Chapter 11

VRESCo – Vienna Runtime Environment for Service-oriented Computing

Waldemar Hummer, Philipp Leitner, Anton Michlmayr, Florian Rosenberg, and Schahram Dustdar

Abstract Throughout the last years, the Service-Oriented Architecture (SOA) paradigm has been promoted as a means to create loosely coupled distributed applications. In theory, SOAs make use of a service *registry*, which can be used by providers to publish their services and by clients to discover these services in order to execute them. However, service registries such as UDDI did not succeed and are rarely used today. In practice, the binding often takes place at design time (for instance by generating client-side stubs), which leads to a tighter coupling between service endpoints. Alternative solutions using dynamic invocations often lack a data abstraction and require developers to construct messages on XML or SOAP level. In this paper we present *VRESCo*, the *Vienna Runtime Environment for Service-oriented Computing*, which addresses several distinct issues that are currently prevalent in Service-Oriented Architecture (SOA) research and practice. VRESCo reemphasizes the importance of registries to support dynamic selection, binding and invocation of services. Service providers publish their services and clients retrieve the data stored in the registry using a specialized query language. The data model distinguishes between abstract *features* and concrete service implementations, which enables grouping of services according to their functionality. An abstracted message format allows VRESCo to *mediate* between services which provide the same feature

Waldemar Hummer
Vienna University of Technology, Austria e-mail: waldemar@infosys.tuwien.ac.at

Philipp Leitner
Vienna University of Technology, Austria e-mail: leitner@infosys.tuwien.ac.at

Anton Michlmayr
Vienna University of Technology, Austria e-mail: michlmayr@infosys.tuwien.ac.at

Florian Rosenberg
CSIRO ICT Centre, GPO Box 664, Canberra ACT 2601, Australia e-mail: florian.rosenberg@csiro.au

Schahram Dustdar
Vienna University of Technology, Austria e-mail: dustdar@infosys.tuwien.ac.at

but use a different message syntax. Furthermore, VRESCo allows for explicit *versioning* of services. In addition to functional entities, the VRESCo service metadata model contains QoS (Quality of Service) attributes. Clients can be configured to dynamically rebind to different service instances based on the QoS data. The paper presents an illustrative scenario taken from the telecommunications domain, which serves as the basis for the discussion of the features of VRESCo.

11.1 Introduction

In the course of the last years, software engineering research and practice have put remarkable focus on the Service-Oriented Architecture (SOA) [20] paradigm, which propagates the use of services – autonomous applications made available in a computer network using standardized interface description and message exchange – as a means to create decoupled, distributed, composite applications in heterogeneous environments. *Web Services* [29] represent the most common way of implementing SOAs. Conceptually, SOA involves three main actors: 1) *service providers* implement services and make them available at a certain location (endpoint) in the network; 2) *service registries* store information about services, and providers can publish their services in such registries; 3) *service consumers* discover (find) services by querying a service registry, bind to the obtained service references and execute the services' operations. This model of three collaborating actors is often referred to as the *SOA triangle*. It has been argued that currently the SOA triangle is actually *broken* [16], since the binding between consumer and service provider often happens at design-time and service registries are rarely used in practice. This is largely due to the limited success of service registry standards such as UDDI [19].

Furthermore, dynamic binding and invocation of services is often only supported for services having the same syntactical (or technical) interface. Since there is no standard mechanism to describe the logical equivalence of service operations using service metadata, service consumers are experiencing difficulties determining whether two service implementations actually perform the same task. Even if clients are aware of the concrete service endpoints that are available for a certain task, the problem of different syntactical interfaces remains. One possibility would be to provide input message templates for all possible service versions at design time. However, a more dynamic and flexible approach to mediation between diverse interfaces is desirable. The same applies to cases in which different versions of one and the same service exist. So far, service *versioning* is not directly supported by service registries such as UDDI. An end-to-end solution for service versioning, which allows to transparently switch to the latest version of a service, is desirable. Dynamic service selection and binding should also be possible based on other non-functional attributes, such as the availability or response time.

In this chapter we address some of the issues and shortcomings that are prevalent in current SOA research and practice. We present *VRESCo*, the *Vienna Runtime Environment for Service-oriented Computing*. The discussion of the capabilities of the

VRESCo framework is based on an illustrative example scenario. The scenario contains a number of challenges for Service Oriented Computing (SOC). We identify and describe these issues and subsequently present appropriate solutions based on VRESCo. VRESCo has in part been developed within the FP7 Network of Excellence project S-Cube, within work package WP-JRA-2.3 (Self-* Service Infrastructure and Discovery Support). Within S-Cube, Vienna University of Technology is the beneficiary responsible for the development of the VRESCo prototype.

The remainder of this paper is structured as follows. In Section 11.2 we describe the example scenario that serves as the basis for the discussion of VRESCo. Section 11.3 comprises the main part of the paper, in which we describe the concepts and features of VRESCo. In this part, we always try to establish a link between the abstract concepts and the reference implementation of the scenario. In Section 11.4 we discuss existing work related to VRESCo. Since the VRESCo framework embraces a number of rather distinct research areas, the related work discussion can only focus on a few selected aspects. The paper concludes with a summary and final remarks in Section 11.5.

11.2 Example Scenario

To demonstrate the capabilities of the VRESCo framework, we consider an example scenario taken from the telecommunications domain. The scenario models a process that allows customers to port a phone number from their current CPO (cell phone operator), say *CPO1*, to another operator, say *CPO2*. The idea behind number porting is that customers are able to choose freely between providers of telecommunication services, without having to give up the phone number of their existing contract. When a customer decides to switch to provider CPO2, she signs the new contract and is temporarily assigned a new phone number (`TempNumber`). In the following, the customer (or, in fact, the new provider CPO2 acting on behalf of the customer) instructs her old provider, CPO1, to issue the number porting process and transfer the existing number (`DesiredNumber`) to provider CPO2.

Figure 1 illustrates the scenario process as well as the internal and external services that are involved. For the sake of brevity, error handling routines are not depicted in the figure. However, each process activity includes reasonable integrity checks concerning the input parameters and the current execution state. Input to the process are two phone numbers, the existing number with CPO1 (`DesiredNumber`), and the temporary number with CPO2 (`TempNumber`) and the time for which the number porting is scheduled.

The first step in the process is to contact a *CRM* (Customer Relationship Management) *Service* in order to look up the customer that is associated with `DesiredNumber`. Secondly, the partner CPO is determined dynamically. For that purpose, each CPO operating on the market publishes a *CPO Service* that can be used to query whether a certain number belongs to that CPO. The one CPO answering with a positive response is the partner for this process execution. Next, the *Number Porting*

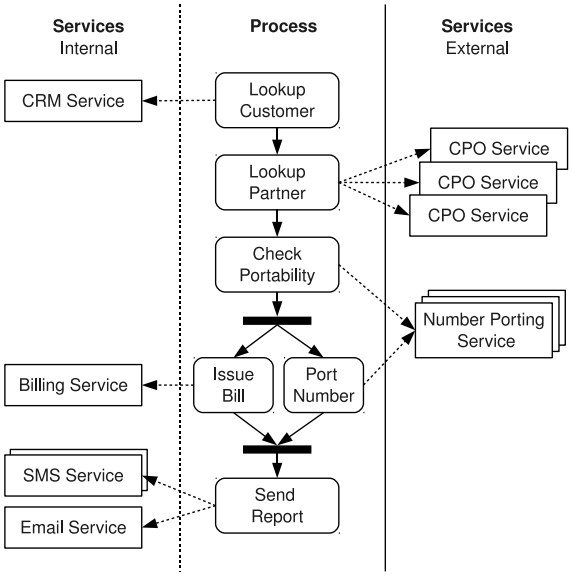

Fig. 11.1: Example scenario

Service of the partner CPO is invoked to check the portability status. In case porting is not possible at the time, appropriate measures need to be taken. Assuming that no error has occurred, the process can now execute two activities in parallel: performing the actual number porting with the partner *Number Porting Service* and issuing a bill using the internal *Billing Service*. As soon as both activities are finished, a report is sent to the customer via the internal *Email Service* and one instance of the (replicated) *SMS Service*.

