Vienna University of Technology
Information Systems Institute
Distributed Systems Group

End-to-End Support for QoS-Aware Service Selection, Invocation and Mediation in VRESCo

Under Review for Publication in IEEE Transactions on Services Computing

Anton Michlmayr, Florian Rosenberg,
Philipp Leitner and Schahram Dustdar
lastname@infosys.tuwien.ac.at

TUV-1841-2009-03 May 26, 2009

Service-oriented Computing has recently received a lot of attention from both academia and industry. However, current service-oriented solutions are often not as dynamic and adaptable as intended because the publish-find-bind-execute cycle of the SOA triangle is not entirely realized. In this paper, we highlight some issues of current Web service technologies, with a special emphasis on service metadata, querying, invocation and mediation. We present the Vienna Runtime Environment for Service-oriented Computing (VRESCo) that aims at solving a number of these issues. Among others, VRESCO *provides support for service metadata and querying, monitoring of Quality of Service, dynamic binding and invocation including service mediation, service notifications, and service compositions. Additionally, we give an evaluation that proves the performance and usefulness of our system.*

Keywords: Service-oriented Computing, Service Runtimes, Web Services

Argentinierstr. 8/184-1
A-1040 Vienna, Austria
phone: +43 1 58801-18402
fax: +43 1 58801-18491
URL: http://www.infosys.tuwien.ac.at/

End-to-End Support for QoS-Aware Service Selection, Invocation and Mediation in VRESCo

Anton Michlmayr, *Member, IEEE,* Florian Rosenberg, *Member, IEEE,*
Philipp Leitner, *Member, IEEE,* and Schahram Dustdar, *Member, IEEE*

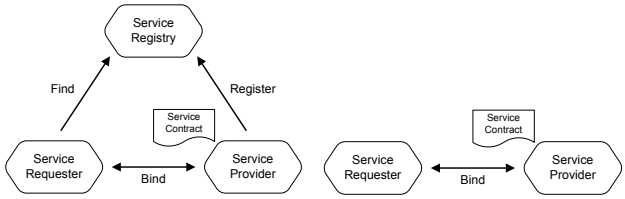
Abstract—Service-oriented Computing has recently received a lot of attention from both academia and industry. However, current service-oriented solutions are often not as dynamic and adaptable as intended because the publish-find-bind-execute cycle of the SOA triangle is not entirely realized. In this paper, we highlight some issues of current Web service technologies, with a special emphasis on service metadata, querying, invocation and mediation. We present the Vienna Runtime Environment for Service-oriented Computing (VRESCo) that aims at solving a number of these issues. Among others, VRESCo provides support for service metadata and querying, monitoring of Quality of Service, dynamic binding and invocation including service mediation, service notifications, and service compositions. Additionally, we give an evaluation that proves the performance and usefulness of our system.

1 Introduction

During the last few years, Service-oriented Architecture (SOA) and Service-oriented Computing (SOC) [1] has gained acceptance as a paradigm for mastering 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 1a. *Service providers* implement services and make them 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 the platform-independent service descriptions, one can implement flexible applications with respect to manageability and adaptivity. For instance, services can easily be exchanged at runtime, and service consumers can switch to alternative services seamlessly. This increases the organizational agility [2], allows companies to soften the disruptive effects of changes in the business or IT environment, and quickly react to unexpected events (such as new competitors entering the market). Web services [3] represent the most common realization of SOA, building on the main standards SOAP [4] for messaging-based communication, WSDL [5] for service interface descriptions, and UDDI [6] for service registries.

All authors are with the Distributed Systems Group, Vienna University of Technology, Vienna, Austria.
E-mail: {anton,florian,leitner,dustdar}@infosys.tuwien.ac.at

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 [7] did not succeed as intended, which is partly because of their limited querying support that only provides keyword-based matching of registry content and does not consider the 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 2005. As a result, service registries are often missing in service-centric systems, leading to point-to-point solutions where service endpoints are exchanged at design-time (e.g., using E-Mail or phone) and service consumers statically bind to these endpoints (see Figure 1b).



(a) SOA Model (b) SOA Practice

Fig. 1: Basic SOA Model – Theory vs. Practice

Besides that, support for dynamic binding and invocation of services is often restricted to services having the same technical interface. In this regard, the lack of service metadata makes it difficult for service consumers to know if two services actually perform the same task. Furthermore, support for Quality of Service (QoS) is necessary to enable service selection based on functional and non-functional quality attributes. Finally, event processing can be used to get notified about changing QoS attributes which may trigger dynamic re-binding to services with better QoS.

The contribution of this paper is threefold: Firstly, we discuss the issues we see in current SOC research and practice by describing the problems that arise when building SOC applications with current tools and frameworks. Secondly, we introduce the VRESCo service runtime that aims at solving some of these issues. We de-

scribe the details of VRESCo with a special emphasis on service metadata, querying, invocation, and mediation. Finally, we provide an extensive performance evaluation that shows the applicability of our approach.

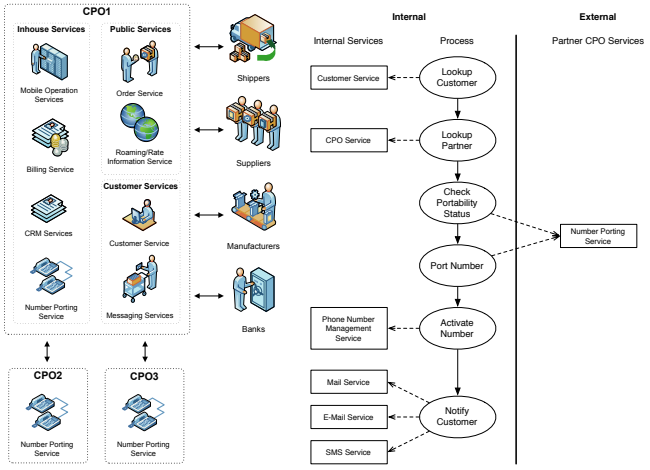
The remainder of this paper is organized as follows: Section 2 presents an illustrative example and summarizes some issues we see in SOC research and practice. Section 3 then introduces related approaches while Section 4 describes the details of the VRESCo runtime environment. Section 5 gives a thorough evaluation of our work, and Section 6 finally concludes the paper.

2 Motivation and Problem Statement

This section first introduces a motivating example which is used throughout the paper. Then, we derive the problems developers face when engineering service-centric systems with current tools and frameworks.

2.1 Motivating Example

Figure 2 shows a typical enterprise application scenario from the telecommunications domain that is used throughout the paper to highlight current problems and our proposed solution. The overview of this case study is depicted in Figure 2a.



(a) Case Study Overview (b) Number Porting Process

Fig. 2: CPO Case Study

In this figure, cell phone operator CPO1 provides different kinds of services: Firstly, *public services* (e.g., rate information service) can be used by everyone. Secondly, *customer services* (e.g., short messaging service) are used by customers of CPO1. Thirdly, *inhouse services* (e.g., CRM services) 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 and CPO3). As discussed later, these scenario bears several challenges that are typical in service-centric software engineering.

