This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

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

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

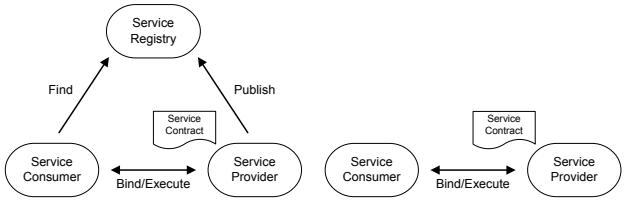
Abstract—Service-oriented Computing has recently received a lot of attention from both academia and industry. However, current service-oriented solutions are often not as dynamic and adaptable as intended because the publish-find-bind-execute cycle of the SOA triangle is not entirely realized. In this paper, we highlight some issues of current Web service technologies, with a special emphasis on service metadata, Quality of Service, service querying, dynamic binding and service mediation. Then, we present the Vienna Runtime Environment for Service-oriented Computing (VRESCo) that addresses these issues. We give a detailed description of the different aspects by focusing on service querying and service mediation. Finally, we present a performance evaluation of the different components, together with an end-to-end evaluation to show the applicability and usefulness of our system.

Index Terms—Web Services Publishing and Discovery, Metadata of Services Interfaces, Advanced Services Invocation Framework

1 Introduction

During the last few years, Service-oriented Architecture (SOA) and Service-oriented Computing (SOC) [1] has gained acceptance as a paradigm for addressing the complexity that distributed computing generally involves. In theory, the basic SOA model consists of three actors that communicate in a loosely coupled way as shown in Figure 1a. *Service providers* implement services and make them available in *service registries*. *Service consumers* (also called *service requesters*) query service information from the registry, bind to the corresponding service provider, and finally execute the service. Due to 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, which increases organizational agility. Web services [2] represent the most common realization of SOA, building on the standards SOAP [3] for communication, WSDL [4] for service interface descriptions, and UDDI [5] for registries.

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 [6] did not succeed. We think this is partly due to their limited querying support that only provides 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 2005. As a result, service registries are often missing in service-centric systems (i.e., no *publish* and *find* primitives). This leads to point-to-point solutions where service endpoints are exchanged at design-time (e.g., using E-mail) and service consumers statically bind to them (see Figure 1b).



(a) SOA Model (b) SOA Practice

Fig. 1: SOA Theory vs. Practice (adapted from [7])

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 non-functional QoS attributes such as response time (in addition to functional attributes).

In this paper, 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. The main contribution is the presentation of the VRESCo service runtime environment that aims at solving some of these issues. To be more specific, the present paper focuses on service metadata, QoS and service querying, plus dynamic binding, invocation, and mediation of services. Additionally, we provide an extensive performance evaluation of the different components and an end-to-end evaluation of the overall

- *Anton Michlmayr, Philipp Leitner and Schahram Dustdar are with the Distributed Systems Group, Vienna Univ. of Technology, Argentinierstrasse 8, 1040 Vienna, Austria. E-mail: {lastname}@infosys.tuwien.ac.at*
- *Florian Rosenberg is with CSIRO ICT Centre, GPO Box 664, Canberra ACT 2601, Australia. E-mail: florian.rosenberg@csiro.au*

Digital Object Indentifier 10.1109/TSC.2010.20

runtime, that shows the applicability of our approach.

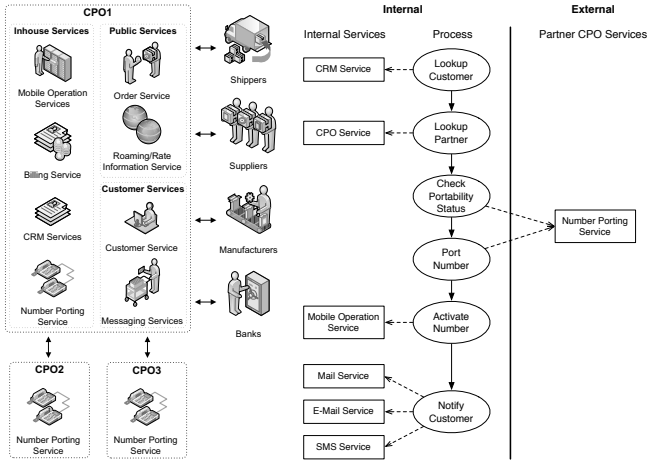
The remainder of this paper is organized as follows: Section 2 presents an illustrative example and summarizes some issues of SOC research and practice. Section 3 describes the details of the VRESCo runtime environment, while Section 4 gives a thorough evaluation of our work. Section 5 introduces related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



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

Fig. 2: CPO Case Study

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., SMS Service) are used by customers of CPO1, whereas *Inhouse Services* (e.g., CRM Services) represent internal services which should only be accessed by the different departments of CPO1. Besides that, CPO1 consumes services from its partners (e.g., cell phone manufacturers and suppliers) and competitors (e.g., CPO2 and CPO3). As discussed later, this scenario bears several challenges that are typical in service-centric software engineering.