11.2.1 Involved Web Services

Table 1 lists the Web services involved in the scenario, as well as the operations that are provided by these services.

Since the *CRM Service* is a core element not only in the number porting process, but in the IT architecture of the telecommunications provider as a whole, its requirements have changed in the past and will continue to evolve in the future. Not only the behavioral aspects change, but the renewed implementations often employ a different service interface. This raises the demand for explicit service *versioning* support.

For the *CPO Service* and the *Number Porting Service*, one instance (replica) exists for every CPO involved in the scenario. However, the replicas are not entirely identical: although they require the same (abstract) parameters and generate the same result, the concrete syntax (i.e., XML schema) of the exchanged messages is slightly different for each CPO.

Table 11.1: Web services involved in the example scenario

CRM Service			
Operation: GetCustomer	Input	Number:	`PhoneNumber`
	Output	MatchingCustomer:	`Customer`
CPO Service			
Operation: IsRegisteredNumber	Input	Number:	`PhoneNumber`
	Output	IsRegistered:	`boolean`
Number Porting Service			
Operation: CheckPortability	Input	TempNumber:	phone number
		DesiredNumber:	phone number
	Output	PortingPossible:	porting status
Operation: SchedulePorting	Input	TempNumber:	phone number
		DesiredNumber:	phone number
		ScheduledTime:	timestamp
	Output	Result:	porting result
Billing Service			
Operation: IssueBill	Input	TheCustomer:	`Customer`
		ServiceToBill:	`PortingResult`
Notification Services (Email Service, SMS Service)			
Operation: NotifyCustomer	Input	TheCustomer:	`Customer`

Consider the operation *SchedulePorting* of the *Number Porting Service*. The schemas of the different input messages for three operators (CPO1, CPO2, CPO3) are illustrated in Figure 11.2. The data contained in each of the messages is the existing *temporary number*, the *desired number* and the *time* that the number porting shall be scheduled for. For CPO2 and CPO3, the *country code* (e.g., *43* for Austria) and the *area code* (e.g., *686*) of the phone numbers need to be specified separately, while for CPO1 this information is included in the phone number string (e.g., *"0043 686 1234567"*). Since CPO2 expects only one `CountryCode` argument, it is expected that the temporary number and the desired number are from the same country. The scheduled time is expressed as a timestamp and is of type `string` for CPO1 and of type `long` for CPO2 and CPO3. Similarly, the `CheckPortability` operation of the three CPOs has slightly different message schemas, which will not be discussed in more detail for the sake of brevity.

The *SMS Service*, which is heavily used across the overall system, is replicated in several instances. In order to maximize throughput, the requests to this service should be distributed to the deployed replicas. Hence, the request for SMS notification at the end of the number porting process should be handled by the service instance which currently shows the best performance.

11.2.2 SOC Challenges

To sum up, the presented number porting scenario addresses the following SOC challenges:

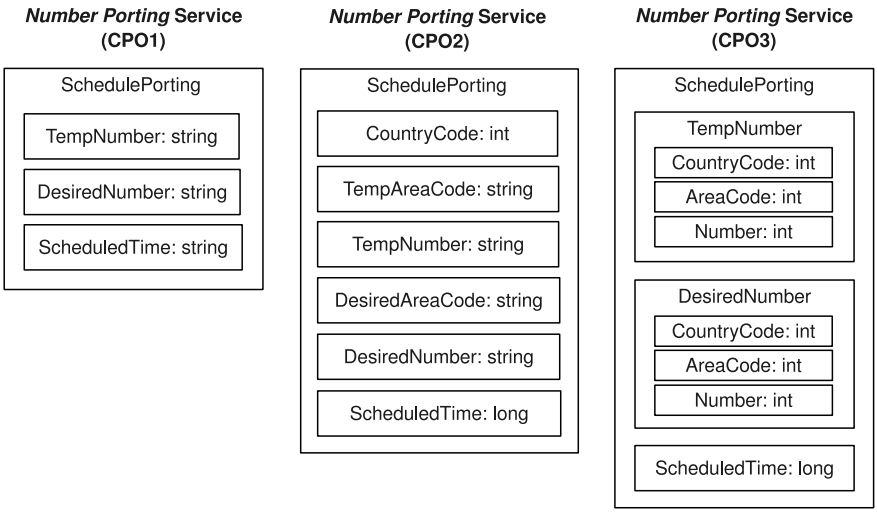

Fig. 11.2: Input messages of operation *SchedulePorting*

- **Service Metadata**: It is desirable to have a data model for describing the services that participate in the system. The description should cover both functional and non-functional characteristics. Functional attributes include the services' technical interface in the form of its operations and their input and output parameters. Non-functional characteristics concern *Quality of Service* (QoS) aspects such as the performance, security or price of a service. To account for services which perform similar or identical tasks, the description of the abstract functionality provided by a service should be separated from the definition of concrete endpoints. This is particularly required for the replicated versions of the *SMS Service* and the different versions of the *Number Porting Service*.
- **Service Versioning**: When services evolve over time, it is important that new versions of a service can be published and easily integrated into the existing process. Ideally, the integration should happen automatically and transparently, and it should be possible to either always bind to the latest version of a service or to explicitly switch back to older versions of a service. In the scenario, this is particularly important for the *CRM Service*, which exists in different versions and is subject to ongoing changes.
- **Service Interface Mediation**: Each CPO participating in the example scenario uses a slightly different input message schema for the operation `SchedulePorting` of the *Number Porting Service*. The demand for transparently exchanging endpoints of this operation results in the necessity for mediation between service interfaces. To that end, the commonalities of the different concrete interfaces provided by CPO1, CPO2 and CPO3 have been identified and combined in an abstract interface description of this operation. For instance, the abstract interface specifies that the `SchedulePorting` operation requires

two telephone numbers and one timestamp (see Table 11.1). Based on the abstract interface definition, it should be possible to define for each concrete implementation how the elements of the abstract interface can be mapped to the elements of the concrete interface. After defining the mapping at design time, the conversion should be conducted automatically at runtime.

- **Dynamic Service Invocation**: Since hardwired solutions of service invocations are inflexible and usually fail in case of even the smallest modification to the system (such as changing the value of an XML namespace or extending an XML schema with an additional element), it is advisable to reach a maximum level of dynamicity. Ideally, the message exchange should be expressed in an abstracted, high-level way and the actual invocations and message transformations should be performed by an underlying framework.
- **Service Rebinding**: Related to service versioning and dynamic invocation, rebinding denotes the capability of a client to bind to a new service or service interface. Rebinding may happen at different occasions – either in *periodic* intervals, *on demand* or *event-based*. For instance, the *CRM Service* is frequently changed and hence its clients' binding should be reconsidered periodically. Furthermore, the scenario is dynamic in the sense that the version of an existing *Number Porting Service* may change and that new partner CPOs may join the market or existing CPOs may cease to exist. As soon as any changes in the structure of the system arise, clients shall be notified of the modifications and rebind to the target service(s).
- **QoS-Based Service Selection**: The *SMS Service*, which is used to send a notification to the customer at the end of the number porting process, exists in several instances to serve the high demand for this functionality. To offer the clients a good performance of this service, requests should be distributed among all deployed replicas. The decision which replica to use is based on QoS (Quality of Service) characteristics, e.g. the availability of the service and the response time of the operation *NotifyCustomer*.

11.3 The VRESCo Solution

In the following we present the VRESCo framework, based on a reference implementation of the *number porting* scenario described in Section 11.1. This section is divided into several subsections, each of which addresses a particular aspect of the SOC challenges mentioned in Section 11.2. Furthermore, the VRESCo features are applied to the concrete solution of the example scenario.

11.3.1 System Overview

As has been mentioned in the introductory section of this chapter, the VRESCo platform is an effort to compensate issues and shortcomings in current SOA solutions. VRESCo strives to recover the "*broken*" SOA triangle [16] by focusing on the use of

service metadata published in a registry, as well as dynamic binding and invocation of services.

An overview of the VRESCo architecture is depicted in Figure 11.3. The system is implemented in C# and makes use of the *Windows Communication Foundation* (WCF) [11]. The *VRESCo Runtime Environment* is a server application that is invoked using the *VRESCo Client Library*. The interfaces of the server components are exposed as Web services and the communication between clients and the VRESCo runtime uses the SOAP [28] messaging protocol.