Figure 2b shows a simplified version of the number porting process. 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. After the customer has been looked up in the customer service, the number porting service of the old CPO has to be invoked. If the porting was successful, the new number is activated by the mobile operations service. Finally, a notification is sent to the customer using the preferred notification mechanism (e.g., SMS, E-Mail, etc.).

2.2 SOC Challenges and Contributions

Adaptive service-oriented systems bring along several distinct requirements, leading to a number of challenges that have to be addressed. In this section, we summarize the current challenges we see most important. These challenges also represent the core contributions of the VRESCo approach.

- *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 [8] 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 pre-defined 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 [6] and ebXML [7]. 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 [9]. Besides service discovery, another issue is how to select a service from a pool of possible service candidates [10]. 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 (QoS).* In enterprise scenarios QoS plays a crucial role [11]. This includes functional attributes such as response time, availability or throughput, and non-functional attributes such as cost or security. The QoS model should ideally be extensible to allow service providers to adapt

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 1: Related Enterprise Registry Approaches

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 Service Level Agreements (SLA).

- *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 attributes). However, in practice this is currently 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 service might differ, but the number porting process of CPO1 should still be able to seamlessly switch between them at runtime.
- *Service Versioning.* Like any piece of software, services are subject to permanent change regarding their interfaces and implementation. Current registry standards provide limited support for versioning of registry data but cannot handle the differences between various service revisions, for instance as shown in [12]. 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).
- *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., service is created) and complex events regarding QoS (e.g., average response time of service X has changed) and invocations (e.g., service X has been invoked), supporting complex event processing [13]. Users can subscribe to various events of interest, and get notified per E-Mail or Web service notifications (e.g., WS-Eventing [14]). Such notifications may trigger adaptive behavior (e.g., rebinding to other services).

3 Related Work

In this section, we review related work and state of the art concerning service repositories and service metadata, as well as service selection, invocation, and mediation.

Currently, several approaches and standards for service registries/repositories exist. We have compared some existing solutions with the VRESCo runtime, trying to cover the full range of established standards, mature open-source frameworks and commercial tools. We consider the standards UDDI [6] and ebXML [7] (with special emphasis on the registry part), the Mule ESB and Galaxy [15] as service repository, the WSO2 ESB and registry [16], and the closed-source IBM WebSphere [17] solution (including ESB, service registry and repository).

Our findings are shown in Table 1, and are structured according to the challenges introduced in Section 2. Generally, 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 WebSphere provides an extensive structured metadata model (e.g., supporting OWL). 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 [18] 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 attributes is not widely available – it is to some extent possible in WSO2 and WebSphere, and fully supported by VRESCo. WSO2 supports only QoS in terms of WS-Security and WS-ReliableMessaging. However, none of these frameworks except VRESCo integrates QoS monitoring facilities. Integration of dynamic binding, invocation and mediation of services is for obvious reasons 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. All systems except UDDI and VRESCo allow to maintain multiple versions of metadata in the registry. However, only VRESCo has what we consider an end-to-end support for service versioning that allows to easily switch between service versions at runtime. Finally,

all approaches provide support for basic notifications (e.g., if services are published) using E-Mail, WS notifications or Atom. Only WebSphere and VRESCo allow clients to subscribe to more complex events and event patterns using a rich subscription language.

Besides UDDI and ebXML, there are other standards for describing service metadata [8]. Some of them are used by semantic Web service approaches [19] (such as OWL-S [20], WSML [21] and SAWSDL [22]). It should be noted, however, that the VRESCo service metadata model introduced in Section 4.2 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 [23]. In these approaches, service querying and selection play a crucial role, especially regarding service composition (e.g., [11], [24], [25]). However, the query models of current registries and Web service search engines [26] mainly focus on keyword-based matching of service properties which often do not cover the rich semantics of service metadata.

Yu and Bouguettaya [27] 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, and also includes a QoS model. 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, we address dynamic binding and service mediation since service interfaces of different service providers are not always identical in practice.

Dynamic binding of services has been addressed by other approaches (e.g., [28], [29]). Pautasso and Alonso [28] discuss various binding models for services, together with different points in time when bindings are evaluated. They present a flexible binding model in the JOpera system where binding is done using reflection and does not require a specific language construct. Di Penta et. al. [29] present the WS-Binder framework for enabling dynamic binding within WS-BPEL processes. Their approach uses proxies to separate abstract services from concrete services instances. Both approaches have in common that they rather focus on dynamic binding with respect to composition environments whereas VRESCo addresses binding at the core SOA level.

4 System Description

This section describes in detail the VRESCo runtime which was first sketched in [9]. Besides an architectural overview, this includes service metadata and querying, as well as dynamic binding and invocation mechanisms together with our service mediation approach.

4.1 Overview

The architectural overview of VRESCo is shown in Figure 3. The VRESCo core services are provided as Web services that can be accessed either directly using SOAP or by using the client library that provides a simple API. Furthermore, the DAIOS framework [30] has been integrated into the client library, and provides dynamic and asynchronous invocations of Web services. The access control layer guarantees that only authorized clients can access the core services, which is handled using claim-based access control and certificates [31]. Services and associated metadata [32] are stored in the service registry which is accessed using an Object-Relational Mapping (ORM) layer. Finally, the QoS monitor [33] is responsible for regularly measuring the QoS values of services. The overall runtime environment is implemented in C# using the Windows Communication Foundation [34]. Due to the platform-independent architecture, the client library is currently provided for C# and Java.

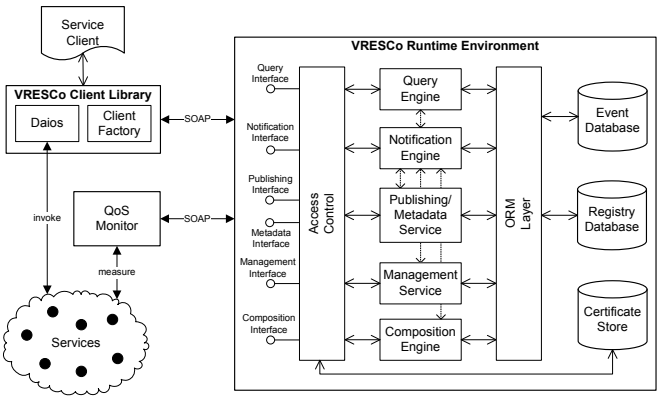


Fig. 3: VRESCo Overview Architecture

There are several VRESCo core services. The Publishing/Metadata Service is used to publish services and metadata into the registry database. Furthermore, the Management Service is responsible for managing user information (e.g., name, password, etc.) whereas the Query Engine is used to query all information stored in the database. The task of the Notification Engine [35] is to inform users when certain events of interest occur inside the runtime, while the Composition Engine [36] finally provides mechanisms to compose services by considering QoS attributes. In this paper, we focus on the main requirements for our client-side mediation approach which are the Metadata Service (including the models used for metadata, services and QoS), the Query Engine, as well as dynamic binding and invocation mechanisms.