According to European law, consumers can keep their mobile phone number when switching to another CPO. Figure 2b shows a simplified number porting process (depicted as oval boxes). 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 CRM Service, the external 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, E-mail, etc.).

2.2 SOC Challenges

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. The contribution of VRESCo is to address these challenges in a comprehensive service runtime environment.

- *Service Metadata.* Service interface description languages such as WSDL focus on the interface needed to invoke a 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.
- *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 [5] and ebXML [6]. 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 [7]. Besides service discovery, another issue is how to select a service from a pool of service candidates [9] by means of a querying language. For instance, CPO1 may want to select the SMS Service with the highest availability.
- *Quality of Service (QoS).* In enterprise scenarios QoS plays a crucial role [10]. This includes both network-level attributes (e.g., latency and availability), and application-level attributes (e.g., response time and throughput). The QoS model should be extensible to allow service providers to adapt it for their needs. Furthermore, QoS must be monitored accordingly so that users can be notified when the measured values violate 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. However, in practice this requires identical service interfaces, which is often not the case. Therefore, we argue that the *bind* and *execute* primitives of SOA are not solved sufficiently. This raises the need for mechanisms that mediate between alternative services possibly having different interfaces.

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.

Besides these core challenges, other aspects such as *service versioning* [11] or *event processing* [12] are of crucial importance for SOC. However, a detailed description is out of scope of this paper, and the interested reader is referred to our previous work.

3 System Description

This section describes in detail the VRESCo runtime which was first sketched in [7]. Besides an architectural overview, we discuss service metadata and querying, as well as dynamic binding together with our service mediation approach.

3.1 Overview

The architectural overview of VRESCo is shown in Figure 3, which is adapted from [13]. 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 [14] has been integrated into the Client Library, and provides stubless, protocol-independent, and message-driven invocation of 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 [13]. Services and associated metadata are stored in the Registry Database which is accessed using the Object-Relational Mapping (ORM) Layer. Finally, the QoS Monitor is responsible for regularly measuring the current QoS values. The overall system is implemented in C# using the Windows Communication Foundation [15]. Due to the platform-independent architecture, the Client Library can be provided for different platforms (e.g., C# and Java).

There are several 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 the Registry Database. The Notification Engine informs users when certain events of interest occur inside the runtime, while the Composition Engine [16] provides mechanisms to compose services by specifying hard and soft constraints on 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 for metadata, services and QoS), the Query Engine, and the dynamic binding, invocation and mediation mechanisms.

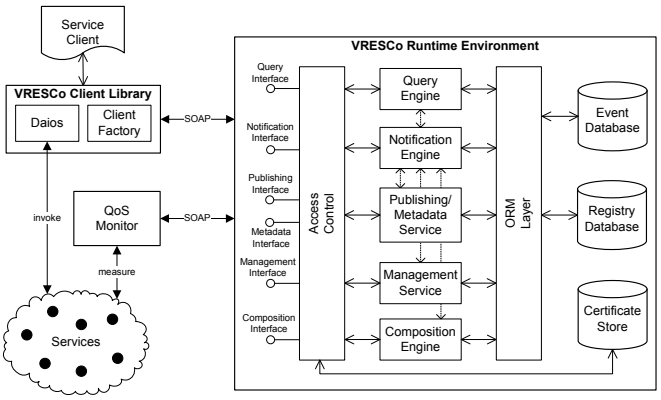


Fig. 3: VRESCo Overview Architecture

3.2 Service Metadata

The VRESCo runtime provides a service metadata model capable of storing information about services. This is needed to capture the purpose of services, which enables mediation between services that perform the same task. In this section, we describe service metadata and give examples from the CPO case study.

3.2.1 Metadata Model

The VRESCo metadata model introduced in [17] is depicted in Figure 4. The main building blocks of this model are *concepts*, which represent the definition of entities 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` and `Port_Number`). *Features* are associated with *categories* which express the purpose of services (e.g., `PhoneNumberPorting`).
- *Data concepts* represent concrete entities in the domain (e.g., `customer` or `phone_number`) 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)` expresses the portability status of a given *argument* `Number`).

