

[89] Leitner, P., Michlmayr, A., and Dustdar, S. Towards Flexible Interface Mediation for Dynamic Service Invocations. In *Proceedings of the 3rd Workshop on Emerging Web Services Technology (WEWST'08)* (Nov. 2008).

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

[91] Leitner, P., Michlmayr, A., Rosenberg, F., and Dustdar, S. Selecting Web Services Based on Past User Experiences. In *Proceedings of the IEEE Proceedings of the IEEE Asia-Pacific Services Computing Conference (APSCC'09)* (December 2009), IEEE Computer Society.

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

[93] Leitner, P., Rosenberg, F., Michlmayr, A., Huber, A., and Dustdar, S. A Mediator-Based Approach to Resolving Interface Heterogeneity of Web Services. In *Emerging Web Service Technologies, Volume III*, W. Binder and S. Dustdar, Eds. Birkhäuser, July 2009, pp. 55–74. DOI: 10.1007/978-3-0346-0104-7_4.

[94] Leitner, P., Wetzstein, B., Rosenberg, F., Michlmayr, A., Dustdar, S., and Leymann, F. Runtime Prediction of Service Level Agreement Violations for Composite Services. In *Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing (NFPSLAM-SOC'09)* (November 2009).

[95] Li, G., Cheung, A., Hou, S., Hu, S., Muthusamy, V., Sherafat, R., Wun, A., Jacobsen, H.-A., and Manovski, S. Historic Data Access in Publish/Subscribe. In *Proceedings of the Inaugural International Conference on Distributed Event-Based Systems (DEBS'07)* (2007), ACM, pp. 80–84. DOI: 10.1145/1266894.1266908.

[96] Liberty, J., and Xie, D. *Programming C# 3.0*. O'Reilly Media, Inc., 2007.

[97] Liebig, C., Malva, M., and Buchmann, A. P. Integrating Notifications and Transactions: Concepts and X^2TS Prototype. In *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO'00)* (2001), Springer-Verlag, pp. 194–214. DOI: 10.1007/3-540-45254-0_18.

[98] Liebig, C., and Tai, S. Middleware Mediated Transactions. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)* (2001), IEEE Computer Society, p. 340. DOI: 10.1109/DOA.2001.954099.

[99] Lin, B., Gu, N., and Li, Q. A Requester-based Mediation Framework for Dynamic Invocation of Web Services. In *Proceedings of the IEEE International Conference on Services Computing (SCC'06)* (2006), IEEE Computer Society, pp. 445–454. DOI: 10.1109/SCC.2006.13.

[100] Lodi, G., Panzieri, F., Rossi, D., and Turrini, E. SLA-Driven Clustering of QoS-Aware Application Servers. *IEEE Transactions on Software Engineering 33*, 3 (2007), 186–197. DOI: 10.1109/TSE.2007.28.

[101] Löwy, J. *Programming WCF Services*. O'Reilly Media, Inc., 2007.

[102] Luckham, D. *The Power of Events*. Addison-Wesley, 2002.

[103] Luckham, D. C., and Vera, J. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering 21*, 9 (1995), 717–734. DOI: 10.1109/32.464548.

[104] Mahambre, S. P., S.D, M. K., and Bellur, U. A Taxonomy of QoS-Aware, Adaptive Event-Dissemination Middleware. *IEEE Internet Computing 11*, 4 (2007), 35–44. DOI: 10.1109/MIC.2007.77.

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

[106] Medjahed, B. Dissemination Protocols for Event-Based Service-Oriented Architectures. *IEEE Transactions on Services Computing 1*, 3 (2008), 155–168. DOI: 10.1109/TSC.2008.13.

[107] Menascé, D. A. QoS Issues in Web Services. *IEEE Internet Computing 6*, 6 (2002), 72–75. DOI: 10.1109/MIC.2002.1067740.

[108] Mendling, J., and Hafner, M. From WS-CDL Choreography to BPEL Process Orchestration. *Journal of Enterprise Information Management 22* (2008), 525–542. DOI: 10.1108/17410390810904274.

[109] Michlmayr, A., and Fenkam, P. Integrating Distributed Object Transactions with Wide-Area Content-Based Publish/Subscribe Systems. In *Proceedings of the 4th International Workshop on Distributed Event-Based Systems (DEBS'05)* (2005), IEEE Computer Society, pp. 398–403. DOI: 10.1109/ICDCSW.2005.80.

[110] Michlmayr, A., Fenkam, P., and Dustdar, S. Architecting a Testing Framework for Publish/Subscribe Applications. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)* (September 2006), IEEE Computer Society, pp. 467–474. DOI: 10.1109/COMPSAC.2006.28.

[111] Michlmayr, A., Fenkam, P., and Dustdar, S. Specification-Based Unit Testing of Publish/Subscribe Applications. In *Proceedings of the 5th International Workshop on Distributed Event-based Systems (DEBS 2006)* (July 2006), IEEE Computer Society. DOI: 10.1109/ICDCSW.2006.103.

[112] Michlmayr, A., Leitner, P., Rosenberg, F., and Dustdar, S. Publish/Subscribe in the VRESCo SOA runtime (Demo paper). In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)* (July 2008), ACM, pp. 317–320. DOI: 10.1145/1385989.1386031.

[113] Michlmayr, A., Leitner, P., Rosenberg, F., and Dustdar, S. Event Processing in Web Service Runtime Environments. In *Principles and Applications of Distributed Event-based Systems*, A. Hinze and A. Buchmann, Eds. IGI Global, 2010. (forthcoming).

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

[115] Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. Comprehensive QoS Monitoring of Web Services and Event-Based SLA Violation Detection. In *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing (MW4SOC'09)* (Nov. 2009), ACM, pp. 1–6. DOI: 10.1145/1657755.1657756.

[116] Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. Service Provenance in QoS-Aware Web Service Runtimes. In *Proceedings of the 7th International Conference on Web Services (ICWS'09)* (July 2009), IEEE Computer Society, pp. 115–122. DOI: 10.1109/ICWS.2009.32.

[117] Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. *IEEE Transactions on Services Computing (TSC)* (2010). (forthcoming).

[118] Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. Selective Service Provenance in the VRESCo Runtime. *International Journal on Web Services Research (JWSR)* (2010). (forthcoming).