4.2 Service Metadata and Mapping

The VRESCo runtime provides a rich service metadata model capable of storing additional information about services in the registry. This is needed to capture the purpose of services to enable querying and mediating between services that perform the same task.

4.2.1 Service Metadata Model

The VRESCo metadata model introduced in [32] is depicted in Figure 4. 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 one domain that perform the same task (e.g., `Check_Status` and `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 either return `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 a given phone number).

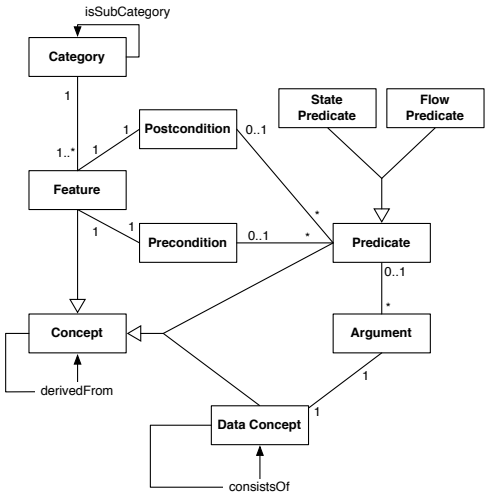


Fig. 4: Service Metadata Model

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` from our CPO case study could have the *flow predicate* `requires(Customer)` as *pre-* and `produces(PortabilityStatus)` as *postcondition*.
- *State predicates* express some global behavior that is 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`.

4.2.2 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 5 basically follows the Web service notation as introduced by WSDL with extensions to enable service versioning [37], represent QoS and enable eventing on a service runtime level.

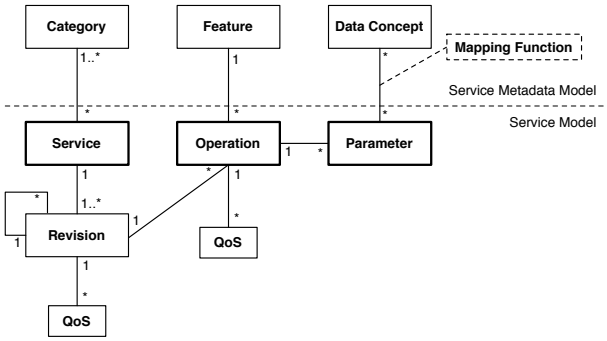


Fig. 5: Service Model to Metadata Model Mapping

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 that represent a complete versioning graph of a concrete service (for details see [37]). Both, revisions and operations can have a number of QoS attributes (*QoS*) representing all service-level attributes as described below. 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 (if a service is not available, all operations are generally also unavailable). In addition, services, revisions, operations and metadata can have a number of associated events (not shown in Figure 5 for brevity). These events are raised whenever an action is performed, e.g., invoking a service, publishing a new service or creating a new category [35].

4.2.3 Mapping Metadata to Concrete Services

In order to associate metadata to concrete services in the service model we have to establish a mapping between metadata and services. The mapping is shown in Figure 5, 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. Currently we assume a 1:1 mapping between features and operations; every feature is implemented in exactly one service operation, and every operation implements exactly one feature of a category. However, we plan to provide support for more complex mappings using the VRESCo composition engine [25] (i.e., features will be represented as compositions of several service operations).

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

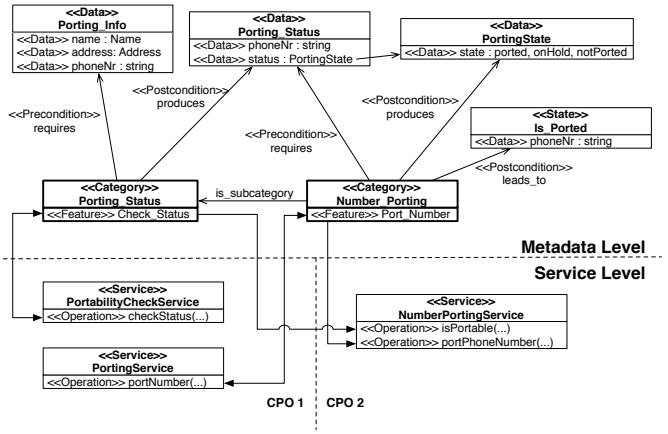


Fig. 6: Mapping Example

Figure 6 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 `Number_Porting` leads to state `Is_Ported`).

The mapping from metadata to service level is now 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 might differ. In that case, various mapping operators (e.g., ==, concat, stringToInt, etc.) can be used to mediate between different service interfaces. Service mediation is discussed in more detail in Section 4.4.2.

4.2.4 QoS Model

Besides the functional attributes described in the service metadata model, a set of QoS attributes is associated with each service revision and operation. These QoS attributes can be either specified manually using the VRESCo Management Service, or measured automatically, e.g., using the QoS monitor introduced in [33]. This monitor has been integrated into VRESCo and follows a client-side approach using aspect-orientation and low-level TCP analysis. As a result, monitoring can be done without access to the actual service implementation. For each service revision, a monitoring schedule can be defined that specifies when the monitor should trigger the measurement. Each individual QoS measurement is thereby published as QoS event into the runtime. The average QoS values can then be aggregated based on the information stored in the QoS events.

Attribute	Formula	Unit
Price	n/a	per invocation
Reliable Messaging	n/a	{true, false}
Security	n/a	{None, X.509,...}
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	$q_{av}(t_0, t_1, t_d) = 1 - \frac{t_d}{t_1 - t_0}$	percent
Accuracy	$q_{ac}(r_f, r_t) = 1 - \frac{r_f}{r_t}$	percent
Throughput	$q_{tp}(t_0, t_1, r) = 1 - \frac{r}{t_1 - t_0}$	invocations/s

TABLE 2: QoS Attributes

Table 2 is adapted from [36] and briefly summarizes the QoS attributes that are currently considered in VRESCo. For each attribute we list the distinct name, the formula how the attribute is calculated (or "n/a" if it is deterministic, such as price, reliable messaging and security) and the unit. The latency $q_{la}(n)$ represents the time a request needs on the wire. It is calculated as the average value of n individual measuring points q_{la_i}. The response time $q_{rt}(n)$ consists of the latency for request

and response plus the execution time of the service. The availability $q_{av}(t_0, t_1, t_d)$ represents the probability a service is up and running (t_0, t_1 are timestamps, t_d is the total time the service was down). The 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, the throughput $q_{tp}(t_0, t_1, r)$ represents the maximum number of requests a service can process within a certain period of time (denoted as $t_1 - t_0$) where r is the total number of requests during that time. In addition to this pre-defined QoS attributes, users can define additional QoS properties for service revisions or operations.

4.3 Querying Language

The VRESCo Query Language (VQL) provides a means to query all information stored in the registry (i.e., services and service metadata including QoS). In this section, we discuss the requirements for VQL, followed by the VQL architecture and the query specification.