Furthermore, *features* can have *pre-* and *postconditions* expressing logical statements that have to hold before and after the execution of the *feature*. Both types of conditions are composed of multiple *predicates*, each having a number of optional *arguments*. These *arguments* 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 a *feature*. For instance, the *feature* `Check_Status` from our CPO case study could have the *flow predicate* `requires(Customer)` as *precondition* and `produces(Portability-Status)` as *postcondition*.

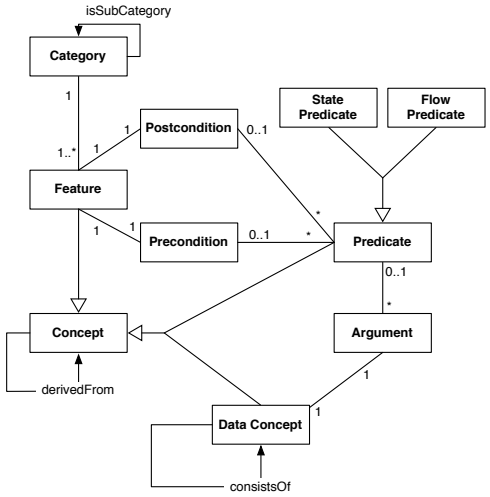


Fig. 4: Service Metadata Model [17]

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

3.2.2 Service Model

The VRESCo service model constitutes the basic information of concrete services that are managed by VRESCo. 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 and represent QoS on a service runtime level.

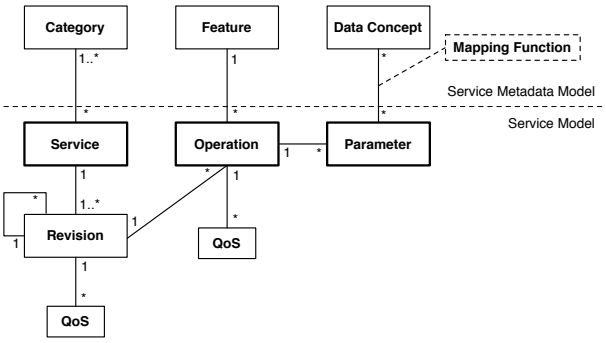


Fig. 5: Service Model to Metadata Model Mapping

A concrete *service* (e.g., Number Porting Service of CPO1) defines the basic information of a service (e.g., name, description, owner, etc.) and consists of a least one service revision. A *service revision* (e.g., the most recent version, or a stable one) contains all technical information that is necessary to invoke the service (e.g., a reference to the WSDL file) and represents a collection of *operations* (e.g, `Check_Status`). Every operation may have a number of *input parameters* (e.g., `Customer`), and may return one or more *output parameters* (e.g., `PortabilityStatus`). Revisions can have parent and child revisions that represent a complete versioning graph of a concrete service [11]. Both revisions and operations can have a number of *QoS attributes* (e.g., response time is 1200 ms) 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 (if a service is not available, all operations are generally also unavailable). In Section 3.5, we show how concrete services are mapped to the metadata and service model in order to perform service mediation.

3.2.3 QoS Model

Besides functional attributes described in the metadata model, non-functional attributes are also important. For instance, in our case study CPO1 may want to always bind to the Notification Service having the lowest response time. Therefore, QoS attributes can be associated with each service revision and operation in VRESCo. These QoS attributes can be either specified manually using the Management Service, or measured automatically (e.g., using the QoS Monitor introduced in [18]).

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	$q_{tp}(t_0, t_1, r) = \frac{r}{t_1 - t_0}$	invocations/s

TABLE 1: QoS Attributes

Table 1 briefly summarizes the QoS attributes that are currently considered in VRESCo. Latency represents the time a request needs on the wire. It is calculated as the average value of n individual measuring points. Response time consists of the latency for request and response plus the execution time of the service. Availability 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). Accuracy 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 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 these pre-defined QoS attributes, users can define additional QoS properties for service revisions or operations.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

3.3 Querying Approach

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 architecture of VQL followed by query specification and query processing.

3.3.1 Architecture

The VQL architecture was driven by the following requirements. 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 [19], 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 [20]).

Moreover, it should be possible to define both mandatory and optional criteria by introducing different querying strategies that enable fuzzy or priority-based querying (e.g., services must have a response time below 500 ms and should be provided by company X). Finally, 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).