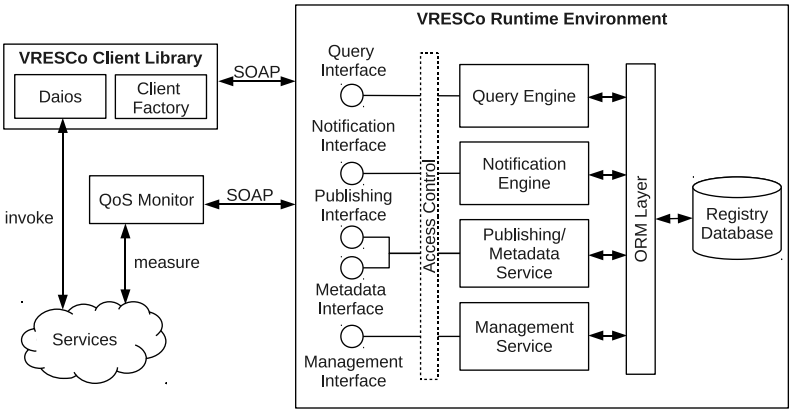

Fig. 11.3: VRESCo architecture overview, adapted from [15]

The *VRESCo Runtime Environment* is made up of different sub-components:

- The *Query Engine* allows to search for any entity that is stored within the runtime. To that end, a specialized query language (*VRESCo Query Language*, VQL) is offered, which will be further discussed in Section 11.3.3.
- With the aid of the *Notification Engine* clients are able to subscribe for notification of events that occur during execution of the runtime [13].
- The *Publishing/Metadata Service* offers two interfaces, the *Metadata Interface* for adding entries to the *metadata model* and the *Publishing Interface* for registering the description of a service implementation using the *service model*. The distinction between these to models will be clarified in Section 11.3.2.
- The *Management Service* is responsible for storing user information and handling the user access rights.

Additionally to the aforementioned parts, the VRESCo runtime employs a fifth core component, the *Composition Engine*. However, a discussion of the composition mechanism is out of the scope of this paper and the interested reader will find a detailed description in [22]. All communication is secured using an *Access Control*

layer that checks user credentials (username, password), handles encryption and applies signatures to the exchanged messages.

On the client side, Daios [9], a framework for dynamic and asynchronous service invocation, is used to conduct the message exchange with the involved Web services. Daios uses an abstracted message format, the *Daios Message*, which provides for protocol-independence and support for both SOAP and REST based services. The dynamic messaging approach will be briefly presented in Section 11.3.6. The *Client Factory* is used to instantiate proxy objects that communicate with the interfaces of the VRESCo Runtime Environment. Since these interfaces are exposed as Web services, SOAP is used as the messaging protocol between clients and the runtime.

Finally, since VRESCo allows for selection and composition of services based on *Quality of Service* (QoS) attributes, a *QoS Monitor* [14] is deployed as a standalone component, which regularly measures the availability, performance and accuracy of the target Web services. The QoS Monitor feeds back the acquired measurement values to the VRESCo runtime via the *Publishing Interface*. The runtime aggregates the stored measurements and calculates average values.

11.3.2 Metadata Model and Service Model

VRESCo uses a *metadata model* to describe functionalities offered by Web services in an abstract way. A *category* is a named entity that describes the general purpose of services and serves as an umbrella item for all other entities in that domain (e.g., the VRESCo category of our example scenario is named `TelcoSystem`). A category contains an arbitrary number of *features*, which describe a concrete action in the system. In the scenario, a feature named `PortNumber` describes the action of actually performing the porting with a target partner CPO. As illustrated in Figure 11.4, the base element in the model is the abstract *Concept*, which is specialized by the three sub-entities *Data Concept*, *Predicate* and *Feature*. All concept entities are composable, i.e., a concept can be derived from another concept. *Data concepts* define the data types that occur in the system and are either atomic (string, number, etc.) or composed of other data concepts. For instance, in the *number porting* scenario we make use of a data concept named `PhoneNumber`, which itself consists of three elements named `CountryCode`, `AreaCode` and `Number`.

Additionally, the metadata model allows for the definition of *Precondition*s and *Postcondition*s that need to be fulfilled when a feature is executed. Both types of conditions are associated with a number of *Predicate*s, which may each have multiple *Argument*s. The arguments are described by an associated data concept. Two types of predicates are distinguished: *Flow Predicate*s indicate constraints related to the data flow such as data required or produced by a feature (predicates `requires` and `produces`); *State Predicate*s express global constraints that need to be fulfilled. For a more detailed description of the metadata model we refer to [15].

The *service model* constitutes the information about concrete services managed by VRESCo. A *service* is available in one or more *revisions*. *Revisions* are the basis for

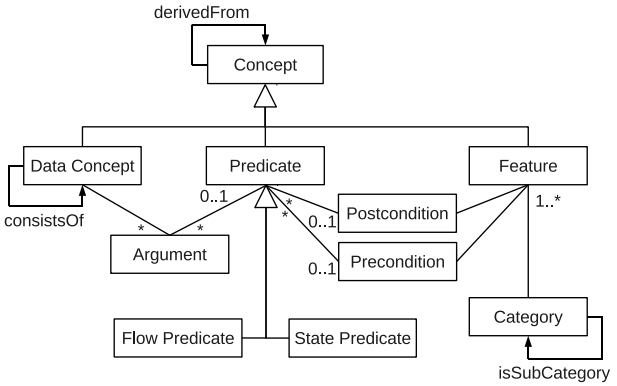

Fig. 11.4: Service metadata model, adapted from [15]

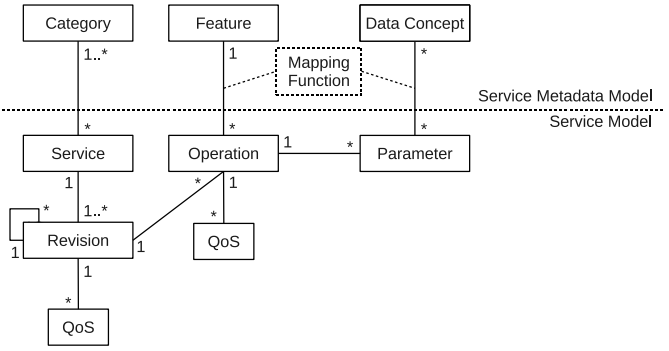

Fig. 11.5: Mapping between service model and metadata model, adapted from [15]

service versioning, which will be discussed in more detail in Section 11.3.4. Figure 11.5 depicts the mapping between the service metadata model and the (concrete) service model. *Services* are mapped to *categories*, a service *operation* is one concrete implementation of a *feature* (note that one *feature* may be implemented by several different *operations*) and operation *parameters* are mapped using *data concepts*. The *mapping function* between the latter is the basis for the VRESCo service mediation, which will be further discussed in Section 11.3.7.

In addition to the functional characteristics of services, VRESCo also saves non-functional attributes in the form of *QoS* attributes. Operation-specific QoS data are attributes such as *response time* or *accuracy*, whereas attributes such as *latency* apply to a service revision. A more detailed description of QoS attributes is given in Section 11.3.5.

11.3.3 Service Querying

The *VRESCo Querying Language* (VQL) constitutes an interface to query the data stored in the registry. VQL queries are based on the entities and relations of the data model presented in Section 11.3.2. In that sense, the query language abstracts from the concrete database schema that is used to store the model entities. Following the Query Object Pattern [6], VQL provides an API to programmatically construct queries from a set of *criteria*. Criteria are defined either as *mandatory* using the function `Add` or as *optional* using the function `Match`. Mandatory criteria must be fulfilled for all elements of the result set that is obtained by executing the query, whereas optional criteria are treated differently depending on the *querying strategy*:

- The *EXACT* querying strategy treats all criteria as mandatory, regardless of whether they are `Add` or `Match` criteria.
- With *PRIORITY* querying, each `Match` criterion is prioritized using a numeric *weight* attribute. The higher its weight, the more importance a criterion receives during the selection process of the query engine. The final result is sorted by the sum of priority values. For instance, a `Match` criterion with priority weight 4 takes precedence over the sum of two criteria with weights 1 and 2, because $4 > (1+2)$.
- The *RELAXED* strategy is a specialized case of *PRIORITY* querying, where the priority weight of each `Match` criterion is 1. This method simply distinguishes between mandatory and optional parameters and sorts the final result list based on the number of optional (`Match`) criteria that are fulfilled.