4.3.1 Requirements

The design of VQL was driven by a few core aspects and requirements that are briefly summarized below:

- *View-based querying.* The VRESCo architecture implements the data access via a data access layer (DAL) using dedicated data access objects (DAO). However, these DAOs contain database-specific attributes such as IDs (that map to the primary keys of database records) or versioning information for optimistic locking. Therefore, these DAOs are only used internally and referred to as *core objects*. For transmission over the network these entities are transformed into so-called *user objects* that basically contain the same information but without any database-specific fields (and fields that are not intended for the clients). The querying capabilities must be able to deal with both views (depending on whether the query is issued client- or server-side).
- *Type-safety and Security.* Each VQL query should be type-safe in the sense that the result of a query can be parameterizable with specific data types from the service or metadata model (e.g., `ServiceRevision` or `Feature`). Additionally, all query attributes should be subject to runtime existence checks to rule out parameters that do not match a property in the corresponding core or user object. Finally, queries should be fail-safe against well-know security issues such as SQL injection.
- *Object-oriented interface and expression library.* In order to generate VQL queries at runtime, an object-oriented API for specifying these queries should be available (similar to the Hibernate Criteria API [38]). VQL does not provide a declarative language for specifying a query (such as SQL), which makes is simpler in terms of the implementation because no query parser is necessary. Therefore, a rich library of expressions is required that can be used to formulate the queries in an object-oriented manner.
- *Mandatory and optional criteria.* When querying specific information about a service or certain aspects of the metadata model, it is often desired to differentiate between mandatory and optional expressions in a query. For example, one may issue a query to find all services implementing the feature `Send_SMS` which is active and optionally having the QoS attribute response time set to less than 1500 ms. In order to achieve these requirements, different querying strategies have to be provided.

4.3.2 Architecture

These requirements have been addressed by implementing querying capabilities as part of VRESCo. The basic architecture is depicted in Figure 7.

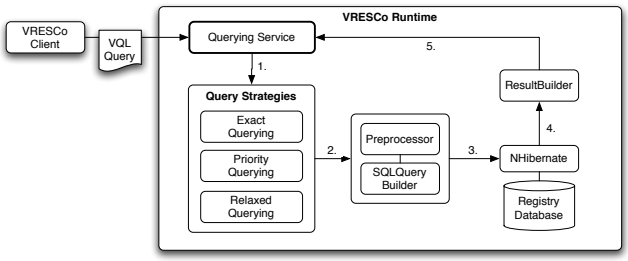


Fig. 7: VRESCo Query Processing Architecture

On the client-side the user specifies a query using an object-oriented interface which is provided by the client library. After specifying the query (using a `VQuery` object) it is sent to the *QueryingService* for execution. Depending on the client's *querying strategy*, VQL selects the corresponding querying strategy based on the strategy design pattern [39] and generates the query accordingly (step 1). VQL leverages the existing and well-established query language SQL as its query execution language, therefore, a `VQuery` instance – representing an in-memory object graph of a query – that was received from the client is preprocessed using the *Preprocessor* component (step 2). This preprocessing includes inspection of the query expressions, the criteria and whether the client queries core or user objects. If the client queries user objects, the query expressions are first transformed to address the properties of the core objects. This is done according to pre-defined mapping annotations that map core and user objects. Additionally, the preprocessing checks if all expressions correspond to attributes in the service- and metadata model. The result of the preprocessing is a generated SQL query that corresponds to the initial `VQuery` (*SQLQueryBuilder* component). When the query is fully generated, a NHibernate session is created to execute the query (step 3). After successful execution, the *ResultBuilder* component takes the result from the NHibernate session (step 4) and transforms it back into the resulting object that was specified as a template parameter in the `VQuery` object (step 5).

4.3.3 Query Specification

In general, VQL queries consist of a set of criteria where each criterion has a number of expressions. Both criteria and expressions are specified using the querying API provided by the VQL library. Therefore, in contrast to other query languages such as SQL, VQL does not provide a declarative querying language which makes it easier to use. Query criteria can either be `Add` and `Match`. These criteria have different execution semantics depending on the querying strategy (discussed below). However, the main motivation is to allow the specification of mandatory (`Add`) and optional (`Match`) criteria. Besides that, VQL provides a set of expressions that can be used to express common query constraints such as comparison (e.g., smaller, greater, equal, etc.) and logical operators (e.g., AND, OR, NOT, etc.). These expressions are summarized in Table 3.

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

TABLE 3: VQL Expressions

Listing 1 shows an example query to find services that implement the `Notify_Customer` feature. In general, VQL queries are parameterized using an expected return type. In this case the type `ServiceRevision` (line 2) expresses that the result of the query is a list of revisions. In our example, two `Add` criteria (lines 5–7) are used to state that a service has to be active and that each service has to implement the `Notify_Customer` feature from the CPO case study. Additionally, three `Match` criteria are added (lines 8–15). The first criterion expresses that a resulting service should be in a category starting with "Porting". The second and third criterion define the optional QoS attributes (response time and availability). All three `Match` criteria use the priority value as third parameter to define the relative importance of a criterion.

The query execution is finally triggered by instantiating an `IVRESCoQuerier` object and invoking the `FindByQuery` method using the specific querying strategy, e.g., `QueryMode.Priority` in our example (lines 18–19). Furthermore, the query can be limited to a given number of results (e.g., 100 in our example).

4.3.4 Querying Strategies

The querying strategy influences how queries are executed, thus, it defines the behavior of the SQL generation. In a nutshell, `Add` criteria are transformed to simple predicates within the SQL `WHERE` clause whereas `Match` are handled as SQL sub-selects.

```
1  // create a query object
2  var query = new VQuery(typeof(ServiceRevision));
3
4  // add query expressions
5  query.Add(Expression.Eq("IsActive", true));
6  query.Add(Expression.Eq("Operations.Feature.Name",
7                          "NotifyCustomer"));
8  query.Match(Expression.Like("Service.Category.Name",
9    "Porting", LikeMatchMode.Start), 5);
10 query.Match(Expression.Eq("QoS.Property.Name",
11                           "ResponseTime") &
12             Expression.Lt("QoS.DoubleValue", 1500), 3);
13 query.Match(Expression.Eq("QoS.Property.Name",
14                           "Availability") &
15             Expression.Gt("QoS.DoubleValue", 0.95), 1);
16
17 // execute the query
18 IVRESCoQuerier querier = VRESCoClientFactory.
       CreateQuerier("username", "password");
19 var results = querier.FindByQuery(query, 100, QueryMode.
       Priority) as IList<ServiceRevision>;
```

Listing 1: VQL Sample Query

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

The *priority querying* strategy involves priority values for single criteria in order to accomplish a weighted matching. Therefore, each `Match` criterion allows to append a weight to specify the priority of this criterion. In Listing 1, the priority values are "5", "3" and "1" (i.e., the constraint on response time is more important than on availability). In contrast, `Add` criteria do not allow to specify a weight because they are mandatory.