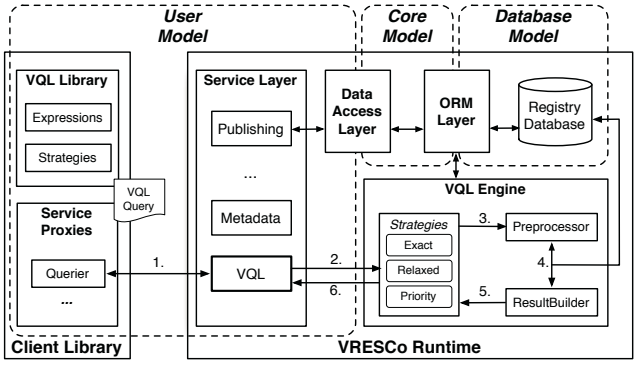


Fig. 6: VQL Architecture

The architecture of the VQL framework is shown in Figure 6. 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 [19] 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 corresponding mapping between the two models is defined at design time using .NET attributes [21].

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 [20] 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 Engine (step 2). The details of query processing (steps 3–5) are described in Section 3.3.3. Finally, the results are sent back to the query requester (step 6).

3.3.2 Query Specification

After describing requirements and architecture of the querying framework, we present how queries are specified. 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 specify the ordering of the query results (e.g., sort ascending by ID).
- *Querying Strategy* S finally defines how the query should be executed (e.g., exact or fuzzy matches).
- *Result Limit* L can be used to restrict the number of results (e.g., 10 or 0, which represents no limit).

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 3.3.4. However, the main motivation is to allow the specification of mandatory and optional criteria.

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 2 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.

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 2: VQL/SQL Translation

Listing 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 5–7) 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 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 8–14). 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 13 represents a shortcut for an `And` expression. All three `Match` criteria use priority values as third parameter to define the importance of a criterion.

```
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",
7                         "NotifyCustomer"));
8 query.Match(Expression.Eq("Service.Owner.Company",
9                           "CompanyX"), 1);
10 query.Match(Expression.Like("Tags.Property.Name",
11     "STABLE", LikeMatchMode.Start), 3);
12 query.Match(
13     Expression.Eq("QoS.Property.Name", "ResponseTime") &
14     Expression.Lt("QoS.DoubleValue", 1000.0), 5);
15
16 // execute query
17 var querier = VRESCoClientFactory.CreateQuerier(
18     "username", "password");
19 var results = querier.FindByQuery(query, 10,
20     QueryMode.Priority) as IList<ServiceRevision>;
```

Listing 1: VQL Sample Query

The query is finally executed (lines 17–20) 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.

3.3.3 Query Processing

Query processing is illustrated in Figure 6. When the query is sent to the VQL Engine, the specified querying strategy is executed, which is implemented using the strategy design pattern [22]. The query is forwarded to the *Preprocessor* component (step 3), 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 4). 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. For both models, however, the *ResultBuilder* guarantees type-safety of the results, which are finally sent back to the client (step 5).