Upon execution, VQL query objects are internally converted to an according SQL query string and finally executed on the underlying database.

```
1  var querier = VRESCoClientFactory.CreateQuerier("username", "password");
2  var query = new VQuery(typeof(Service));
3  query.Add(Expression.Eq("Category.Features.Name", "PortNumber"));
4  query.Add(Expression.Eq("Revisions.IsActive", "true"));
5  var list =
6      querier.FindByQuery(query,QueryMode.Exact) as IList<ServiceRevision>;
7
8  // create client proxy for each Number Porting Service
9  foreach(var service in list) {
10     query = new VQuery(typeof(ServiceRevision));
11     query.Add(Expression.Eq("Service.ID", service.ID));
12     query.Add(Expression.Eq("Tags.Property.Name", "LATEST"));
13     DaiosProxy proxy = querier.CreateRebindingProxy(query,
14         QueryMode.Exact, 1, new PeriodicRebindingStrategy(1000*60*60));
15     ...
16 }
```

Listing 11.1: VQL Example Query Using EXACT Strategy

Listing 11.1 illustrates an initialization routine of the number porting scenario which obtains the references to the *Number Porting Service*s of the involved CPOs and creates a client for each instance. Firstly, a *Querier* instance is created with the aid of the VRESCo client factory. For the creation of the `VQuery` object in line 2, a

parameter is used to specify the expected return type of the result (`Service`). Lines 3 and 4 add criteria to the query, one related to the name of the feature the service implements (needs to contain the string *"PortNumber"*) and the second mandating that at least one revision of this service needs to be active. The query uses an *EXACT* query strategy and hence treats all criteria as mandatory. In lines 5 and 6 of the code listing, the query is executed and the result is received as a list of services. Subsequently, the code loops over all result entities and creates a new `VQuery` that expects `ServiceRevision` results (line 10). The criterion in line 11 is used to have the service identifier (`ID`) of the result match the `ID` of the service in the current loop execution. Line 12 specifies that only the latest version of a service should be returned (more details on modeling service versions with VRESCo is given in Section 11.3.4). Finally, the constructed query object serves as the basis for creating a `DaoisProxy` that is utilized to perform the dynamic service invocations later on (see Section 11.3.6). More sophisticated VQL examples that include alternative querying strategies and `Match` criteria will follow in the remaining parts of this chapter.

11.3.4 Service Versioning

Just as any other IT artifact may evolve over time, Web services are also subject to changes made by the service providers. For example, services may be extended by new functionalities, adapted to changed environments or restructured to better fit the overall IT architecture. The process of services being modified over time is collectively referred to as service *evolution*. When a service evolves, usually the older versions of this service are still available in order not to break the operability of existing clients. Hence it is an important requirement for service registries to handle service evolution and to document the changes of service interfaces. Current registries such as UDDI [19] do not directly support different versions of the same service.

The distinction between *services* and *revisions* in the VRESCo service model allows for explicit versioning of services [8]. The service model defines *tags*, which describe a revision. VRESCo defines six default revision tags, while custom tags can also be added by service providers. Table 11.2 lists the default tags, their meaning and who they are assigned by. The tags `INITIAL`, `HEAD` and `LATEST` are automatically assigned by the VRESCo runtime, whereas all other tags are added manually by the provider.

Relationships between service versions can be visualized as a *service version graph*, where the nodes constitute service revisions and directed edges point from a revision node to all its successor revisions. Figure 11.6 depicts the relation between the revisions of the *CRM Service*. The first version (*v1*) is tagged `INITIAL`. For the second version, the `Customer` data type has been extended by an additional field and the implementation has been thoroughly tested such that *v2* deserves the tag `STABLE`. Version 3 has been implemented as a WCF service [11] – making use of the integrated security and encryption framework – and is therefore tagged *wcf*. Ver-

Table 11.2: Predefined service revision tags

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

sion 4, on the other hand, is an unsecured version of the CRM Service, realized with JAX-WS [1]. Finally, revision *v5* represents an extension of the WCF implementation. This is the last revision in the WCF branch and also the most recent version in the version graph.

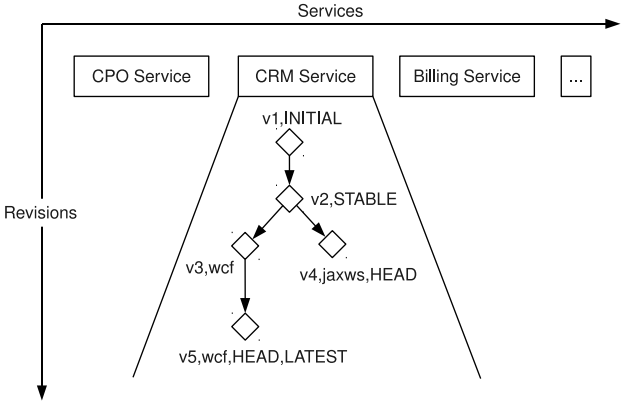

Fig. 11.6: *CRM Service* version graph

11.3.5 QoS-Based Service Selection

As mentioned in the *number porting* scenario description (see Section 11.2), the *SMS Service*, which is used to send a notification to the customer at the end of the number porting process, is replicated (i.e., exists in several instances) to serve the high demand for this functionality. Ideally, the requests should be equally distributed to all replicas to avoid the situation that one service instance becomes overloaded.

The VRESCo runtime stores QoS data that apply to both service *revisions* and *operations*. The QoS information may be either added manually using the *Management Service* or measured automatically with the aid of the *QoS Monitor*. VRESCo cur-

[1] `https://jax-ws.dev.java.net/`

rently defines the QoS attributes that are listed in Table 11.3 [15]. While *Price*, *Reliable Messaging* and *Security* are static values that are manually entered by service providers, the remaining attributes are *calculated* by the QoS Monitor. *Latency* denotes the (average) time required to transport a request over the network. *Response Time* is the sum of latency for request and response plus the execution duration of the service operation. *Availability* is the time during which the service is up and running,

Table 11.3: Predefined QoS attributes

Attribute	Type	Unit	Applies To
Price	static	$ / invocation	Revision, Operation
Reliable Messaging	static	{true, false}	Revision
Security	static	{None, X.509}	Revision, Operation
Latency	calculated	ms	Revision
Response Time	calculated	ms	Operation
Availability	calculated	percent	Revision
Accuracy	calculated	percent	Revision, Operation
Throughput	calculated	invocations / sec	Revision, Operation

expressed as the percentage of the total time. The probability that a service produces a valid result is the *Accuracy*, which is calculated as the ratio of successful requests to total requests. Finally, *Throughput* is the number of requests a service can process during a given period of time. The QoS model is extensible and enables service providers to define and publish custom QoS data in addition to the aforementioned QoS attributes.

Since the QoS entities are part of the service model, QoS-related criteria can be included in VQL queries. Listing 11.2 shows the VRESCo implementation of selecting the "most suitable" *SMS Service* instance. Most suitable in this context means that the service should have

1. most importantly, a high availability ($> 90\%$),
2. the smallest possible response time,
    a. less than 100 ms (priority weight 3), or
    b. less than 200 ms (priority weight 2), or
    c. less than 500 ms (priority weight 1);
3. if possible, an accuracy of at least 80%.