The *relaxed querying* strategy represents a special variant of *priority querying*, in the sense that each `Match` criterion has the same priority. Thus, this strategy simply distinguishes between optional and mandatory criteria in this regard. It also allows to define fuzzy queries by relaxing the query constraints which can be useful when no exact match can be found for a given query.

4.4 Dynamic Binding, Invocation and Mediation

One of the motivations for the VRESCo project was to support dynamic binding and invocation, as well as service mediation, which is discussed in this section.

4.4.1 Dynamic Binding and Invocation

Dynamic binding is claimed to be one of the main advantages of service-oriented architecture. In practice, however, services are often bound using pre-generated stubs that do not provide support for dynamic binding. Similar to querying strategies, we use the strategy pattern to implement a number of different rebinding strategies.

Strategy	Proxy reconsiders binding...
`Fixed`	never
`Periodic`	periodically
`OnDemand`	on client requests
`OnInvocation`	prior to service invocations
`OnEvent`	on event notifications

TABLE 4: Rebinding Strategies

All rebinding strategies from Table 4 have their advantages and disadvantages. *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. Finally, *OnEvent* rebinding uses the VRESCo event notification engine to combine the advantages of all strategies by allowing users to precisely define in which situations rebinding should be performed.

The event notification engine introduced in [35] is based on the open source event processing engine Esper [40]. Therefore, subscriptions are defined using the Esper Event Processing Language (EPL) which is similar to SQL and provides various complex event processing mechanisms such as event patterns, sliding event windows and statistical functions on event streams. As a result, in contrast to existing service repository approaches, we provide support for complex event processing. In VRESCo, events are published when certain situations occur (e.g., new service is published, metadata is added, QoS changes, etc.) while notifications are sent per E-Mail or Web service notifications (e.g., WS-Eventing [14]). More details on VRESCo eventing, event access control, and how this can be leveraged to support service provenance can be found in our previous work ([35], [31]). which has been omitted due to space restrictions.

Besides dynamic binding, dynamic invocation of services represents another important goal of service-centric systems. In this regard, we aim at stubless, protocol-independent, and message-driven invocation of services using the DAIOS framework [30]. To give an example, Listing 2 continues Listing 1 and shows a service invocation from our CPO case study. The query from Listing 1 is used to create a proxy using the *periodic* strategy in line 21 (i.e., the proxy reconsiders its binding every minute). In lines 23–26, the input message for

```
20 // continued from Listing 1...
21 var proxy = querier.CreateRebindingMappingProxy(
       query, QueryMode.Exact, 100,
       new PeriodicRebindingStrategy(60000));
22
23 DaiosMessage request = new DaiosMessage();
24 request.SetString("ReceiverNr", "0043-12345678");
25 request.SetString("SenderNr", "0043-98765432");
26 request.SetString("Message", "Number has been ported!");
27
28 DaiosMessage result = proxy.RequestResponse(request);
```

Listing 2: VRESCo Service Invocation

the `Notify_Customer` feature is built, and the corresponding service is finally executed in line 28 using the request-response pattern. In addition, DAIOS also supports asynchronous and one-way communication.

4.4.2 Service Mediation

The VRESCo Mapping Framework (VMF) defines the necessary concepts and mechanisms to handle the mapping from abstract features to concrete service operations from the metadata model as described in Section 4.2. The mediation approach follows the notation of the "feature-driven" metadata model. Therefore, a client that wants to invoke a service in VRESCo does not provide the input of the concrete service directly but already in the conceptual high-level representation, i.e., the feature input in VRESCo terminology. The runtime takes care of lowering and lifting 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).

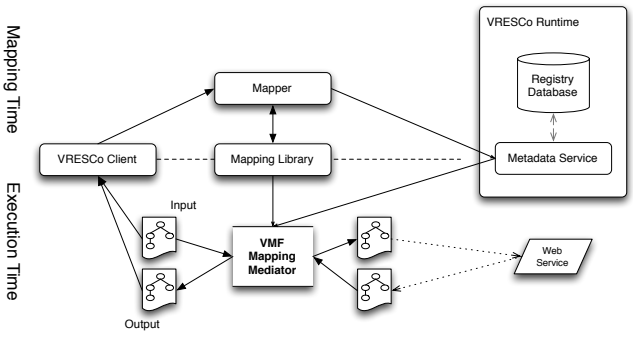


Fig. 8: VMF Architecture

Figure 8 shows an overview of the VMF architecture. Generally, VMF comprises two main components. Firstly, at mapping time, the *Mapper* component is used to create lifting and lowering information (i.e., *Mapping Rules*) for each service. This information is stored in the VRESCo registry database using the Metadata Service. Secondly, at execution time, VMF injects a *Mediator* component, which is responsible for the mediation process itself (the concrete implementation follows the ideas presented in [41]). This mediator retrieves the lifting and lowering information from the VRESCo Metadata Service at runtime, and executes the corresponding mapping.

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

TABLE 5: VMF Mapping Functions

The actual mappings make use of the VMF *Mapping Library*, which includes a number of helpful predefined data manipulation operations. These operations implement some often-used data conversion functionality, such as data type conversion, string manipulation, mathematical functions or logical operators. Furthermore, more complex mappings can be defined in the CS-Script language [42]. We have summarized the provided mapping functions in Table 5.

```
1  // query NotifyCustomer and SendSMS1 instances using VQL
2
3  // create mapper from feature and operation
4  Mapper mapper = metadataService.CreateMapper(
       NotifyCustomer, SendSMS1);
5
6  // map feature message to operation message
7  Assign messageAssign = new Assign(
8    mapper.FeatInParams[0].GetChild("Message"),
9    mapper.OpInParams[0]);
10 mapper.AddMappingFunction(messageAssign);
11
12 // get AreaCode, convert to int and map it to operation
13 Substring acSenderStr = new Substring(
14     mapper.FeatInParams[0].GetChild("SenderNr"), 0, 4);
15 acSenderStr = mapper.AddMappingFunction(acSenderStr);
16 ConvertToInt acSenderInt = new ConvertToInt(
17     acSenderStr.Result);
18 acSenderInt = mapper.AddMappingFunction(acSenderInt);
19 mapper.AddMappingFunction(new Assign(acSenderInt.Result,
       mapper.OpInParams[1]));
20
21 // get SenderNr, convert to int and map it to operation
22 Substring senderNrStr = new Substring(
23     mapper.FeatInParams[0].GetChild("SenderNr"), 4, 8);
24 senderNrStr = mapper.AddMappingFunction(senderNrStr);
25 ConvertToInt senderNrInt = new ConvertToInt(
26     senderNrStr.Result);
27 senderNrInt = mapper.AddMappingFunction(senderNrInt);
28 mapper.AddMappingFunction(new Assign(
29     senderNrInt.Result, mapper.OpInParams[2]));
30
31 // the same steps have to be done for ReceiverNr
```

Listing 3: VMF Mapping Example

Listing 3 illustrates how a concrete mapping (either lifting or lowering) is defined in VMF, using the scenario from the CPO case study process shown in Figure 2b. The feature `Notify_Customer` requires as input the data concepts `Message`, `SenderNr` and `ReceiverNr` (data type *string*). The `SendSMS1` operation of `SMS-Service1` requires the parameter `Message` (data type *string*), but sender and receiver number are splitted into area code and number (data type *integer*). Phone numbers contain an area code with four digits, followed by a number with eight digits. Line 4 shows how the mapper is created for feature `Notify_Customer` and operation `SendSMS1`. Both objects have to be queried using VQL before the mapper can be created (not shown in Listing 3 for brevity). The `Assign` function used in lines 7–10 acts as connector to link the `Message` from the feature to the `Message` of the operation, whereas `mapper.AddMappingFunction()` adds the function to the mapping. Lines 13–19 get the area code from the feature's `SenderNr` as substring and convert it with the `ConvertToInt` function to an integer which is finally assigned to operation's input parameter `AreaCodeSender`. In lines 22– 29 the same is done to map the sender number.