[119] Michlmayr, A., Rosenberg, F., Platzer, C., Treiber, M., and Dustdar, S. Towards Recovering the Broken SOA Trianlge – A Software Engineering Perspective. In *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering (IW-SOSWE'07)* (Sept. 2007), ACM, pp. 22–28. DOI: 10.1145/1294928.1294934.

[120] Microsoft Cooperation. *Toward Converging Web Service Standards for Resources, Events, and Management (WS-EventNotification)*, March 2006. `http://msdn.microsoft.com/en-us/library/aa480724.aspx` (Last accessed: December 10, 2009).

[121] Moreau, L., Groth, P., Miles, S., Vazquez-Salceda, J., Ibbotson, J., Jiang, S., Munroe, S., Rana, O., Schreiber, A., Tan, V., and Varga, L. The Provenance of Electronic Data. *Communications of the ACM 51*, 4 (2008), 52–58. DOI: 10.1145/1330311.1330323.

[122] Moreau, L., Plale, B., Miles, S., Goble, C., Missier, P., Barga, R., Simmhan, Y., Futrelle, J., McGrath, R. E., Myers, J., Paulson, P., Bowers, S., Ludaescher, B., Kwasnikowska, N., den Bussche, J. V., Ellkvist, T., Freire, J., and Groth, P. The Open Provenance Model (v1.01), July 2008. `http://www.openprovenance.org/` (Last accessed: October 12, 2009).

[123] Moser, O., Rosenberg, F., and Dustdar, S. Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proceedings of the 17th International Conference on World Wide Web (WWW'08)* (2008), ACM, pp. 815–824. DOI: 10.1145/1367497.1367607.

[124] Mühl, G., Fiege, L., and Pietzuch, P. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[125] MuleSoft, Inc. *Mule Galaxy, v1.5.1*, November 2009. `http://www.mulesoft.org/display/GALAXY/Home` (Last accessed: Novermber 30, 2009).

[126] Neuman, C., Yu, T., Hartman, S., and Raeburn, K. RFC4120: The Kerberos Network Authentication Service (V5), July 2005. `http://tools.ietf.org/html/rfc4120` (Last accessed: December 9, 2009).

[127] Object Management Group (OMG). *CORBA Event Service, v1.2*, October 2004. `http://www.omg.org/cgi-bin/doc?formal/2004-10-02` (Last accesssed: May 28, 2009).

[128] Object Management Group (OMG). *CORBA Notification Service, v1.1*, October 2004. `http://www.omg.org/cgi-bin/doc?formal/2004-10-11` (Last accesssed: May 28, 2009).

[129] Object Management Group (OMG). *Data Distribution Service, v1.2*, January 2007. `http://www.omg.org/spec/DDS/1.2/` (Last accesssed: May 28, 2009).

[130] Open SOA. *Service Component Architecture (SCA)*, March 2007. `http://osoa.org/display/Main/Service+Component+Architecture+Home` (Last accessed: July 3, 2009).

[131] Open SOA. *SCA Assembly Extensions for Event Processing and Pub/Sub*, April 2009. `http://osoa.org/download/attachments/35/SCA_Assembly_Extensions_for_Event_Processing_and_PubSub_V1_0.pdf?version=1` (Last accessed: July 3, 2009).

[132] Oracle, Inc. *Oracle Fusion Middleware*, 2009. `http://www.oracle.com/us/products/middleware/` (Last accessed: Novermber 30, 2009).

[133] Organization for the Advancement of Structured Information Standards (OASIS). *WS-Reliable Messaging v1.1*, Nov. 2004. `http://www.oasis-open.org/committees/wsrm/` (Last accessed: August 19, 2009).

[134] Organization for the Advancement of Structured Information Standards (OASIS). *ebXML Registry Services and Protocols*, May 2005. `http://www.oasis-open.org/committees/regrep` (Last accessed: December 3, 2009).

[135] Organization for the Advancement of Structured Information Standards (OASIS). *Security Assertion Markup Langugage v2.0*, Mar. 2005. `http://www.oasis-open.org/committees/security/` (Last accessed: August 19, 2009).

[136] Organization for the Advancement of Structured Information Standards (OASIS). *Universal Description, Discovery and Integration (UDDI)*, February 2005. `http://www.oasis-open.org/committees/uddi-spec/` (Last accessed: December 3, 2009).

[137] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Distributed Management (WSDM)*, February 2006. `http://oasis-open.org/committees/wsdm/` (Last accessed: October 3, 2009).

[138] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Notification (WS-Notification)*, October 2006. `http://oasis-open.org/committees/wsn/` (Last accessed: December 3, 2009).

[139] Organization for the Advancement of Structured Information Standards (OASIS). *WS-Security v1.1*, Feb. 2006. `http://www.oasis-open.org/committees/wss` (Last accessed: June 22, 2009).

[140] Organization for the Advancement of Structured Information Standards (OASIS). *Web Service Business Process Execution Language (WS-BPEL) v2.0*, April 2007. `http://www.oasis-open.org/committees/wsbpel/` (Last accessed: June 22, 2009).

[141] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Transactions (WS-TX) v1.2*, February 2009. `http://www.oasis-open.org/committees/ws-tx/` (Last accessed: October 22, 2009).

[142] Ostrowski, K., and Birman, K. Extensible Web Services Architecture for Notification in Large-Scale Systems. In *Proceedings of the 4th International Conference on Web Services (ICWS'06)* (2006), IEEE Computer Society, pp. 383–392. DOI: 10.1109/ICWS.2006.63.

[143] Pallickara, S., and Fox, G. NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware (Middleware'03)* (2003), Springer-Verlag New York, Inc., pp. 41–61.

[144] Papazoglou, M. P., Traverso, P., Dustdar, S., and Leymann, F. Service-Oriented Computing: State of the Art and Research Challenges. *IEEE Computer 40*, 11 (2007), 38–45. DOI: 10.1109/MC.2007.400.

[145] Pautasso, C., and Alonso, G. Flexible Binding for Reusable Composition of Web Services. In *Proceedings of the 4th International Workshop on Software Composition (SC'2005), Edinburgh, UK* (2005), Springer, pp. 151–166. DOI: 10.1007/11550679_12.

[146] Peiris, C., Mulder, D., Bahree, A., Chopra, A., Cicoria, S., and Pathak, N. *Pro WCF: Practical Microsoft SOA Implementation*. Apress, Berkely, CA, USA, 2007.

[147] Pietzuch, P. R. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambridge, Feb. 2004. `http://www.doc.ic.ac.uk/%7Epeter/manager/doc/thesis/prp_thesis.pdf` (Last accessed: May 27, 2009).