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

```
1: if ( isUserObject(R) ) then
2:     R ←MapUserToCoreObject(R)
3: end if
4: assocInfo ← R
5: for all ( crit ∈ C ) do
6:     for all ( expr ∈ GetExpressions(crit) ) do
7:         assocInfo←assocInfo ∪ ResolveAssoc(expr)
8:         propInfo← params ∪ ResolveProp(expr)
9:     end for
10: end for
11: query ← BuildFrom(assocInfo, propInfo, S)
12: query ← BuildWhere(query, assocInfo, propInfo, S)
13: query ← BuildOrder(query, O)
14: 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 1–3). The *Preprocessor* then iterates over all criteria and expressions (lines 5–10). 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 8 of Listing 1: The property path `Service.Owner.Company` represents two associations `Service` and `Owner` that will be resolved using 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 11–13), according to the VQL/SQL translation shown in Table 2.

3.3.4 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 2).

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 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 criterion in order to accomplish a weighted matching of results. Therefore, each `Match` criterion allows to append a weight to specify its priority, 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 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.

3.4 Dynamic Binding

Dynamic binding is claimed to be one of the main advantages of SOA. 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. We summarize all available strategies in Table 3.

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

TABLE 3: Rebinding Strategies

All rebinding strategies 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 background queries on a regular basis, which 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 since service bindings are checked before every invocation. Finally, *OnEvent* rebinding uses the VRESCo Event Notification Engine [12] to combine the advantages of all strategies. Therefore, clients use subscriptions for defining in which situations to rebind, which is then triggered by events.

3.5 Service Mediation

Dynamic binding as described above naturally brings up the problem of how differences in service interfaces can be resolved at runtime. In this section, we introduce the VRESCo Mapping Framework (VMF) that handles the mapping from abstract features to concrete service operations (as described in Section 3.2), and perform mediation between different services that implement the same feature. The elements of the service model are mapped to our service metadata model as follows (see Figure 5): 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 VRESCo Composition Engine [16] (i.e., features will be represented as compositions of several service operations). The input and output parameters of 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 scripts, which will be discussed extensively below.

In general, the mediation approach follows the "feature-driven" metadata model. Therefore, a client that wants to invoke a service does not provide the input of the concrete service directly but 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).

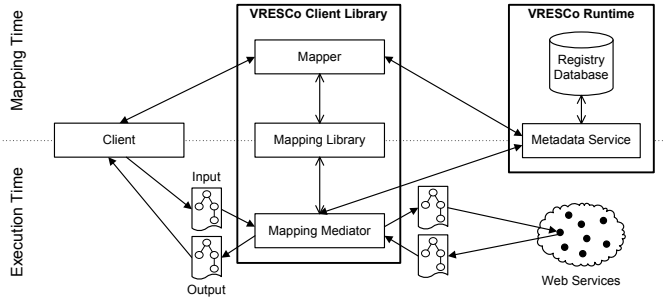


Fig. 7: VMF Architecture

Figure 7 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 scripts for each service. This information is stored in the VRESCo Registry Database using the Metadata Service. Secondly, at execution time, DAIOS is used as a dynamic service invocation framework. The *Mediator* component is used as an interceptor in DAIOS following the ideas presented in [23]. This mediator retrieves the lifting and lowering scripts from the VRESCo Metadata Service at runtime, and executes the corresponding mapping. This is done by applying all mapping functions sequentially, in the order they have been specified. In that sense, VMF implements an imperative, interpreted domain-specific language. In its current form, VMF does not optimize mapping scripts in any way.

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`)
Math	Basic mathematical and logical operations (e.g., `addition`, `round`, `and`, `or`)
CSScript	Define complex mappings directly in C#

TABLE 4: 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 4 (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 [24]. Essentially, this allows to deploy custom mapping functions by using the full power of the C# programming language. For instance, this can be used to invoke external Web services at mediation time.

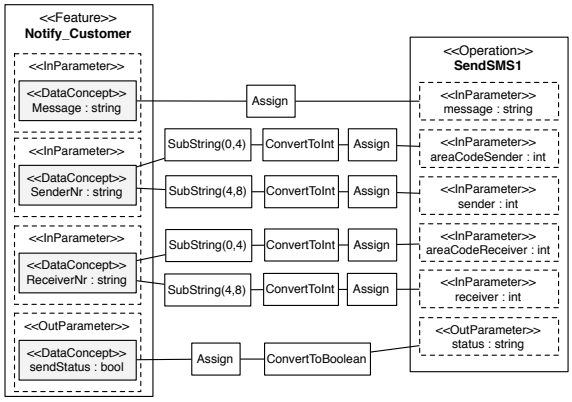


Fig. 8: VMF Mapping Example

We give a concrete mapping example in Figure 8. In this example, the abstract feature `Notify_Customer` from the CPO case study (see Section 2) is mapped to the concrete operation `SendSMS1`. The feature provides three input parameters and produces one output parameter. The parameter `Message` is identical in both interfaces, and can therefore be mapped directly (using only `Assign`). 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.

Listing 2 illustrates the first two mappings (`Message` and `SenderNr`) in C# code. Lines 4–5 show how the Mapper is created for feature `Notify_Customer` and operation `SendSMS1`. Both objects have to be queried beforehand (not shown in Listing 2 for brevity). The `Assign` function is again used as a connector to link the `Message` from the feature to the `Message` of the operation, whereas `mapper.AddMappingFunction()` adds the function to the mapping. Lines 14–21 get the area code from the feature's `SenderNr` as substring and convert it with the `ConvertToInt` function to an integer