5 Evaluation

In this section, we give an evaluation of the VRESCo runtime focusing on the topics covered in this paper. More precisely, we show the runtime performance regarding service querying, rebinding, mediation, and eventing. All tests 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 SQL Server 2008.

5.1 Querying Performance

First of all, we give the performance results of the querying engine which have been measured by querying for service revisions from a specific service owner that belong to a given category and have a certain response time. All measurements represent the average values of 10 repetitive runs. Table 6 compares the querying strategies provided by VQL. It shows that EXACT querying is faster than RELAXED and PRIORITY which have similar performance characteristics. However, the difference between EXACT and RELAXED/PRIORITY is almost constant. Table 7 shows the comparison between VQL, Hibernate Querying Language (HQL) and Structured Query Language (SQL) using the EXACT strategy. For this experiment, we manually translated the query to both HQL and SQL. The results show that VQL queries are only slightly slower than native SQL queries, whereas VQL and HQL perform equally well.

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

TABLE 6: VQL Querying Strategies (in ms)

The results in the previous tables represent the performance characteristics of single and simple queries. To give a more complex and real-life example, Figure 9

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

TABLE 7: Query Performance (in ms)

illustrates the runtime performance of feature resolution in VRESCo, as used internally by the VRESCo composition engine. Feature resolution is the process of finding all service candidates for a service composition that implement a given feature, and additionally fulfill other constraints such as QoS. The figure shows how long this step takes depending on the number of features in a composition and the number of service candidates per feature. For instance, in a composition of 100 features where each feature has 10 service candidates the feature resolution needs 734 ms, while it grows to roughly 4500 ms for 100 candidates per feature.

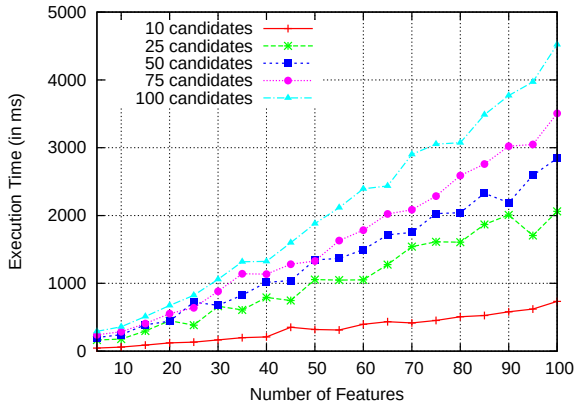


Fig. 9: Feature Resolution Performance

5.2 Rebinding Performance

In the following subsection, we give an evaluation of the different rebinding strategies introduced in Section 4.4. The evaluation is done using the Web service testbed Genesis [43]. 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 of services 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 10. It should be noted that the response time of the best service is decreasing since we increase the variance. 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.

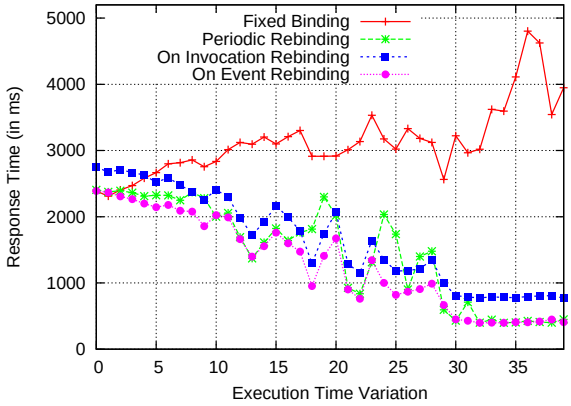


Fig. 10: Rebinding Strategies Performance

Obviously, clients with *fixed* binding usually perceive the worst response time because they are always bound to the same service. Clients using *periodic* rebinding mostly use services with good response time. However, since rebinding is done in pre-defined intervals the bindings are 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 the service is invoked. However, this leads to a constant overhead of about 400 ms which is needed to check the binding and update if necessary. Finally, clients with *OnEvent* rebinding always bind to the best service without invocation overhead because the clients are notified asynchronously when the QoS changes and better services get available. However, the (optional) VRESCo eventing support must be turned on and the client needs a listener Web service. Thus, all rebinding strategies have their strengths and weaknesses, and it depends on the specific situation which strategy to use.

5.3 Mediation Performance

Besides rebinding, we have also evaluated the overhead introduced by the VRESCo mediation facilities. We have again used the Genesis testbed for these tests.

Figure 11 depicts the response time of a single Web service invocation depending on the size of the message sent to the service. We have evaluated four different mediation scenarios: no mediation, mediation using only the VMF built-in functions, mediation using only CS-Script and finally mediation using both built-in functions and CS-Script. Unsurprisingly, unmediated invocations

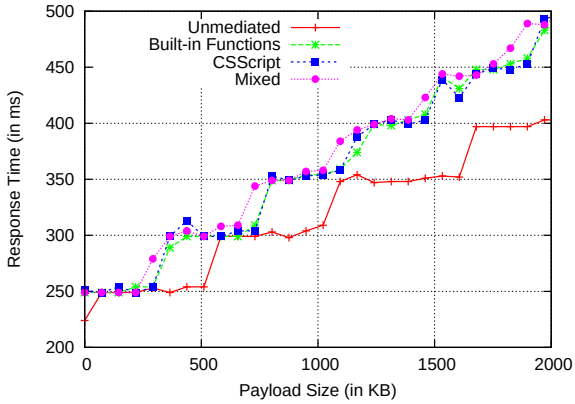


Fig. 11: Mediation Performance (Message Size)

are generally faster than any type of mediation. All types of mediation introduce a similar amount of overhead, which depends solely on the size of the message. For small messages the overhead is in the area of 25 ms, which seems acceptable. However, the overhead increases significantly with the size of the data to move. This is due to data manipulation operations taking longer for bigger message sizes.