[148] Pietzuch, P. R., and Bacon, J. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)* (July 2002), IEEE Computer Society, pp. 611–618. DOI: 10.1109/ICDCSW.2002.1030837.

[149] Platzer, C., and Dustdar, S. A Vector Space Search Engine for Web Services. In *Proceedings of the 3rd European IEEE Conference on Web Services (ECOWS'05)* (November 2005). DOI: 10.1109/ECOWS.2005.5.

[150] Platzer, C., Rosenberg, F., and Dustdar, S. Web Service Clustering using Multidimensional Angles as Proximity Measures. *ACM Transactions on Internet Technology 9*, 3 (2009), 1–26. DOI: 10.1145/1552291.1552294.

[151] Ponnekanti, S. R., and Fox, A. Interoperability Among Independently Evolving Web Services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware (Middleware'04)* (2004), Springer-Verlag New York, pp. 331–351. DOI: 10.1007/b101561.

[152] Porter, G., and Katz, R. H. Effective Web Service Load Balancing through Statistical Monitoring. *Communications of the ACM 49*, 3 (2006), 48–54. DOI: 10.1145/1118178.1118201.

[153] Raimondi, F., Skene, J., and Emmerich, W. Efficient Online Monitoring of Web-Service SLAs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16)* (2008), pp. 170–180. DOI: 10.1145/1453101.1453125.

[154] Rajbhandari, S., and Walker, D. W. Incorporating Provenance in Service Oriented Architecture. In *Proceedings of the International Conference on Next Generation Web Services Practices (NWeSP'06)* (2006), IEEE Computer Society, pp. 33–40. DOI: 10.1109/NWESP.2006.18.

[155] Ran, S. A Model for Web Services Discovery with QoS. *SIGecom Exchanges 4*, 1 (2003), 1–10. DOI: 10.1145/844357.844360.

[156] Red Hat Middleware, LLC. *JBossWS*, August 2009. `http://www.jboss.org/jbossws/` (Last accessed: October 13, 2009).

[157] Red Hat Middleware, LLC. *NHibernate Reference Documentation*, 2009. `http://www.nhibernate.org/` (Last accessed: October 13, 2009).

[158] Rosenberg, F. *QoS-Aware Composition of Adaptive Service-Oriented Systems*. PhD thesis, Vienna University of Technology, June 2009.

[159] Rosenberg, F., Celikovic, P., Michlmayr, A., Leitner, P., and Dustdar, S. An End-to-End Approach for QoS-Aware Service Composition. In *Proceedings of the 13th IEEE International Enterprise Computing Conference (EDOC'09), Auckland, New Zealand* (September 2009), IEEE Computer Society, pp. 151–160. DOI: 10.1109/EDOC.2009.14.

[160] Rosenberg, F., Leitner, P., Michlmayr, A., Celikovic, P., and Dustdar, S. Towards Composition as a Service - A Quality of Service Driven Approach. In *Proceedings of the 1st IEEE Workshop on Information and Software as Service (WISS'09)* (March 2009), IEEE Computer Society, pp. 1733–1740. DOI: 10.1109/ICDE.2009.153.

[161] Rosenberg, F., Leitner, P., Michlmayr, A., and Dustdar, S. Integrated Metadata Support for Web Service Runtimes. In *Proceedings of the Middleware for Web Services Workshop (MWS'08)* (2008), IEEE Computer Society, pp. 361–368. DOI: 10.1109/EDOCW.2008.38.

[162] Rosenberg, F., Michlmayr, A., and Dustdar, S. Top-Down Business Process Development and Execution using Quality of Service Aspects. *Enterprise Information Systems 2*, 4 (Nov. 2008), 459–475. DOI: 10.1080/17517570802395626.

[163] Rosenberg, F., Platzer, C., and Dustdar, S. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the 4th International Conference on Web Services (ICWS'06)* (Sept. 2006), IEEE Computer Society, pp. 205–212. DOI: 10.1109/ICWS.2006.39.

[164] Rozsnyai, S., Vecera, R., Schiefer, J., and Schatten, A. Event Cloud - Searching for Correlated Business Events. In *Proceedings of the 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)* (2007), IEEE Computer Society, pp. 409–420. DOI: 10.1109/CEC-EEE.2007.47.

[165] RSS Advisory Board. *Really Simple Syndication (RSS)*, 2009. `http://www.rssboard.org/rss-specification` (Last accessed: October 31, 2009).

[166] Sayre, R. Atom: The Standard in Syndication. *IEEE Internet Computing 9*, 4 (2005), 71–78. DOI: 10.1109/MIC.2005.74.

[167] Schmidt, M.-T., Hutchison, B., Lambros, P., and Phippen, R. The Enterprise Service Bus: Making Service-Oriented Architecture Real. *IBM Systems Journal 44*, 4 (2005), 781–797.

[168] Segall, B., Arnold, D., Boot, J., Henderson, M., and Phelps, T. Content Based Routing with Elvin4. In *Proceedings of the AUUG2K Conference* (june 2000). `http://www.elvin.org/papers/auug2k/auug2k.pdf` (Last accesssed: October 4, 2009).

[169] Shatsky, Y., and Gudes, E. TOPS: A New Design for Transactions in Publish/Subscribe Middleware. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)* (2008), ACM, pp. 201–210. DOI: 10.1145/1385989.1386015.

[170] Shilo, O. *CS-Script – The C# Script Engine*. `http://www.csscript.net/` (Last accessed: November 19, 2009).

[171] Shukla, D., and Schmidt, B. *Essential Windows Workflow Foundation (Microsoft .Net Development Series)*. Addison-Wesley Professional, 2006.

[172] Simmhan, Y. L., Plale, B., and Gannon, D. A Survey of Data Provenance in e-Science. *SIGMOD Record 34*, 3 (2005), 31–36. DOI: 10.1145/1084805.1084812.

[173] Simmhan, Y. L., Plale, B., and Gannon, D. Karma2: Provenance Management for Data-Driven Workflows. *International Journal of Web Services Research 5*, 2 (2008), 1–22.

[174] Sivashanmugam, K., Verma, K., and Sheth, A. Discovery of Web Services in a Federated Registry Environment. In *Proceedings of the 2nd International Conference on Web Services (ICWS'04)* (2004), IEEE Computer Society, pp. 270–278. DOI: 10.1109/ICWS.2004.48.