```
// query NotifyCustomer and SendSMS1 instances using VQL

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

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

// get AreaCode, convert to int and map it to operation
Substring acSenderStr = new Substring(
    mapper.FeatInParams[1], 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]));
```

Listing 2: VMF Mapping Example Code

which is finally assigned to operation's input parameter `AreaCodeSender`. All further mappings from Figure 8 are implemented analogously.

4 Evaluation

In this section, we give an evaluation of the VRESCo runtime focusing on the topics covered in this paper. The purpose of this evaluation is twofold: Firstly, we show the runtime performance regarding service querying, rebinding, and mediation by using synthetic data. The main goal of this evaluation is to analyze the performance impact of each aspect in isolation. Secondly, we combine these aspects into a coherent end-to-end evaluation using an order processing workflow. The main goal 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. Moreover, 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 generation tool Genesis [25]. These testbeds are described in detail in the corresponding subsections.

4.1 Querying Performance

First of all, we show the performance of the VQL Engine, which has been measured using the query shown in Listing 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. It should be noted that 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

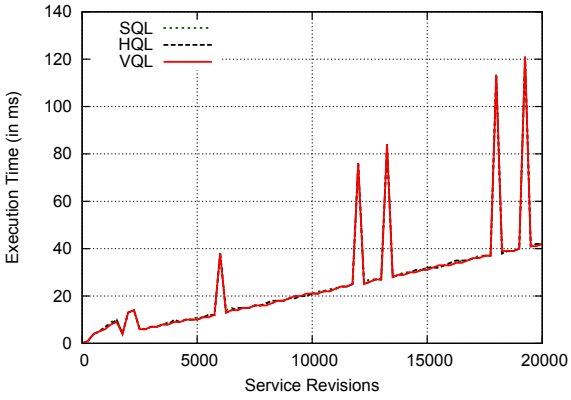


Fig. 9: Query Performance (NL)

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.

Figure 9 compares the performance of the queries generated by SQL, HQL and VQL. Therefore, the query from Listing 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. In this regard, all approaches exhibit the same peaks, which are due to internal processing of the database.

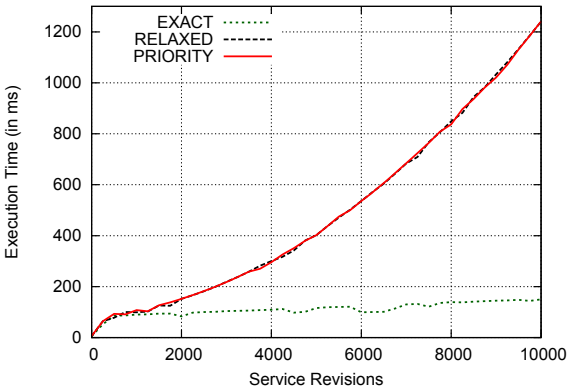


Fig. 10: Querying Strategies (User, L10)

Figure 10 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 5 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 (G) indicates how long the *Preprocessor* needs to analyze and generate the query. Execution (E) depicts the actual query execution time, while Conversion (C) represents the time needed by the *ResultBuilder* to convert the query results.

Revisions	User			Core		
	G	E	C	G	E	C
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 5: VQL Query Processing (in ms, User/Core, L10)

The results show that G 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, E is almost equal for both approaches. Finally, the table indicates that C 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 [19]). In this setting, the time for C is constant for all revisions due to the result limit of 10.

4.2 Rebinding Performance

In the following subsection, we give an evaluation of the different rebinding strategies introduced in Section 3.4. 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 11. 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.

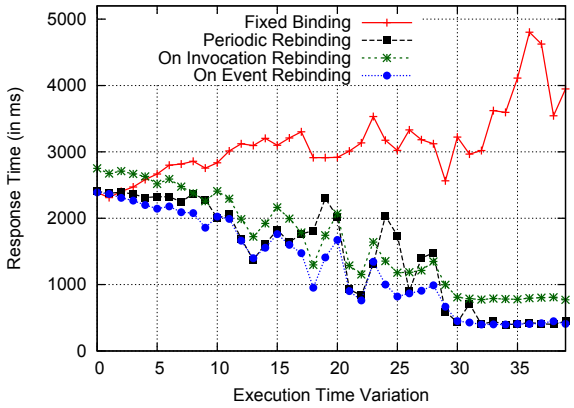


Fig. 11: Rebinding Strategies Performance

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. It should be noted that the performance of the Event Engine is sufficient which is detailed in [12]. Thus, all rebinding strategies have their strengths and weaknesses, and it depends on the specific situation which strategy to use.

4.3 Mediation Performance

Besides rebinding, we have also evaluated the overhead introduced by the VRESCO mediation facilities. We have again used the GENESIS tool for these tests.

Figure 12 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 (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,

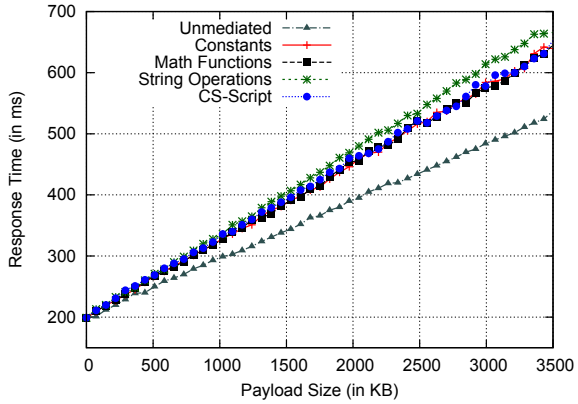


Fig. 12: Mediation Performance (Message Size)

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.

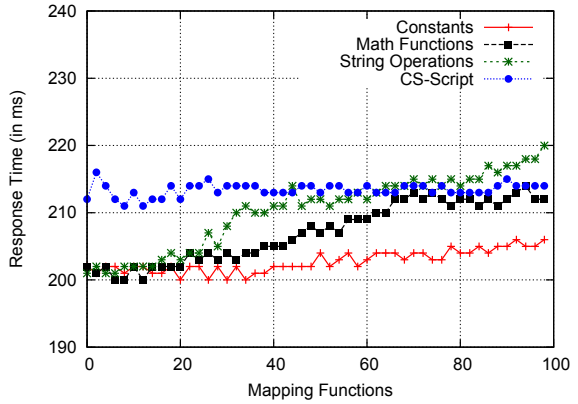


Fig. 13: Mediation Performance (Mediation Steps)

In Figure 13 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, which seems acceptable. As before, the overhead introduced by string operations heavily depends on the size of the strings to modify. 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 around 10 ms since the main overhead is the initialization of the scripting engine, while the execution of the actual script is negligible (as long as the script does not do any heavy computation, which would not be typical for mapping scenarios).

4.4 End-to-End Evaluation and Discussion