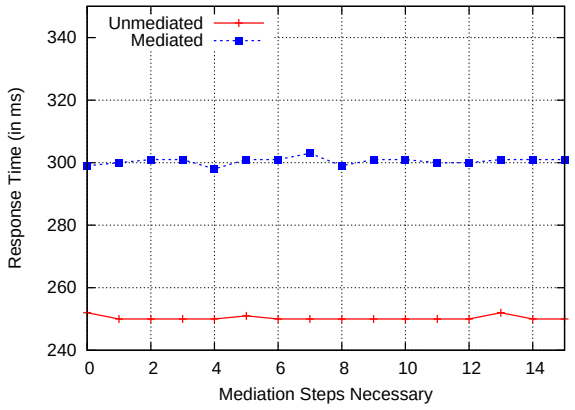


Fig. 12: Mediation Performance (Mediation Steps)

In Figure 12 we have evaluated how the overhead introduced by mediation depends on the amount of mediation necessary. As we can see, the overhead is independent of the amount of mediation necessary, i.e., it is not relevant for the mediation overhead if only simple transformations or more complex ones are necessary. This result differs from what we have reported earlier in [41]. In this work, we have compared various DAIOS mediators including one based on SAWSDL [22] 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.

5.4 Eventing Performance

Finally, we have evaluated the performance of the eventing engine by measuring the throughput of the actual matching between events and subscriptions using a simulation of QoS events. These events were continuously published internally while we increased the number of subscribers and the percentage of matching subscriptions. We measured how many events can be processed per second. For our tests, we ran the experiment for 10 seconds and took the average value of 10 repetitive runs. It should be noted that we do not consider the time needed to actually notify the interested subscribers here (since this is done by a dedicated notification delivery thread pool and varies significantly depending on the notification mechanism, such as E-Mail or Web services).

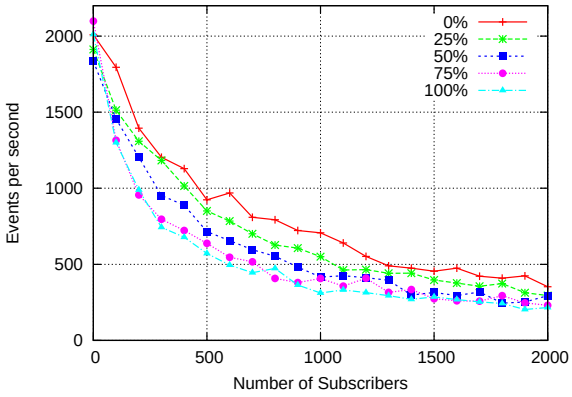


Fig. 13: Eventing Throughput

The results are depicted in Figure 13. It can be seen that the results clearly decrease with the number of matching subscriptions. The throughput is around 2000 events per second without subscriptions and converges to around 300 events per second for 2000 subscriptions.

5.5 Discussion

To conclude the evaluation, we elaborate the experience gained during the implementation of the case study introduced in Section 2 using VRESCO, and discuss the performance results with respect to this case study.

In the first step, we have defined the six features including their input and output data concepts of the process shown in Figure 2b. Then we have implemented the services and published them into the registry. Finally, for features having multiple service candidates (e.g., `Notify_Customer`), we additionally defined the VMF mapping as shown in Listing 3. Defining the metadata has to be carried out as part of the process design which certainly requires some effort, but this is necessary in order to leverage adaptive behavior provided by the dynamic binding and mediation capabilities. This means that there is a tradeoff between the time needed for this effort and the gained flexibility. For instance, if a partner CPO provides a new number porting service, it can be easily integrated into the process by mapping to the existing feature. The process itself remains untouched and there is no downtime involved when integrating this new service.

In the second step, we have implemented the process in C# for reasons of simplicity (due to the fact that it is a simple sequence of activities). For finding service candidates at runtime (e.g., number porting service from partner CPOs), we can make use of the VQL querying mechanism. The performance results of feature resolution (see Figure 9) demonstrate the good query performance in this setting (6 features with less than 10 candidates per feature are queried in less than 60 ms). Once queried, the service candidates can be invoked in a uniform manner due to the service mediation capabilities that use the mapping defined before. As can be seen in Figure 12, the mediation time is independent from the number of mediation steps (i.e., even complex mappings requiring various mediation functions). Additionally, according to Figure 11 the overhead of mediation for the number porting messages is around 50 ms for message payloads up to 1500 KB.

While implementing the process we had to decide which rebinding strategy to use. 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. *Periodic* rebinding seems not adequate since we expect that the services do not change frequently. Since number porting is not time-critical, we opt for *OnInvocation* which has a constant invocation overhead but always finds the best available service, or even better *OnEvent* which also eliminates this invocation overhead. Figure 13 demonstrates that the event throughput is high enough to deal with 2000 (or more) concurrent clients. For the `Notify_Customer` feature, the query used for rebinding should consider current QoS attributes of service candidates (e.g., if multiple notification services are deployed). In contrast to existing query languages, VQL provides the required expressiveness to query optional or prioritized attributes regarding QoS as shown in Listing 1.

6 Conclusion

One of the main promises of Service-oriented Computing was the provisioning of loosely-coupled applications based on the publish-find-bind-execute cycle. In practice, however, these promises could often not be kept due to the lack of expressive service metadata and type-safe querying facilities, explicit support for QoS, as well as support for dynamic binding, invocation and mediation. In this paper, we have proposed the QoS-aware VRESCo runtime environment which has been designed with these requirements in mind. VRESCo offers an extensive structured metadata model and VQL as type-safe query language. Furthermore, we provide dynamic binding, invocation and mediation mechanisms that use pre-defined service mappings. We have evaluated our work regarding performance and discussed the results together with the experience gained in the CPO case study. The results show that the VRESCo runtime is applicable to large-scale adaptive service-centric systems.

As part of our ongoing and future work we want to link the VRESCo eventing [35] and composition [25] mechanisms. Furthermore, we envision to integrate SLA enforcement capabilities on top of VRESCo. Finally, we plan to build a Web-based runtime management tool.

Acknowledgements

We would like to thank Lukasz Juszczyk for providing the Web service testbed Genesis, and our master students Andreas Huber and Thomas Laner for their contribution to VRESCo.

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *IEEE Computer*, vol. 40, no. 11, pp. 38–45, 2007.

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

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

[4] *SOAP Version 1.2*, http://www.w3.org/TR/soap, World Wide Web Consortium (W3C), 2003, uRL: http://www.w3.org/TR/soap/.

[5] *Web Services Description Language (WSDL) 1.1*, http://www.w3.org/TR/wsdl, World Wide Web Consortium (W3C), 2001, uRL: http://www.w3.org/TR/wsdl.

[6] *Universal Description, Discovery and Integration (UDDI)*, Organization for the Advancement of Structured Information Standards (OASIS), Feb. 2005, http://oasis-open.org/committees/uddi-spec/.

[7] *ebXML Registry Services and Protocols*, Organization for the Advancement of Structured Information Standards (OASIS), Mar. 2005, http://oasis-open.org/committees/regrep.

[8] D. Bodoff, M. Ben-Menachem, and P. C. Hung, "Web metadata standards: Observations and prescriptions," *IEEE Software*, vol. 22, no. 1, pp. 78–85, 2005.