[175] Skene, J., Lamanna, D. D., and Emmerich, W. Precise Service Level Agreements. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)* (2004), pp. 179–188. DOI: 10.1109/ICSE.2004.1317440.

[176] Spanoudakis, G., Zisman, A., and Kozlenkov, A. A Service Discovery Framework for Service Centric Systems. In *Proceedings of the IEEE International Conference on Services Computing (SCC'05)* (2005), IEEE Computer Society, pp. 251–259. DOI: 10.1109/SCC.2005.17.

[177] Sun Microsystems, Inc. *Java Messaging Service (JMS) v1.1*, March 2002. `http://java.sun.com/products/jms/` (Last accesssed: October 4, 2009).

[178] Sun Microsystems, Inc. *Jini, v2.0.2*, June 2005. `http://java.sun.com/products/jini/2_0_2index.html` (Last accesssed: March 31, 2009).

[179] Sun Microsystems, Inc. *Wiseman v1.0*, June 2007. `http://wiseman.dev.java.net/` (Last accessed: October 4, 2009).

[180] Tai, S., Mikalsen, T. A., Rouvellou, I., and Sutton, Jr, S. M. Dependency-Spheres: A Global Transaction Context for Distributed Objects and Messages. In *Proceedings of the 5th International Enterprise Distributed Object Computing Conference (EDOC'01)* (2001), IEEE Computer Society, pp. 105–117. DOI: 10.1109/EDOC.2001.950427.

[181] Tan, V., Groth, P. T., Miles, S., Jiang, S., Munroe, S., Tsasakou, S., and Moreau, L. Security Issues in a SOA-Based Provenance System. In *Provenance and Annotation of Data – Post-Proceedings of the International Provenance and Annotation Workshop (IPAW'06)*, L. Moreau and I. Foster, Eds. Springer, 2006, pp. 203–211. DOI: 10.1007/11890850_21.

[182] Thio, N., and Karunasekera, S. Automatic Measurement of a QoS Metric for Web Service Recommendation. In *Proceedings of the Australian Software Engineering Conference (ASWEC'05)* (2005), pp. 202–211. DOI: 10.1109/ASWEC.2005.16.

[183] TIBCO Sofware, Inc. *TIBCO Rendezvous*, 2008. `http://www.tibco.com/software/messaging/rendezvous/` (Last accessed: December 3rd, 2009).

[184] Treiber, M., and Dustdar, S. Active Web Service Registries. *IEEE Internet Computing 11*, 5 (2007), 66–71. DOI: 10.1109/MIC.2007.99.

[185] Tsai, W.-T., Wei, X., Zhang, D., Paul, R., Chen, Y., and Chung, J.-Y. A New SOA Data-Provenance Framework. In *Proceedings of the 8th International Symposium on Autonomous Decentralized Systems (ISADS'07)* (2007), IEEE Computer Society, pp. 105–112. DOI: 10.1109/ISADS.2007.5.

[186] Vargas, L., Pesonen, L. I. W., Gudes, E., and Bacon, J. Transactions in Content-Based Publish/Subscribe Middleware. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops (ICDCSW '07)* (2007), IEEE Computer Society, p. 68. DOI: 10.1109/ICDCSW.2007.85.

[187] Vinoski, S. More Web Services Notifications. *IEEE Internet Computing 8*, 3 (2004), 90–93. DOI: 10.1109/MIC.2004.1297279.

[188] Vinoski, S. The More Things Change . . . *IEEE Internet Computing 8*, 1 (2004), 87–89. DOI: 10.1109/MIC.2004.1260709.

[189] Vogels, W. Web Services Are Not Distributed Objects. *IEEE Internet Computing 7*, 6 (2003), 59–66. DOI: 10.1109/MIC.2003.1250585.

[190] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., and Ferguson, D. F. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall PTR, 2005.

[191] Wetzstein, B., Leitner, P., Rosenberg, F., Brandic, I., Dustdar, S., and Leymann, F. Monitoring and Analyzing Influential Factors of Business Process Performance. In *Proceedings of the 13th IEEE International Enterprise Computing Conference (EDOC'09), Auckland, New Zealand* (2009), IEEE Computer Society, pp. 141–150. DOI: 10.1109/EDOC.2009.18.

[192] World Wide Web Consortium (W3C). *XML Path Language*, November 1999. `http://www.w3.org/TR/xpath` (Last accessed: October 17, 2009).

[193] World Wide Web Consortium (W3C). *OWL-S: Semantic Markup for Web Services*, November 2004. `http://www.w3.org/Submission/OWL-S/` (Last accessed: November 18, 2009).

[194] World Wide Web Consortium (W3C). *Web Service Addressing*, August 2004. `http://www.w3.org/Submission/ws-addressing/` (Last accessed: October 17, 2009).

[195] World Wide Web Consortium (W3C). *Web Service Execution Environment (WSMX)*, June 2005. `http://www.w3.org/Submission/WSMX/` (Last accesssed: November 18, 2009).

[196] World Wide Web Consortium (W3C). *Web Service Modeling Ontology (WSMO)*, June 2005. `http://www.w3.org/Submission/WSMO/` (Last accesssed: November 18, 2009).

[197] World Wide Web Consortium (W3C). *Web Services Choreography Description Language (WS-CDL)*, November 2005. `http://www.w3.org/TR/ws-cdl-10/` (Last accesssed: October 3, 2009).

[198] World Wide Web Consortium (W3C). *Web Service Policy 1.2 - Framework (WS-Policy)*, April 2006. `http://www.w3.org/Submission/WS-Policy/` (Last accessed: October 3, 2009).

[199] World Wide Web Consortium (W3C). *Web Services Eventing (WS-Eventing)*, March 2006. `http://www.w3.org/Submission/WS-Eventing/` (Last accessed: December 3, 2009).

[200] World Wide Web Consortium (W3C). *Extending and Versioning Languages*, November 2007. `http://www.w3.org/2001/tag/doc/versioning` (Last accessed: October 3, 2009).

[201] World Wide Web Consortium (W3C). *Semantic Annotations for WSDL and XML Schema*, Aug. 2007. `http://www.w3.org/TR/sawsdl/` (Last accesssed: November 18, 2009).

[202] World Wide Web Consortium (W3C). *Simple Object Access Protocol (SOAP) v1.2*, Arpil 2007. `http://www.w3.org/TR/soap12/` (Last accessed: November 18, 2009).