The end-to-end scenario combines all aforementioned aspects (i.e., querying, rebinding, mediation and invocation) into a larger order processing case study with the goal of ordering new cell phone contracts online (including mobile phone and SIM card). We implemented this workflow in C#. It consists of 19 overall activities split into 4 subprocesses. Basically, the process starts upon receiving an order via the company Web site. Afterwards the internal stock is checked for the availability of the phone and the SIM card. If one of those components is missing, it is ordered by using one of the internal or external suppliers, which is followed by a contracting subprocess. This subprocess creates a new contract and, if necessary, it adds a new customer to the CRM system. If the customer wants to transfer her old number, the number porting subprocess as depicted in Figure 2b is executed. Finally, the payment and shipping subprocesses are enacted and the cell phone number is activated in the GSM network.

The services used in the case study have been deployed on a different machine using GENESIS [25]. For each internal service (e.g., CRM, contracting) we have deployed only one alternative, whereas for each external service (e.g., Credit Card Service) multiple alternatives are available (between 60 and 250). For the internal notification service which is used to notify customers of their order status (using SMS, E-mail, mail, etc.) 30 alternatives are provided. This service is the only one that requires significant mediation. We use GENESIS to simulate a response time of 30–100 ms for each service.

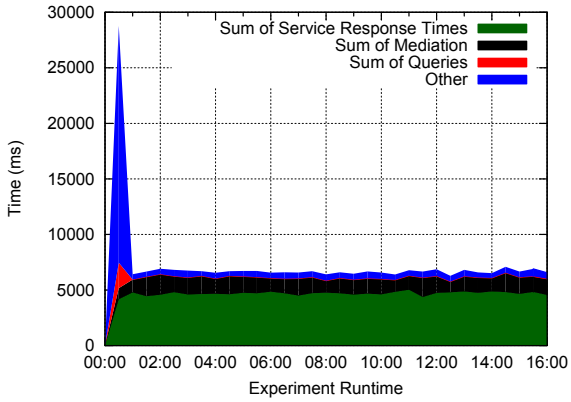


Fig. 14: End-to-End Performance

In Figure 14, 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. In order to get a big number of rebindings during our experiment time we have chosen a rebinding interval of 5 sec. 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 12. Together, service response times and mediation accounts 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 logics. Please note that querying and an occasional rebinding still happens after the initialization phase, but it is no longer part of the average process execution times (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 (as shown in Figure 14). In case of the *OnInvocation* rebinding strategy, there would be querying and rebinding overhead in the overall process execution time, whereas for *OnDemand* and *OnEvent* 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, or even better *OnEvent* which also eliminates this invocation overhead.

5 Related Work

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

Currently, several approaches and standards for service registries exist. We have compared some existing solutions with the VRESCo runtime, considering a carefully selected range of established standards, mature open-source frameworks and commercial tools. We consider the standards UDDI [5] and ebXML [6] (with special emphasis on the registry), Mule ESB and Galaxy repository [26], WSO2 ESB and registry [27], and IBM WebSphere [28] (including ESB, service registry and repository). Our comparison in Table 6 is structured according to the challenges introduced in Section 2.

Generally, all systems allow to store service metadata. 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, whereas 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 [29]). 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 QoS only in terms of WS-Security and WS-ReliableMessaging. However, none of these frameworks except VRESCo provide 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. All systems except UDDI and VRESCo allow to store multiple versions of service 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 [11]. Finally, all approaches provide basic event notifications (e.g., if services are published) using E-mail, Web service 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 [30] (such as OWL-S [31], WSML [32] and SAWSDL [33]). 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 [34]. In these approaches, service querying and selection play a crucial role, especially regarding service composition (e.g., [10], [35], [16]). However, the query models of current registries and Web service search engines [36] mainly focus on keyword-based matching of service properties which often do not cover the rich semantics of service metadata.

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 6: Related Enterprise Registry Approaches

Yu and Bouguettaya [37] 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., [38], [39]).

Pautasso and Alonso [38] 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. [39] present the WS-Binder framework for enabling dynamic binding within WS-BPEL processes. Their approach uses proxies to separate abstract services from concrete service 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.

6 Conclusion