[9] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar, "Towards Recovering the Broken SOA Triangle – A Software Engineering Perspective," in *Proceedings of the 2nd International Workshop on Service Oriented Software Engineering (IW-SOSWE'07), co-located with ESEC/FSE'07*, 2007.

[10] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Transactions on the Web*, vol. 1, no. 6, p. 6, 2007.

[11] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "Qos-aware middleware for web services composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311–327, May 2004.

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

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

[14] *Web Services Eventing (WS-Eventing)*, W3C, 2006, http://www.w3.org/Submission/WS-Eventing/.

[15] *Mule Galaxy, v1.5.1*, MuleSource, Inc., Jan. 2009, http://www.mulesource.com/products/galaxy.php.

[16] *WSO2 Registry, v2.0*, WSO2, Inc., Feb. 2009, http://wso2.org/projects/registry.

[17] *WebSphere Service Registry and Repository, v6.2*, IBM, Inc., Jul. 2008, http://www.ibm.com/software/integration/wsrr.

[18] R. Sayre, "Atom: The Standard in Syndication," *IEEE Internet Computing*, vol. 9, no. 4, pp. 71–78, 2005.

[19] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic web services," *IEEE Intelligent Systems*, vol. 16, no. 2, 2001.

[20] *OWL-S: Semantic Markup for Web Services*, World Wide Web Consortium (W3C), 2004, http://www.w3.org/Submission/OWL-S/ (Last accesssed: July 28, 2008).

[21] *Web Service Modeling Language (WSML)*, ESSI WSMO Working Group, Aug. 2008, http://www.wsmo.org/wsml/wsml-syntax.

[22] *Semantic Annotations for WSDL and XML Schema*, http://www.w3.org/TR/sawsdl/, World Wide Web Consortium (W3C), 2007, http://www.w3.org/TR/sawsdl/.

[23] Q. Yu, X. Liu, A. Bouguettaya, and B. Medjahed, "Deploying and managing web services: issues, solutions, and directions," *The VLDB Journal*, vol. 17, no. 3, pp. 537–572, 2008.

[24] J. Harney and P. Doshi, "Selective querying for adapting web service compositions using the value of changed information," *IEEE Transactions on Services Computing*, vol. 1, no. 3, pp. 169–185, 2008.

[25] F. Rosenberg, "Qos-aware composition of adaptive service-oriented systems," Ph.D. dissertation, Vienna University of Technology, Jun. 2009.

[26] C. Platzer and S. Dustdar, "A Vector Space Search Engine for Web Services," in *Proceedings of the 3rd European IEEE Conference on Web Services (ECOWS'05)*, 2005.

[27] Q. Yu and A. Bouguettaya, "Framework for web service query algebra and optimization," *ACM Transactions on the Web (TWEB)*, vol. 2, no. 1, pp. 1–35, 2008.

[28] C. Pautasso and G. Alonso, "Flexible Binding for Reusable Composition of Web Services." in *Proceedings of the 4th International Workshop on Software Composition (SC'2005)*, 2005, pp. 151–166.

[29] M. D. Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo, and E. D. Nitto, "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)*. New York, NY, USA: ACM Press, 2006, pp. 74–80.

[30] P. Leitner, F. Rosenberg, and S. Dustdar, "DAIOS – Efficient Dynamic Web Service Invocation," *IEEE Internet Computing*, vol. 13, no. 3, pp. 30–38, 2009.

[31] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "Service Provenance in QoS-Aware Web Service Runtimes," in *Proceedings of the 7nd International Conference on Web Services (ICWS'09)*. IEEE Computer Society, jul 2009.

[32] F. Rosenberg, P. Leitner, A. Michlmayr, and S. Dustdar, "Integrated Metadata Support for Web Service Runtimes," in *Proceedings of the Middleware for Web Services Workshop (MWS'08), co-located with EDOC'08*. IEEE Computer Society, Sep. 2008.

[33] F. Rosenberg, C. Platzer, and S. Dustdar, "Bootstrapping Performance and Dependability Attributes of Web Services," in *Proceedings of the IEEE International Conference on Web Services (ICWS'06), Chicago, USA*, Sep. 2006.

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

[35] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar, "Advanced Event Processing and Notifications in Service Runtime Environments," in *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)*. ACM, 2008.

[36] F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar, "Towards composition as a service - a quality of service driven approach," in *Proceedings of the 1st IEEE Workshop on Information and Software as Service (WISS'09), co-located with ICDE'09*. IEEE Computer Society, March 2009.

[37] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar, "End-to-End Versioning Support for Web Services," in *Proceedings of the International Conference on Services Computing (SCC 2008)*. IEEE Computer Society, Jul. 2008.

[38] *Hibernate Reference Documentation v3.3.1*, Red Hat, Inc., 2008, http://www.hibernate.org/.

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

[40] *Esper Reference Documentation*, EsperTech, 2009, http://esper.codehaus.org/.

[41] P. Leitner, A. Michlmayr, and S. Dustdar, "Towards flexible interface mediation for dynamic service invocations," in *Proceedings of the 3rd Workshop on Emerging Web Services Technology (WEWST'08), co-located with ECOWS'08*, 2008.

[42] "CS-Script – The C# Script Engine." [Online]. Available: http://www.csscript.net/

[43] L. Juszczyk, H.-L. Truong, and S. Dustdar, "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'08)*. IEEE Computer Society, 2008, pp. 131–140.

Anton Michlmayr received the MSc degree in computer science from Vienna University of Technology in 2005. He is currently a PhD candidate and university assistant in the Distributed Systems Group at Vienna University of Technology. His research interests include software architectures for distributed systems with an emphasis on distributed event-based systems and service-oriented computing. More information can be found at http://www.infosys.tuwien.ac.at/Staff/michlmayr.

Florian Rosenberg is a PhD candidate and university assistant in the Distributed System Group at Vienna University of Technology graduating in June 2009. His general research interests include service-oriented computing and software engineering. He is particularly interested in all aspects related to QoS-aware service composition. More information can be found at http://www.infosys.tuwien.ac.at/Staff/rosenberg.

Philipp Leitner has a BSc and MSc in business informatics from Vienna University of Technology. He is currently a PhD candidate and university assistant at the Distributed Systems Group at the same university. Philipp's research is focused on middleware for distributed systems, especially for SOAP-based and RESTful Web services. Additionally, he has done work in the area of P2P computing, network management and security of distributed systems. More information can be found at http://www.infosys.tuwien.ac.at/Staff/leitner.

Schahram Dustdar is Full Professor of Computer Science with a focus on Internet Technologies heading the Distributed Systems Group, Institute of Information Systems, Vienna University of Technology (TU Wien) where he is director of the Vita Lab. He is also Honorary Professor of Information Systems at the Department of Computing Science at the University of Groningen (RuG), The Netherlands. He is Chair of the IFIP Working Group 6.4 on Internet Applications Engineering and a founding member of the Scientific Academy of Service Technology. More information can be found at http://www.infosys.tuwien.ac.at/Staff/sd.