[203] World Wide Web Consortium (W3C). *Web Service Description Language (WSDL) v2.0*, June 2007. `http://www.w3.org/TR/wsdl20/` (Last accessed: November 18, 2009).

[204] WCF Performance Counters, 2009. `http://msdn.microsoft.com/en-us/library/ms735098.aspx` (Last accessed: December 3, 2009).

[205] WSO2, Inc. *WSO2 Registry, v2.0*, Feb. 2009. `http://wso2.org/projects/registry` (Last accessed: Novermber 30, 2009).

[206] Wun, A., and Jacobsen, H.-A. A Policy Management Framework for Content-Based Publish/Subscribe Middleware. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware (Middleware'07)* (2007), Springer-Verlag, pp. 368–388. DOI: 10.1007/978-3-540-76778-7_19.

[207] XMethods, 2009. `http://www.xmethods.com/` (Last accessed: October 3, 2009).

[208] Yu, Q., and Bouguettaya, A. Framework for Web Service Query Algebra and Optimization. *ACM Transactions on the Web 2*, 1 (2008), 1–35. DOI: 10.1145/1326561.1326567.

[209] Yu, Q., Liu, X., Bouguettaya, A., and Medjahed, B. Deploying and Managing Web services: Issues, Solutions, and Directions. *The VLDB Journal 17*, 3 (2008), 537–572. DOI: 10.1007/s00778-006-0020-3.

[210] Yu, T., Zhang, Y., and Lin, K.-J. Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. *ACM Transactions on the Web 1*, 6 (2007), 6. DOI: 10.1145/1232722.1232728.

[211] Zdun, U., Hentrich, C., and Aalst, W. M. P. V. D. A Survey of Patterns for Service-Oriented Architectures. *International Journal of Internet Protocol Technology 1*, 3 (2006), 132–143. DOI: 10.1504/IJIPT.2006.009739.

[212] Zeng, L., Benatallah, B., Ngu, A. H., Dumas, M., Kalagnanam, J., and Chang, H. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering 30*, 5 (May 2004), 311–327. DOI: 10.1109/TSE.2004.1.

[213] Zhang, H., Bradbury, J. S., Cordy, J. R., and Dingel, J. Implementation and Verification of Implicit-Invocation Systems Using Source Transformation. In *Proceedings of the 5th International IEEE Workshop on Source Code Analysis and Manipulation (SCAM 2005)* (Sept. 2005), IEEE Computer Society. DOI: 10.1109/SCAM.2005.15.

Abbreviations

ACL	Access Control Layer
AOP	Aspect-Oriented Programming
API	Application Programming Interface
BPEL	Business Process Execution Language
CBE	Common Base Event
CBR	Content-Based Routing
CEI	Common Event Infrastructure
CEP	Complex Event Processing
CPO	Cell Phone Operator
CRM	Customer Relationship Management
DAIOS	Dynamic and Asynchronous Invocation of Services
DAL	Data Access Layer
DAO	Data Access Object
DSL	Domain-Specific Language
ebXML	Electronic Business using XML
ESB	Enterprise Service Bus
EPL	Event Processing Language
EV	Event Visibility
HQL	Hibernate Query Language
HTTP(S)	Hypertext Transfer Protocol (Secure)
ILC	Instance-Level Claim
JMS	Java Message Service
KPI	Key Performance Indicator
OCL	Object Constraint Language
ORM	Object Relational Mapping
OWL	Web Ontology Language
P2P	Peer-to-Peer
PubSub (P/S)	Publish/Subscribe
QoE	Quality of Experience
QoS	Quality of Service

RBAC	Role-Based Access Control
REST	Representational State Transfer
RLC	Resource-Level Claim
RPC	Remote Procedure Call
RSS	Really Simple Syndication
SaaS	Software as a Service
SAWSDL	Semantic Annotations for WSDL
SCA	Service Component Architecture
SCM	Software Configuration Management
SLA	Service Level Agreement
SMS	Short Messaging Service
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SOC	Service-Oriented Computing
SQL	Structured Query Language
TCP	Transmission Control Protocol
UDDI	Universal Description, Discovery and Integration
UUID	Universally Unique Identifier
UML	Unified Modeling Language
VRESCo	Vienna Runtime Environment for Service-Oriented Computing
VCL	Vienna Composition Language
VMF	VRESCo Mapping Framework
VQL	VRESCo Query Language
WCF	Windows Communication Foundation
WF	Windows Workflow Foundation
WPC	Windows Performance Counters
WSDL	Web Service Description Language
WSDM	Web Service Distributed Management
WSLA	Web Service Level Agreement
WSMO	Web Service Modeling Ontology
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language (XSL) Transformations

Appendix A.

VQL/SQL Query Examples

```
var query = new VQuery(typeof(ServiceRevision));
query.Add(Expression.Eq("IsActive", true));
query.Add(Expression.Eq("Operations.Feature.Name", "NotifyCustomer"));
query.Match(Expression.Eq("QoS.Property.Name", "ResponseTime") &
            Expression.Lt("QoS.DoubleValue", 900), 3);
query.Match(Expression.Eq("QoS.Property.Name", "Availability") &
            Expression.Gt("QoS.DoubleValue", 0.99), 1);
```

Listing A.1: VQL Query

```
SELECT t0.ID as ID
FROM Revision t0
LEFT OUTER JOIN ServiceRevisionRELOperation t3 ON(t0.ID = t3.ServiceRevisionID)
LEFT OUTER JOIN VrescoFunction t1 ON(t3.OperationID = t1.ID)
LEFT OUTER JOIN VrescoFunction t2 ON(t1.FeatureID = t2.ID)
WHERE t0.IsActive = 'true' AND t2.Name = 'NotifyCustomer'
AND t0.ID IN
        (SELECT DISTINCT t4.ID as ID
        FROM Revision t4
        LEFT OUTER JOIN RevisionQoS t5 ON(t4.ID = t5.RevisionID)
        LEFT OUTER JOIN Property t6 ON(t5.PropertyID = t6.ID)
        WHERE (t6.Name = 'ResponseTime' AND t5.DoubleValue < 900))
AND t0.ID IN
        (SELECT DISTINCT t7.ID as ID
        FROM Revision t7
        LEFT OUTER JOIN RevisionQoS t8 ON(t7.ID = t8.RevisionID)
        LEFT OUTER JOIN Property t9 ON(t8.PropertyID = t9.ID)
        WHERE (t9.Name = 'Availability' AND t8.DoubleValue > 0.99))
GROUP BY t0.ID
LIMIT 100
```