We assume that the combination of points 2 and 3 has the same weight as point 1. Listing 11.2 displays the VQL query construction necessary to achieve this specific service selection. The `Add` criterion in line 2 mandates that the service name must include the String *"SMSService"*. The `Match` operation in the following two lines specifies that with a priority weight of 5 the availability should be greater than 0.9 (90%). Note that the '`&`' operator is a shortcut for an `And` operation. With the aid of the following 3 statements, a response time of less than 100, 200 or 500 ms is targeted, with a priority weight of 1 each. Therefore, in sum, the priority weight of

a response time of less than 100 ms is 3 and the priority weight of a response time of less than 200 ms is 2.

```
var query = new VQuery(typeof(ServiceRevision));
query.Add(Expression.Like("Service.Name", "%SMSService%"));
query.Match(Expression.Eq("QoS.Property.Name", "Availability") &
            Expression.Gt("QoS.DoubleValue", 0.9), 5);
query.Match(Expression.Eq("QoS.Property.Name", "ResponseTime") &
            Expression.Lt("QoS.DoubleValue", 100.0), 1);
query.Match(Expression.Eq("QoS.Property.Name", "ResponseTime") &
            Expression.Lt("QoS.DoubleValue", 200.0), 1);
query.Match(Expression.Eq("QoS.Property.Name", "ResponseTime") &
            Expression.Lt("QoS.DoubleValue", 500.0), 1);
query.Match(Expression.Eq("QoS.Property.Name", "Accuracy") &
            Expression.Ge("QoS.DoubleValue", 0.8), 2);

var list =
    querier.FindByQuery(query,QueryMode.Priority) as IList<ServiceRevision>;
var bestServiceInstance = list[0];
... // perform SMS notification using bestServiceInstance
```

Listing 11.2: VQL Query Using QoS Attributes

Assume that five *SMS Service* instances are deployed in the example scenario. A illustrative snapshot of QoS values at a certain point in time is given in Table 11.4. Values that fulfill the QoS criteria named above are printed in bold text. *Total Score* is the sum of priority weights of those QoS criteria that are fulfilled by a service. *SMS Service 1* is ranked first since it has the highest total score of 8 points. Services 2 and 5 are ranked second with 7 points each, and the services 4 (5 points) and 3 (4 points) take positions 3 and 4, respectively.

Table 11.4: Snapshot of QoS characteristics of SMS service instances

Service	Availability	ResponseTime	Accuracy	Total Score	Rank
SMS Service 1	**0.91**	**413 ms**	**0.80**	5 + 1 + 2 = 8	1
SMS Service 2	**0.95**	**194 ms**	0.78	5 + 2 + 0 = 7	2
SMS Service 3	0.81	**156 ms**	**1.00**	0 + 2 + 2 = 4	4
SMS Service 4	0.85	**96 ms**	**0.85**	0 + 3 + 2 = 5	3
SMS Service 5	**0.97**	527 ms	**0.93**	5 + 0 + 2 = 7	2

The VRESCo query engine internally performs the same calculation: for each `Match` criterion *m* in a query with *PRIORITY* (or *RELAXED*) query strategy, it determines for each element *e* in the universal set (in this case, all `ServiceRevision` entities whose name contains the string *"SMSService"*) whether *e* fulfills criterion *m*, and adds the priority value of *m* to the score of *e*. As mentioned in Section 11.3.3, this calculation is actually performed by constructing a corresponding SQL statement, which is executed on the underlying DBMS. The final result is ordered by decreasing score of the elements. Hence, after execution of the query, the variable `list` in Listing 11.2 contains the service revision objects in the order of their QoS score. The first element of the list is assigned to the variable

`bestServiceInstance`, a reference to *SMS Service 1*, which is subsequently used to perform the SMS notification.

11.3.6 Dynamic Service Invocation

Web Service invocations in VRESCo are executed using the *Daios* framework [9]. In contrast to many other Web service client frameworks that rely on code generation and client-side stubs to invoke services (such as Apache Axis 2 [2]), Daios seeks to provide *stub-less* communication with Web services. Furthermore, the Daios framework is *protocol-independent* in the sense that it transparently distinguishes and handles SOAP and REST invocations. This is achieved by using an abstracted *Daios message* format, which describes request and response messages in a high-level way, leaving protocol-specific details to the lower, internal layers of Daios. In the following, we briefly present the dynamic binding capability of VRESCo and discuss the message-centric invocation approach of Daios.

11.3.6.1 Dynamic Binding

The ability of clients to dynamically bind to services is often claimed to be one of the key advantages of SOA. However, in practice binding often happens at design time using generated stubs. Daios overcomes this issue and provides a service invocation framework that is build upon the dynamic invocation principle. The process of binding to a service involves several steps. Firstly, the client needs to retrieve and parse the service interface description, or *service contract*. WSDL is the standard description language for Web services that use SOAP [28] as the messaging protocol, whereas WADL [24] (Web Application Description Language) is often used to describe REST-based services. Typically both WSDL and WADL parsing involves processing XML schema definitions (XSD) that define the data types of the exchanged documents. Next, a service *proxy* is created from the in-memory representation of the service contract. When the client issues an invocation, the provided input is matched with the message definitions contained in the service contract.

The frequency and occasion at which binding takes place determine the accuracy of the proxy, but also influence the runtime performance since rebinding causes a certain overhead. VRESCo distinguishes the following rebinding strategies:

- `Fixed`: Fixed proxies perform binding upon creation but never rebind to the target service. They are used in scenarios where rebinding is not required.
- `Periodic`: With this strategy, the proxy reconsiders the binding periodically in fixed intervals, which is inefficient if the frequency of invocations is low.
- `OnInvocation`: This strategy causes the proxy to update its binding prior to every service invocation. This ensures that the binding is always up to date but obviously results in a large overhead.

[2] `http://ws.apache.org/axis2/`

- `OnDemand`: If rebinding happens upon client request, the overhead is reduced in comparison to `Periodic` and `OnInvocation`, with the drawback that the binding is not always up to date.
- `OnEvent`: This rebinding strategy combines the advantages of all strategies by making use of the VRESCo *Event Notification Engine* [13]. Clients enter subscriptions that define in which situations rebinding should take place. If rebinding is due, the VRESCo runtime triggers a corresponding event notification. A disadvantage of this strategy is that clients need to expose a callback service in the network.

The implementation of the number porting scenario makes use of different rebinding strategies. The proxy for the *CRM Service* is updated periodically every hour (`Periodic`), since this service is subject to frequent changes. A `Periodic` strategy is also applied for the *Number Porting Service*. Because the *Email Service* and the *Billing Service* are very stable and have not evolved over the last years, the message exchange with these services is conducted using a `Fixed` client. In order to select the best-performing instance of the *SMS Service* in each execution of the number porting process, an `OnInvocation` proxy is used for invoking the `Notify` operation. A thorough performance evaluation of the different rebinding strategies has been carried out in [15].

11.3.6.2 Message-Centric Communication

Daios follows a message-oriented approach, i.e., client developers do not "invoke operations" but send and receive messages to and from the service. Daios uses a special message format which abstracts messages from their XML representations. The Daios message represents objects as a collection of name/value pairs. The value of one such pair entry may be either 1) a simple type (string, integer, ...), 2) an array of simple types or 3) a message or an array of messages (recursive construction).

```
1  DaiosProxy proxy = ...; // get proxy for service of partner CPO
2
3  var request = new DaiosMessage();
4  var tempNumber = new DaiosMessage();
5  tempNumber.Set("CountryCode", 43);
6  tempNumber.Set("AreaCode", 686);
7  tempNumber.Set("Number", 1234567);
8  var desiredNumber = new DaiosMessage();
9  desiredNumber.Set("CountryCode", 43);
10 desiredNumber.Set("AreaCode", 677);
11 desiredNumber.Set("Number", 87654321);
12 request.Set("TempNumber", tempNumber);
13 request.Set("DesiredNumber", desiredNumber);
14 request.Set("ScheduledTime", 1267124400);
15
16 DaiosMessage response = proxy.RequestResponse(request);
```

Listing 11.3: Construction of Daios Messages

Listing 11.3 shows how a request to the operation *SchedulePorting* of the *Number Porting Service* is constructed using Daios. The service selection and proxy cre-

ation has been illustrated in Listing 11.1 in Section 11.3.3. Now we assume that the process execution has just determined the partner CPO in the activity *Lookup Partner* and that in line 1 of Listing 11.3 we obtain a reference to the Daios proxy for this partner. To construct the desired request message, three Daios messages are instantiated: a "container" Daios message named `request` and two message instances `tempNumber` and `desiredNumber`. In lines 5-7 and 9-11, respectively, the simple values `CountryCode`, `AreaCode` and `Number` are set. The recursive construction of Daios messages is illustrated in lines 12 and 13.