One of the main promises of SOC is the provisioning of loosely-coupled applications based on the publish-find-bind-execute cycle. In practice, however, these promises can 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 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 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 [12] and composition [16] mechanisms. Furthermore, we envision to integrate SLA enforcement capabilities on top of VRESCo.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-Cube). Additionally, 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] 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.

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

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

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

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

[7] 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*. ACM, 2007.

[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] 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.

[10] 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.

[11] 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, 2008.
[12] 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.
[13] ——, "Service Provenance in QoS-Aware Web Service Runtimes," in *Proceedings of the 7th International Conference on Web Services (ICWS'09)*. IEEE Computer Society, 2009.
[14] P. Leitner, F. Rosenberg, and S. Dustdar, "Daios – Efficient Dynamic Web Service Invocation," *IEEE Internet Computing*, vol. 13, no. 3, pp. 30–38, 2009.
[15] J. Löwy, *Programming WCF Services*. O'Reilly, 2007.
[16] F. Rosenberg, P. Celikovic, A. Michlmayr, P. Leitner, and S. Dustdar, "An End-to-End Approach for QoS-Aware Service Composition," in *Proceedings of the 13th International Enterprise Computing Conference (EDOC'09)*. IEEE Computer Society, 2009.
[17] 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, 2008.
[18] 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)*. IEEE Computer Society, 2006.
[19] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.
[20] *Hibernate Reference Documentation v3.3.1*, Red Hat, Inc., 2008, http://www.hibernate.org/.
[21] J. Liberty and D. Xie, *Programming C# 3.0*. O'Reilly Media, Inc., 2007.
[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[23] 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.
[24] O. Shilo, "CS-Script – The C# Script Engine," 2009, http://www.csscript.net/.
[25] 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.
[26] *Mule Galaxy, v1.5.1*, MuleSoft, Inc., Nov. 2009, http://www.mulesoft.org/display/GALAXY/Home.
[27] *WSO2 Registry, v2.0*, WSO2, Inc., Feb. 2009, http://wso2.org/projects/registry.
[28] *WebSphere Service Registry and Repository, v6.2*, IBM, Inc., Jul. 2008, http://www.ibm.com/software/integration/wsrr.
[29] R. Sayre, "Atom: The Standard in Syndication," *IEEE Internet Computing*, vol. 9, no. 4, pp. 71–78, 2005.
[30] S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic Web Services," *IEEE Intelligent Systems*, vol. 16, no. 2, 2001.
[31] *OWL-S: Semantic Markup for Web Services*, World Wide Web Consortium (W3C), 2004, http://www.w3.org/Submission/OWL-S/.
[32] *Web Service Modeling Language (WSML)*, ESSI WSMO Working Group, 2008, http://www.wsmo.org/wsml/wsml-syntax.
[33] *Semantic Annotations for WSDL and XML Schema*, World Wide Web Consortium (W3C), 2007, http://www.w3.org/TR/sawsdl/.
[34] 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.
[35] 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.
[36] 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)*. IEEE Computer Society, 2005.
[37] 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.
[38] 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)*. Springer, 2005.
[39] 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)*. ACM, 2006.

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 currently a research scientist at the CSIRO ICT Centre in Australia. He received his PhD in June 2009 with a thesis on "QoS-Aware Composition of Adaptive Service-Oriented Systems" while working as a research assistant at the Distributed Systems Group, Vienna University of Technology. His general research interests include service-oriented computing and software engineering. He is particularly interested in all aspects related to QoS-aware service composition and adaptation. More information can be found at http://www.florianrosenberg.com.

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. 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, Vienna University of Technology (TU Wien). He is also Honorary Professor of Information Systems at the Department of Computing Science at the University of Groningen (RuG), The Netherlands. Since 2009 he is an ACM Distinguished Scientist. More information can be found at http://www.infosys.tuwien.ac.at/Staff/sd.