Listing A.2: Translated SQL Query (Exact Strategy - L100)

```
SELECT t0.ID as ID , SUM(COALESCE(t4.P,0) + COALESCE(t8.P,0)) as Priority
FROM Revision t0
LEFT OUTER JOIN
        (SELECT DISTINCT t1.ID as ID , 3 as P
        FROM Revision t1
        LEFT OUTER JOIN RevisionQoS t2 ON(t1.ID = t2.RevisionID)
        LEFT OUTER JOIN Property t3 ON(t2.PropertyID = t3.ID)
        WHERE (t3.Name = 'ResponseTime' AND t2.DoubleValue < 900)) t4 ON(t4.ID = t0.ID)
LEFT OUTER JOIN
        (SELECT DISTINCT t5.ID as ID , 1 as P
        FROM Revision t5
        LEFT OUTER JOIN RevisionQoS t6 ON(t5.ID = t6.RevisionID)
        LEFT OUTER JOIN Property t7 ON(t6.PropertyID = t7.ID)
        WHERE (t7.Name = 'Availability' AND t6.DoubleValue > 0.99)) t8 ON(t8.ID = t0.ID)
LEFT OUTER JOIN ServiceRevisionRELOperation t11 ON(t0.ID = t11.ServiceRevisionID)
LEFT OUTER JOIN VrescoFunction t9 ON(t11.OperationID = t9.ID)
LEFT OUTER JOIN VrescoFunction t10 ON(t9.FeatureID = t10.ID)
WHERE t0.IsActive = 'true' AND t10.Name = 'NotifyCustomer'
GROUP BY t0.ID , t4.P , t8.P
ORDER BY Priority DESC
LIMIT 100
```

Listing A.3: Translated SQL Query (Priority Strategy - L100)

```
SELECT TOP 100 t0.ID as ID , SUM(COALESCE(t4.P,0) + COALESCE(t8.P,0)) as Priority
FROM Revision t0
LEFT OUTER JOIN
        (SELECT DISTINCT t1.ID as ID , 1 as P
        FROM Revision t1
        LEFT OUTER JOIN RevisionQoS t2 ON(t1.ID = t2.RevisionID)
        LEFT OUTER JOIN Property t3 ON(t2.PropertyID = t3.ID)
        WHERE (t3.Name = 'ResponseTime' AND t2.DoubleValue < 900)) t4 ON(t4.ID = t0.ID)
LEFT OUTER JOIN
        (SELECT DISTINCT t5.ID as ID , 1 as P
        FROM Revision t5
        LEFT OUTER JOIN RevisionQoS t6 ON(t5.ID = t6.RevisionID)
        LEFT OUTER JOIN Property t7 ON(t6.PropertyID = t7.ID)
        WHERE (t7.Name = 'Availability' AND t6.DoubleValue > 0.99)) t8 ON(t8.ID = t0.ID)
LEFT OUTER JOIN ServiceRevisionRELOperation t11 ON(t0.ID = t11.ServiceRevisionID)
LEFT OUTER JOIN VrescoFunction t9 ON(t11.OperationID = t9.ID)
LEFT OUTER JOIN VrescoFunction t10 ON(t9.FeatureID = t10.ID)
GROUP BY t0.ID , t4.P , t8.P
ORDER BY Priority DESC
LIMIT 100
```

Listing A.4: Translated SQL Query (Relaxed Strategy - L100)

Appendix B.

Subscription Message Examples

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:not="http://www.vitalab.tuwien.ac.at/vresco/notifications"
                  xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
                  xmlns:a="http://www.w3.org/2005/08/addressing">
  <soapenv:Header>
    <not:SubscriptionQuery>
      select * from QoSRevisionEvent where Revision.Id = 4711
      and Property = 'ResponseTime' and DoubleValue &gt; 500
    </not:SubscriptionQuery>
    <not:Subscriber>admin</not:Subscriber>
  </soapenv:Header>
  <soapenv:Body>
    <even:Subscribe>
      <even:Subscribe xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing">
        <wse:EndTo>
          <a:Address>http://vresco.vitalab.tuwien.ac.at:20005/SubscriptionManagerService</a:Address>
        </wse:EndTo>
        <wse:Delivery wse:Mode="http://schemas.xmlsoap.org/ws/2004/08/eventing/DeliveryModes/Push">
          <wse:NotifyTo>
            <a:Address>net.tcp://vresco.vitalab.tuwien.ac.at:33333/OnVRESCoEvents</a:Address>
          </wse:NotifyTo></wse:Delivery>
        <wse:Expires>P1D</wse:Expires>
      </even:Subscribe>
    </even:Subscribe>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing B.1: Subscription Request Message

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/"
            xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing">
            xmlns:a="http://www.w3.org/2005/08/addressing">
   <s:Body xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <SubscribeResponse xmlns="http://schemas.xmlsoap.org/ws/2004/08/eventing">
         <wse:SubscribeResponse xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing">
            <wse:SubscriptionManager>
               <a:Address>http://vresco.vitalab.tuwien.ac.at:20005/SubscriptionManagerService</a:Address>
               <a:ReferenceParameters>
                  <wse:Identifier>uuid:4ec58981-3718-4641-b25c-b6c7bbe37b4a</wse:Identifier>
               </a:ReferenceParameters>
            </wse:SubscriptionManager>
            <wse:Expires>2010-02-01T15:21:46.606Z</wse:Expires>
         </wse:SubscribeResponse>
      </SubscribeResponse>
   </s:Body>
</s:Envelope>
```

Listing B.2: Subscription Response Message

Appendix C.

Notification Message Example