Once the DaiosMessage has been constructed, the `DaiosProxy` instance is used to perform a synchronous service invocation (see line 15 in Listing 11.3). Note that this proxy follows a `Periodic` rebinding strategy and reconsiders its binding every 60 minutes in order to always bind to the most recent version of the service (see Listing 11.1). Besides the "request response" invocation flavor, Daios also supports asynchronous communication ("fire and forget", "poll object" and "callback"). The achievement of the Daios framework is that all information necessary to construct the final SOAP message (qualified operation names, XML namespaces, etc.) is transparently collected from the target service's WSDL document in the background. Analogously, the SOAP response from the service is internally converted back to the high-level Daios message format. In Section 11.3.7 we will discuss the internals of service invocations in VRESCo and how transformations of input and output messages are performed using a *Mediator Chain*.

11.3.7 Service Mediation

The *Number Porting Service* instances of the three different CPOs participating in the example scenario have the same logical interface in terms of which information needs to be provided by a calling endpoint, but the services differ in their syntactical interface (i.e., the schema of the input messages). In VRESCo terms, the three services provide the same (abstract) *features*, but implement a different concrete *operation* interface. Hence, if service endpoints are to be exchanged transparently, there is a need to map the interfaces to one another. This mapping can be easily achieved using the *VRESCo Mapping Framework* (VMF), which provides capabilities to map between input and output parameters of features and operations stored in the registry.

The VMF architecture is illustrated in Figure 11.7. At design time, or *mapping time*, service providers use the *Mapper* component to enter the mapping information into the mapping database via the *Metadata Service*. The mapping information comprises 1) the (interfaces of the) abstract *features* that are available, 2) the (interfaces of the) *operations* a service offers, 3) which operation implements which feature and 4) how operation interface and feature interface are mapped to one another.

Having published the mapping functions for the operations of all three CPOs, the endpoints for feature `PortNumber` can be dynamically exchanged at *execution time*. Clients simply provide their input in the format that is dictated by the feature's interface. The VRESCo client library then transparently converts the input such that

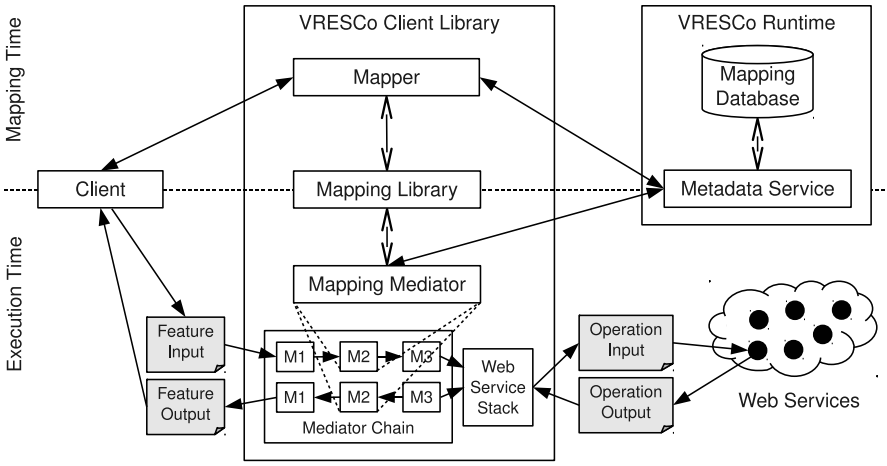

Fig. 11.7: VMF architecture

it matches the interface of the target operation. To that end, *Mapping Mediator*s are plugged in to the client library as a part of the *Mediator Chain*. Each mediator that is part of this chain may perform modifications to the incoming and outgoing messages. Converting a high-level feature input or output message to its equivalent message on operation level is referred to as *lowering*, whereas the opposite is called *lifting*. The *Mapping Mediator* retrieves the lowering and lifting information from the *Metadata Service*. When outgoing messages are handed through the mediator chain, the *Mapping Mediator* performs the lowering before the message is finally serialized to SOAP and sent via the *Web Service Stack*. Incoming messages are firstly deserialized and travel back through the mediator chain in the opposite direction.

The concrete interface mediation concerning the service operation *SchedulePorting* is depicted in Figure 11.8. The *feature* `PortNumber` is stored in the VRESCo metadata model. The metadata model defines that the feature expects the input data *TempNumber*, *DesiredNumber* and *ScheduledTime*. The type of the parameters is defined in the model as *data concepts*: in the case of *TempNumber* and *DesiredNumber* the type is *PhoneNumber* (which contains sub-elements *CountryCode*, *AreaCode* and *Number*), and the type of *ScheduledTime* is *long*. The telecommunication operators CPO1, CPO2 and CPO3 provide a *SchedulePorting* operation, each of which has a slightly different interface (see Section 11.2.1). To create the mapping between the *feature* parameters and the *operations'* parameters, VMF provides a number of *Mapping Functions*. Mapping Functions may be defined for both input and output parameters. Currently, the following Mapping Functions are supported:

- **Assign** functions link one parameter to another parameter of the same data type.
- Parameters may be assigned **constants** of simple data types.

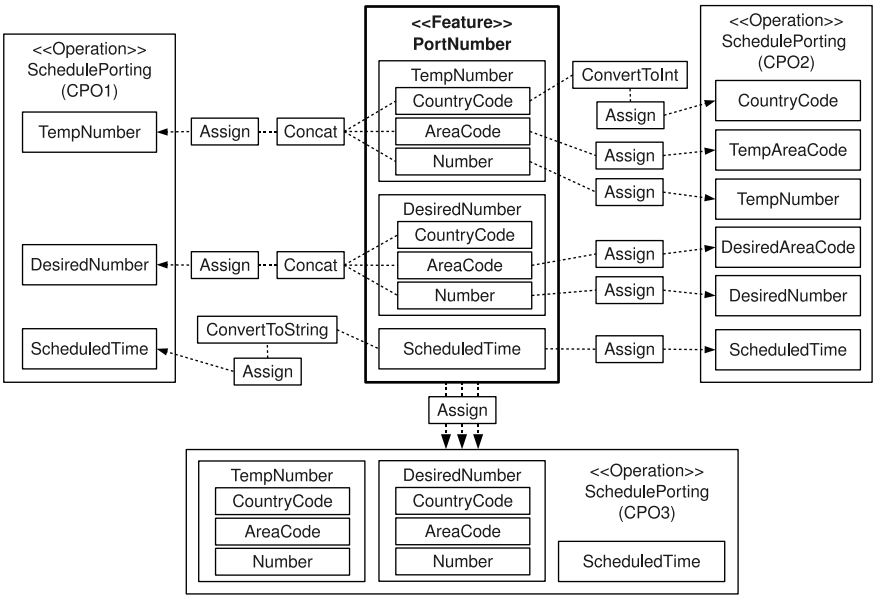

Fig. 11.8: Interface mapping for *SchedulePorting* operations

- Simple data types may be **converted** to other simple types. For example, *convertToInt* converts a given value to an *integer* (if possible) and the *convertToString* function creates a string representation of its input value.
- The set of **array** functions provides for creation of arrays and the access to elements at certain positions.
- Functions such as *Concat* or *Substring* allow for **string manipulation**.
- **Mathematical** operations are supported for *numeric* and *boolean* data types.
- Additionally, more **complex mappings** may be defined directly in C# code. The mapping code is stored in VRESCo as plain text and gets executed upon request using *CS-Script* [3], a scripting engine for C#.

Listing 11.4 contains a code excerpt that shows how mapping functions can be added to the VRESCo runtime. In the listing, the mapping between the service operation *SchedulePorting* of CPO1 and the feature `PortNumber` is implemented. In line 4, a `Mapper` object is created that mediates between the feature and the operation. The mapper allows direct access to the input (and output) parameters of both the feature and the operation. In lines 6 and 7, the parameters are stored to the variables `fInput` and `oInput`. Starting from line 9, the actual mapping functions are constructed. As indicated in Figure 11.8, the mapping functions are applied in a chain, where the result of one function becomes the input of another function. In this example, the "elementary" mappings are two `Concat` (string concatenation) functions

[3] `http://www.csscript.net/`

and one `ConvertToString` function (compare lines 9-22). The results of these functions serve as input to an `Assign` function for each of the three parameters of the operation *SchedulePorting* of CPO1.

```
1  var metaPubl = VReSCOClientFactory.CreateMetaDataPublisher("user", "pass");
2  var feature = ...;        // get 'PortNumber' feature
3  var operation = ...;      // get 'SchedulePorting' operation of CPO1
4  var mapper = metaPubl.createMapper(feature, operation);
5
6  IList<MappingElement> fInput = mapper.FeatureInputParameters;
7  IList<MappingElement> oInput = mapper.OperationInputParameters;
8
9  var concat1 = new Concat();
10 concat1.AddInputElement(fInput[0].Children[0]);
11 concat1.AddInputElement(fInput[0].Children[1]);
12 concat1.AddInputElement(fInput[0].Children[2]);
13 concat1 = mapper.AddFeatureToOperationFunction(concat1);
14
15 var concat2 = new Concat();
16 concat2.AddInputElement(fInput[1].Children[0]);
17 concat2.AddInputElement(fInput[1].Children[1]);
18 concat2.AddInputElement(fInput[1].Children[2]);
19 concat2 = mapper.AddFeatureToOperationFunction(concat2);
20
21 var toString = new ConvertToString(fInput[2]);
22 toString = mapper.AddFeatureToOperationFunction(toString);
23
24 mapper.AddFeatureToOperationFunction(new Assign(concat1.Result,oInput[0]));
25 mapper.AddFeatureToOperationFunction(new Assign(concat2.Result,oInput[1]));
26 mapper.AddFeatureToOperationFunction(new Assign(toString.Result,oInput[2]));
27
28 metaPubl.AddMapping(mapper.GetMapping());
```

Listing 11.4: Creating the Mapping for Operation *SchedulePorting* of CPO1

Finally, the metadata publisher is used to store the constructed mapping instructions in the VRESCo runtime. Registering the mapping for operation *SchedulePorting* of the remaining operators CPO2 and CPO3 works analogously. CPO2 requires only one parameter to be converted to *integer* using the function `ConvertToInt`, the remaining input message parts can be simply copied using the `Assign` function. The mapping for CPO3 is straight-forward since its operation *SchedulePorting* and the `PortNumber` feature are identical in their interface.

11.4 Related Work

In this section we point to existing work that is related to VRESCo. We focus our discussion on the areas of service registries and service metadata, as well as service mediation and dynamic invocation.

In the area of Web service registries, a number of approaches and standards exist. UDDI [19], which was originally proposed as a core Web service standard, models characteristics of services (in the form of `businessService`, `bindingTemplate` and `tModel`) as well as identities of service providers (`businessEntity` contains metadata about a publisher and `publisherAssertion` de-

scribes relations between parties). The technical model (`tModel`) of UDDI supports mainly unstructured data, whereas the VRESCo data model is well-defined and distinguishes between the *metadata model* and the *service model*. UDDI had very limited success and was never fully adopted by the industry, a claim which is supported by the fact that public UDDI registries of Microsoft, SAP and IBM were eventually shut down in 2005. The set of specifications collectively described as *ebXML* (*Electronic Business using XML*) [4] enables enterprises to conduct electronic business over the Internet. Amongst other concepts, ebXML defines a *Registry Information Model* [17] and a *Registry Services* [18] standard. Similar to UDDI, the ebXML data model is rather unstructured, reducing the service description to a collection of links to its technical specification, such as the WSDL document. Furthermore, no distinction is made between abstract service features and concrete service implementations or instances. IBM's *WebSphere Service Registry and Repository* (WSRR) [7] uses a more structured information model, with the ability to automatically generate model entities (called *logical derivations*) from *physical documents* of well-known formats such as WSDL, XSD or WS-Policy. As opposed to VRESCo, WSRR has limited support for metadata versioning in the sense that logical derivations are not manipulable and hence not versionable.

Table 11.5: Comparison of related registry approaches

Category		**UDDI**	**ebXML**	**WebSphere**	**VRESCo**
Service Metadata:	Unstructured	✓	✓	✓	∼
	Structured	∼	∼	✓	✓
Service Querying:	Query Language / API	✓	✓	✓	✓
	Type-Safe Queries	×	×	∼	✓
Service Versioning:	Service Model Versioning	×	✓	✓	✓
	Metadata Versioning	×	✓	∼	×
	End-to-End Support	∼	∼	×	✓
Quality of Service:	Explicit QoS Support	×	×	∼	✓
	QoS Monitoring	×	×	×	✓
Dynamic Service Invocation:	Binding & Invocation	×	×	∼	✓
	Mediation	×	×	✓	✓

The results of the comparison between VRESCo and related approaches are summarized in Table 11.5. Of all discussed approaches, VRESCo is the only framework that provides end-to-end versioning support, which allows to seamlessly rebind and invoke different service revisions are runtime. All mentioned registries allow for querying using a specialized Query language or an API that operates on the datamodel entities, respectively. However, type-safe queries are not supported by most approaches since usually querying is performed on the underlying unstructured model using SQL or the like. Explicit support for non-functional, QoS-related service metadata is not available with UDDI or ebXML. WebSphere offers basic QoS support by means of integration of WS-Policy documents as well as user-defined

[4] `http://www.ebxml.org/`

classifications metadata. However, this approach is limited to static attributes such as security, price or reliable messaging, and leaves out important performance-related characteristics such as the availability of a service or the response time of an operation. In addition to UDDI and ebXML, a number of other standards and approaches in the area of service metadata [2] exist. For instance, OWL-S [26] is an ontology for describing semantic Web services [12] with the aim of automatic Web service discovery, invocation and composition. SAWSDL [27] attempts to include semantic annotations directly into WSDL documents, which describe the service interface. SAWSDL enriches WSDL by semantic models on XML schema level as well as interface and operation level. SAWSDL also uses *lifting* and *lowering* of messages using XSLT transformation. However, it must be emphasized that the VRESCo metadata model addresses enterprise scenarios where metadata is considered a valuable business asset and does therefore not compete with semantic Web approaches, which aim at disclosing all metadata. Furthermore, VRESCo logically and physically separates the service description (e.g., in the form of WSDL) from the metadata description. Related to metadata description is the problem of runtime querying for services that match certain requirements. The authors of [30] present a Web service query algebra based on a formal model of service and operation graphs. The model also includes QoS parameters and allows for defining constraints in the form of inter-service and intra-service dependencies. The query language uses algebraic operators on service functionalities (*functional map* operator), QoS (*quality select* operator) and compositions (*compose* operator). Upon execution of a query, optimization techniques are applied to select the best service execution plan. This work is different to VRESCo since it focuses on the formal service model and the query algebra as well as its optimizations, whereas VRESCo constitutes a runtime for service management and querying. The API of VQL queries is based on the Query Object pattern [6] rather than on formal predicates and clauses. Furthermore, optimizations are left to the underlying database management system.

Obviously, pure registries such as UDDI and ebXML do not support dynamic binding, mediation and invocation. WebSphere supports service mediation using *mediation streams*, a sequence of processing steps that are executed when an input message arrives. Inside mediation streams, a set of mediation primitives can be used to change the format and content of messages. In contrast to VRESCo, mediation streams are general purpose interceptors that are also used to log messages, perform database lookups and so forth, whereas mediation in VRESCo is tailored to define a mapping between abstract features and concrete operations. Of the previously discussed frameworks, integration of dynamic binding and invocation is only implemented by VRESCo. WebSphere does provide the prerequisites for dynamic endpoint selection and for dynamically exchanging the message transport protocol (e.g., HTTP, JMS), but the degree of abstraction does not reach the same level as with the Daios messaging approach, where even the messaging protocol itself (e.g., SOAP, REST) can be transparently switched. Dynamic binding has also been addressed by other approaches. For example, Di Penta et al. [4] present WS-Binder, a framework that enable dynamic binding of services within WS-BPEL [23] processes. In their approach, proxies are used to separate abstract services and con-

crete instances. Similarly, the autors of [21] present a solution to model bindings in the JOpera system using reflection. In contrast to VRESCo, both frameworks focus mainly on service composition environments.