```
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope"
            xmlns:a="http://www.w3.org/2005/08/addressing">
  <s:Header>
    <a:Action s:mustUnderstand="1">http://www.vitalab.tuwien.ac.at/vresco/notifications/notify</a:Action>
    <a:To s:mustUnderstand="1">net.tcp://vresco.vitalab.tuwien.ac.at:33333/OnVRESCoEvents</a:To>
  </s:Header>
  <s:Body>
    <Notify xmlns="http://www.vitalab.tuwien.ac.at/vresco/">
      <newEvents xmlns:b="http://www.vitalab.tuwien.ac.at/vresco/usertypes"
                 xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
        <b:VRESCoEvent i:type="b:QoSRevisionEvent">
          <b:Publisher>admin</b:Publisher>
          <b:PublisherGroup>AdminGroup</b:PublisherGroup>
          <b:SeqNum>59303452-8304-485b-b8bb-dc3e9ae645d3:55</b:SeqNum>
          <b:Timestamp>2010-02-01T10:41:37</b:Timestamp>
          <b:Visibility>ALL</b:Visibility>
          <b:DoubleValue>692</b:DoubleValue>
          <b:Property>ResponseTime</b:Property>
          <b:RevisionId>4711</b:RevisionId>
        </b:VRESCoEvent>
      </newEvents>
      <oldEvents i:nil="true" xmlns:b="http://www.vitalab.tuwien.ac.at/vresco/usertypes"
                              xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
      </oldEvents>
      <subscriptionId>uuid:9214fb42-2094-4563-8255-d12ea8f48369</subscriptionId>
    </Notify>
  </s:Body>
</s:Envelope>
```

Listing C.1: Event Notification Message

Appendix D.

Curriculum Vitae

Personal Information

Current Position:	University Assistant (Faculty Member)
Address (Work):	Distributed Systems Group (DSG)
	Vienna University of Technology
	Argentinierstrasse 8/184-1, 1040 Vienna, Austria
Voice:	++43 1 58801 18452
Fax:	++43 1 58801 18491
E-Mail:	anton@infosys.tuwien.ac.at
WWW:	http://www.infosys.tuwien.ac.at/staff/michlmayr/
Date of Birth:	23.12.1979, Linz (Austria)
Citizenship:	Austrian

Education

PhD Studies in Computer Science **June 2006 - March 2010**
Vienna University of Technology
Thesis: "Event Processing in QoS-Aware Service Runtime Environments"
Advisor: Prof. Schahram Dustdar (TU Wien)
Examiner: Prof. Carlo Ghezzi (Politecnico di Milano)

MSc Studies in Computer Science **October 1999 - April 2005**
Vienna University of Technology
Thesis: "Integrating Transactions with Content-based Publish/Subscribe Middleware"
Advisors: Prof. Mehdi Jazayeri, Dr. Pascal Fenkam
Graduated with honors

Secondary School **September 1990 - June 1998**
Bundesrealgymnasium Steyr Michaelerplatz
Graduated with honors

Languages

German (native)
English (good)
French (basic)

Experience

University Assistant **June 2006 - now**
Vienna University of Technology (DSG)

Research Assistant **August 2005 - May 2006**
Vienna University of Technology (DSG)
Project *RAY* (FWF P16970-N04) funded by the Austrian Research Foundation FWF

Teaching Assistant **March 2002 - June 2004**
Vienna University of Technology
Tutor at the Database and Artifical Intelligence Group for the courses:
- ❏ Data Modeling (181.114 - VU 2.0 - Datenmodellierung)
- ❏ Database Systems (181.379 - LU 2.0 - Datenbanksysteme)

Civilian Service **October 1998 - September 1999**
Lebenshilfe OÖ: Supporting mentally disabled people

Furthermore, I have done summer internships and project work at BMW Motoren Steyr GmbH, Heinrich Moser GmbH and Softlab Austria (Cirquent).

Teaching Activities

Bachelor Courses
- ❏ Scientific Writing (184.121 - SE 2.0 - Grundlagen methodischen Arbeitens)
- ❏ Distributed Systems Lab (184.167 - LU 2.0 - Verteilte Systeme)
- ❏ Project Lab Work (e.g., 184.230 - PR 4.0 - Projektpraktikum)

Master Courses
- ❏ Technologies for Distributed Systems (184.260 - VU 4.0 - Technologien für Verteilte Systeme)
- ❏ Software Architectures (184.159 - VU 2.0 - Software Architekturen)
- ❏ Internet Computing Lab Work (e.g., 184.254 - PR 4.0 - Praktikum aus Internet Computing)

Master Thesis Supervised
- ❏ Thomas Laner: A Semantically Enriched Querying Language for the VRESCo Metamodel
- ❏ Andreas Huber: A Transformation Engine for Resolving Web Service Heterogeneities within the VRESCo Runtime

Awards

Web Service Challenge: 3rd place (http://www.ws-challenge.org/) **July 2007**
Together with Lukasz Juszczyk, Florian Rosenberg, 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), Tokyo, Japan.

Master Thesis Grant **July 2004 - June 2005**
Vienna University of Technology (DSG)

Merit Grant for outstanding achievements **November 2003**
Vienna University of Technology

Research Interests

Software Architectures for Distributed Systems
Service-oriented Computing
Distributed Event-based Systems

Program Committee Memberships

- ❑ 3rd International Workshop on Dynamic and Declarative Business Processes (DDBP'10), Vitoria, ES, Brazil (co-located with EDOC'10)
- ❑ 8th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS'10), Funchal, Portugal (co-located with ICEIS'10)
- ❑ 7th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS'09), Milano, Italy (co-located with ICEIS'09)
- ❑ 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'08), Phuket, Thailand.
- ❑ 6th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems (MSVVEIS'08), Barcelona, Spain (co-located with ICEIS'08).
- ❑ IADIS International Conference on Applied Computing 2008 (AC'08), Algarve, Portugal.
- ❑ 8th ACIS International Conference on Software Engineering, Artifical Intelligence, Networking, and Parallel/Distributed Computing (SNPD'07), Qingdao, China.
- ❑ IADIS International Conference on Applied Computing 2007 (AC'07), Salamanca, Spain.

Publications

Journal Papers

1. Anton Michlmayr, Florian Rosenberg, Philipp Leitner, Schahram Dustdar: “End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo”, IEEE Transactions on Services Computing (TSC), 2010. (forthcoming)
2. Anton Michlmayr, Florian Rosenberg, Philipp Leitner, Schahram Dustdar: “Selective Service Provenance in the VRESCo Runtime”, International Journal on Web Services Research (JWSR), IGI Global, 2010. (forthcoming)
3. Florian Rosenberg, Anton Michlmayr, Schahram Dustdar, “Top-Down Business Process Development and Execution using Quality of Service Aspects”, Enterprise Information Systems, 2(4), 459-475, Taylor & Francis, November 2008.