In the area of service mediation, most approaches to resolving interface incompatibilities use an adapter-based solution [3, 10]. Whereas these adapters are principially similar to the mediators in VRESCo, they are more decoupled from the clients. The mediators approach allows to easily integrate existing domain knowledge, also in the form of more complex mediation concepts (such as mediation based on semantic service metadata), which is not always simple to achieve using adapters. Besides syntactic mediation of service messages [25], other related work focuses on mediation on business protocol level. The authors of [1] identify and characterize different interoperability patterns of business-level interfaces and protocols, and propose possible solutions. In [5], a set of operators is defined to tackle the problem of behavioral service interface adaptation. The behavioral interface is seen as *"a collection of control dependencies defined over a set of message exchanges"*. The paper also provides a graphical notation for the interface transformation algebra that is put forward. In contrast to the mentioned approaches, the VRESCo runtime mediates structural differences of service interfaces, but does not consider mediation on message exchange level.

11.5 Conclusion

One of the key promises of SOC is that it provides for loosely coupled distributed applications based on the publish-bind-find-execute cycle. However, SOC practice often falls short of keeping this promise due to the lack of service metadata, service querying possibilities, explicit QoS support, and solutions for dynamic binding and interface mediation. In this paper we have described *VRESCo*, the *Vienna Runtime Environment for Service-oriented Computing*, which provides a solution for some issues and shortcomings that are prevalent in current SOA research and practice. The challenges addressed by VRESCo have been identified and illustrated based on a *number porting* example scenario. First and foremost, in an attempt to "recover the broken SOA triangle" [16], VRESCo constitutes a service *registry* that is used by providers to store information about services in a (meta-)data model. The distinction between abstract *features* (*metadata model*) and concrete service implementations (*service model*) allows to group service instances that provide an identical functionality. VRESCo enables clients to query the stored information using the specialized query language VQL in order to dynamically select an endpoint for their invocations. The selection may be based on *functional* criteria which concern the interface (or service contract), but also on *non-functional* criteria in the form of *QoS* attributes. The Daios framework employs an abstracted message format and is used in VRESCo to realize *dynamic* and protocol-indendent *invocations*. The VRESCo data model supports explicit *versioning* of service revisions, operations and parameters. User-defined and default *tags* describe the features of a service revision and its position in the *version graph*. With the aid of mapping functions, the VRESCo

runtime mediates between service instances which perform the same task but differ in their technical interface. The VRESCo framework covers several distinct aspects of SOC practice and research. The result of our related work review is that similar solutions in each partial research direction exist, but that VRESCo constitutes a unique combination of service computing concepts and techniques.

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).

References

1. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M., Toumani, F.: Developing Adapters for Web Services Integration. Advanced Information Systems Engineering **3520/2005**, 415–429 (2005)
2. Bodoff, D., Ben-Menachem, M., Hung, P.C.K.: Web Metadata Standards: Observations and Prescriptions. IEEE Softw. **22**(1), 78–85 (2005). DOI http://dx.doi.org/10.1109/MS.2005.25
3. Cavallaro, L., Di Nitto, E.: An approach to adapt service requests to actual service interfaces. In: SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems, pp. 129–136. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1370018.1370041
4. Di Penta, Massimiliano and Esposito, Raffaele and Villani, Maria Luisa and Codato, Roberto and Colombo, Massimiliano and Di Nitto, Elisabetta: WS Binder: a framework to enable dynamic binding of composite web services. In: SOSE '06: Proceedings of the 2006 international workshop on Service-oriented software engineering, pp. 74–80. ACM, New York, NY, USA (2006). DOI http://doi.acm.org/10.1145/1138486.1138502
5. Dumas, M., Spork, M., Wang, K.: Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. Business Process Management **4102/2006**, 65–80 (2006)
6. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
7. International Business Machines Corporation (IBM): WebSphere Service Registry and Repository. `http://www.ibm.com/software/integration/wsrr/` (2002)
8. Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: End-to-End Versioning Support for Web Services. In: SCC '08: Proceedings of the 2008 IEEE International Conference on Services Computing, pp. 59–66. IEEE Computer Society, Washington, DC, USA (2008). DOI http://dx.doi.org/10.1109/SCC.2008.21
9. Leitner, P., Rosenberg, F., Dustdar, S.: DAIOS - Efficient Dynamic Web Service Invocation. IEEE Internet Computing **13**(3), 72–80 (2009)
10. Lin, B., Gu, N., Li, Q.: A requester-based mediation framework for dynamic invocation of web services. In: SCC '06: Proceedings of the IEEE International Conference on Services Computing, pp. 445–454. IEEE Computer Society, Washington, DC, USA (2006). DOI http://dx.doi.org/10.1109/SCC.2006.13
11. Löwy, J.: Programming WCF Services. O'Reilly (2007)
12. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic Web Services. IEEE Intelligent Systems **16**(2), 46–53 (2001). DOI http://dx.doi.org/10.1109/5254.920599
13. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: Advanced event processing and notifications in service runtime environments. In: DEBS '08: Proceedings of the second international conference on Distributed event-based systems, pp. 115–125. ACM, New York, NY, USA (2008). DOI http://doi.acm.org/10.1145/1385989.1386004
14. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: Comprehensive qos monitoring of web services and event-based sla violation detection. In: MWSOC '09: Proceedings of the 4th

International Workshop on Middleware for Service Oriented Computing, pp. 1–6. ACM, New York, NY, USA (2009). DOI http://doi.acm.org/10.1145/1657755.1657756
15. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: End-to-End Support for QoS-Aware Service Selection, Binding and Mediation in VRESCo. IEEE Transactions on Services Computing (TSC) (2010). (forthcoming)
16. Michlmayr, A., Rosenberg, F., Platzer, C., Treiber, M., Dustdar, S.: Towards recovering the broken SOA triangle: a software engineering perspective. In: IW-SOSWE '07: 2nd international workshop on Service oriented software engineering, pp. 22–28. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1294928.1294934
17. Organization for the Advancement of Structured Information Standards: OASIS/ebXML Registry Information Model v2.0. http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebrim.pdf (2002)
18. Organization for the Advancement of Structured Information Standards: ebXML Registry Services. http://www.oasis-open.org/committees/regrep/documents/2.5/specs/ebrs-2.5.pdf (2003)
19. Organization for the Advancement of Structured Information Standards: UDDI Version 3.0.2. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi_v3.htm (2004). URL \url{http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi_v3.htm}. Visited: 2010-01-20
20. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. Computer **40**(11), 38–45 (2007). DOI http://dx.doi.org/10.1109/MC.2007.400
21. Pautasso, C., Alonso, G.: Flexible Binding for Reusable Composition of Web Services. In: Proceedings of the 4th International Workshop on Software Composition (SC'2005). ACM (2006)
22. Rosenberg, F., Celikovic, P., Michlmayr, A., Leitner, P., Dustdar, S.: An End-to-End Approach for QoS-Aware Service Composition. In: EDOC '09: Proceedings of the 2009 IEEE International Enterprise Distributed Object Computing Conference (edoc 2009), pp. 151–160. IEEE Computer Society, Washington, DC, USA (2009). DOI http://dx.doi.org/10.1109/EDOC.2009.14
23. for the Advancement of Structured Information Standards, O.: Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (2007)
24. (Sun Microsystems, Inc), M.H.: Web Application Description Language. http://www.w3.org/Submission/wadl/ (2009). URL http://www.w3.org/Submission/wadl/. Visited: 2010-02-15
25. Szomszor, M., Payne, T.R., Moreau, L.: Automated syntactic medation forweb service integration. In: ICWS '06: Proceedings of the IEEE International Conference on Web Services, pp. 127–136. IEEE Computer Society, Washington, DC, USA (2006). DOI http://dx.doi.org/10.1109/ICWS.2006.34
26. (W3C), W.W.W.C.: OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/ (2004)
27. (W3C), W.W.W.C.: Semantic Annotations for WSDL and XML Schema. http://www.w3.org/TR/sawsdl/ (2007)
28. (W3C), W.W.W.C.: SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). http://www.w3.org/TR/soap12/ (2007). URL http://www.w3.org/TR/soap12/. Visited: 2010-02-15
29. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More. Prentice Hall PTR, Upper Saddle River, NJ, USA (2005)
30. Yu, Q., Bouguettaya, A.: Framework for web service query algebra and optimization. ACM Trans. Web **2**(1), 1–35 (2008). DOI http://doi.acm.org/10.1145/1326561.1326567