Conference, Workshop and Demo Papers

4. Philipp Leitner, Anton Michlmayr, Florian Rosenberg, Schahram Dustdar: "Selecting Web Services Based on Past User Experiences", Proceedings of the IEEE Asia-Pacific Services Computing Conference (APSCC’09), Biopolis, Singapore, December 2009.
5. Anton Michlmayr, Florian Rosenberg, Philipp Leitner, Schahram Dustdar: "Comprehensive QoS Monitoring of Web Services and Event-Based SLA Violation Detection", Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing (MW4SOC’09 @ Middleware’09), Urbana-Champaign, IL, USA, December 2009.
6. Philipp Leitner, Branimir Wetzstein, Florian Rosenberg, Anton Michlmayr, Schahram Dustdar, Frank Leymann: "Runtime Prediction of Service Level Agreement Violations for Composite Services", Proceedings of the 3rd Workshop on Non-Functional Properties and SLA Management in Service-Oriented Computing (NFPSLAM-SOC’09 @ ICSOC/ServiceWave’09), Stockholm, Sweden, November 2009.
7. Florian Rosenberg, Predrag Celikovic, Anton Michlmayr, Philipp Leitner, Schahram Dustdar: "An End-to-End Approach for QoS-Aware Service Composition", Proceedings of the 13th IEEE International Enterprise Computing Conference (EDOC’09), Auckland, New Zealand, September 2009.
8. Anton Michlmayr, Florian Rosenberg, Philipp Leitner, Schahram Dustdar: "Service Provenance in QoS-Aware Web Service Runtimes". Proceedings of the 7th IEEE International Conference on Web Services (ICWS’09), Los Angeles, CA, USA, July 2009.

9. Florian Rosenberg, Philipp Leitner, Anton Michlmayr, Predrag Celikovic, Schahram Dustdar: "Towards Composition as a Service - A Quality of Service Driven Approach", Proceedings of the 1st IEEE Workshop on Information and Software as Service (WISS'09 @ ICDE'09), Shanghai, China, March 2009.
10. Philipp Leitner, Anton Michlmayr, Schahram Dustdar: "Towards Flexible Interface Mediation for Dynamic Service Invocations", Proceedings of the 3rd Workshop on Emerging Web Services Technology (WEWST'08 @ ECOWS'08), Dublin, Ireland, November 2008.
11. Florian Rosenberg, Philipp Leitner, Anton Michlmayr, Schahram Dustdar: "Integrated Metadata Support for Web Service Runtimes", Proceedings of the Middleware for Web Services Workshop (MWS'08 @ EDOC'08), Munich, Germany, September 2008.
12. Philipp Leitner, Anton Michlmayr, Florian Rosenberg, Schahram Dustdar: "End-to-End Versioning Support for Web Services", Proceedings of the International Conference on Services Computing (SCC'08), Honolulu, HI, USA, July 2008.
13. Anton Michlmayr, Florian Rosenberg, Philipp Leitner, Schahram Dustdar: "Advanced Event Processing and Notifications in Service Runtime Environments". Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08), Rome, Italy, July 2008.
14. Anton Michlmayr, Philipp Leitner, Florian Rosenberg, Schahram Dustdar: "Publish/Subscribe in the VRESCo SOA Runtime" (demo paper). Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08), Rome, Italy, July 2008.
15. Florian Rosenberg, Christian Enzi, Anton Michlmayr, Christian Platzer, Schahram Dustdar: "Integrating QoS Aspects in Top-Down Business Process Development using WS-CDL and WS-BPEL", Proceedings of the 11th IEEE International EDOC Conference (EDOC'07), Annapolis, MD, USA, October 2007.
16. Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, Schahram Dustdar: "Towards Recovering the Broken SOA Triangle - A Software Engineering Perspective", Proceedings of the 2nd International Workshop on Service-oriented Software Engineering (IW-SOSWE'07 @ ESEC/FSE'07), Dubrovnik, Croatia, September 2007.
17. 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", Proceedings of the IEEE Joint Conference on E-Commerce Technology and Enterprise Computing, E-Commerce and E-Services, (CEC/EEE'07), Tokyo, Japan, July 2007.
18. Anton Michlmayr, Pascal Fenkam, and Schahram Dustdar: “Architecting a Testing Framework for Publish/Subscribe Applications”, Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06), Chicago, IL, USA, September 2006.

19. Anton Michlmayr, Pascal Fenkam, Schahram Dustdar: “Specification-Based Unit Testing of Publish/Subscribe Applications”, Proceedings of the 5th International Workshop on Distributed Event-Based Systems (DEBS’06 @ ICDCS’06), Lisbon, Portugal, July 2006.
20. Anton Michlmayr and Pascal Fenkam: “Integrating Distributed Object Transactions with Wide-Area Content-Based Publish/Subscribe Systems”, Proceedings of the 4th International Workshop on Distributed Event-Based Systems (DEBS’05 @ ICDCS’05), Columbus, OH, USA, June 2005.

Book Chapters

21. Anton Michlmayr, Philipp Leitner, Florian Rosenberg, Schahram Dustdar: “Event Processing in Web Service Runtime Environments”: Principles and Applications of Distributed Event-based Systems, IGI Global (Editors: Annika Hinze, Alejandro Buchmann), 2010, (forthcoming).
22. Philipp Leitner, Florian Rosenberg, Anton Michlmayr, Andreas Huber, Schahram Dustdar: "A Mediator-Based Approach to Resolving Interface Heterogeneity of Web Services", Post-Proceedings of the 3rd International Workshop on Emerging Web Service Technologies (WEWST’08), Birkhaeuser (Editors: Walter Binder, Schahram Dustdar), July 2009.
23. Florian Rosenberg, Anton Michlmayr, Christoph Nagl, Schahram Dustdar: "Distributed Business Rules within Service-Centric Systems", Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, IGI Global (Editors: Dragan Gasevic, Adrian Giurca, Kuldar Taveter), May 2009.

Poster Presentations

24. Anton Michlmayr, Florian Rosenberg, Philipp Leitner, Schahram Dustdar: "VRESCo – Vienna Runtime Environment for Service-Oriented Computing", 10th International Middleware Conference (Middleware’09), Urbana-Champaign, IL, USA, December 2009.

Theses

25. Anton Michlmayr, "Event Processing in QoS-Aware Service Runtime Environments", PhD Thesis, Vienna University of Technology, March 2010.
26. Anton Michlmayr, "Integrating Transactions with Content-based Publish/Subscribe Middleware", Master Thesis, Vienna University of Technology, April 2005.