Integrated Metadata Support for Web Service Runtimes

Florian Rosenberg, Philipp Leitner, Anton Michlmayr and Schahram Dustdar
Distributed Systems Group, Technical University Vienna
Argentinierstrasse 8/184-1, 1040 Vienna, Austria
lastname@infosys.tuwien.ac.at

Abstract

Service metadata is an important aspect when developing applications following the service-oriented architecture paradigm. Such metadata includes a description of functionalities offered by a service, pre- and postconditions and data that is produced and consumed by a service, as well as a categorization of functionalities in the domain. Providing expressive metadata for services as part of the runtime infrastructure is necessary to leverage adaptability and autonomic behavior such as dynamic (re-)binding, service selection, invocation and composition. In this paper we present a model and its implementation for adding a reasonable amount of service metadata to foster their use in service-oriented applications and describe how to map concrete Web services to this metadata model. Furthermore, we explain our model based on an illustrative example from the telecommunications domain.

1. Introduction

The service-oriented architecture (SOA) paradigm provides a means to develop more flexible software applications by leveraging principles such as loose coupling, dynamic binding and invocation as well as service composition [12]. The core entities of a SOA-based design are services, which implement some functionality and expose it in a platform-independent manner to service consumers. Services are often published in a service registry to achieve better decoupling between providers and consumers. Web services, which build upon the main standards SOAP and WSDL, are one widely adopted realization of SOA.

Applications adhering to this paradigm should be less vulnerable to changes and provide better support for reusing, adding, removing or exchanging services at runtime. In practice, however, an implementation of the SOA paradigm does often not live up to its expectations [9]. One of the biggest challenges is the provisioning of flexible self-adaptive service-oriented applications. Following Cheng et al. [1], self-adaptive applications "assess their own behaviour and change it when the assessment indicates a need to adapt due to evolving functional or non-functional requirements". Within the context of service-oriented applications this aspect is composed of a number of challenges, such as dynamic (re-)binding to different services, autonomic service selection (e.g., based on Quality of Service or QoS for short), service versioning or flexible service composition, just to name a few. We aim at solving some of these issues in the VRESCo project [5, 9]. Its main goal is to develop a service runtime that is targeted to enterprise-level SOA development. In this runtime, all company services can be published, managed and invoked. This does not only include services that are developed internally but also services from business partners that are invoked as part of the application logic.

In this paper, we tackle the problem that current SOA runtimes lack an integrated mechanism allowing to express metadata about services as part of their core runtime functionality. This is necessary to achieve a high degree of decoupling of service consumers and providers, with the ultimate goal of binding to a given "feature" (functionality), that is implemented by concrete services, rather than to concrete service instances themselves. The service runtime environment has to provide the necessary abstractions and mechanisms for realizing this use case. Therefore, we discuss an integrated metadata model for services, which enables application developers to describe the functionality that services offer, the input and output of a service operation and the pre- and postconditions of a service (typically defined in the domain model). A metadata description specifies an abstract service in terms of features that have to be mapped to concrete service instances (including possible transformations if the interfaces do not match). Additionally, the model provides a way to categorize services according to common business functionality. Please note that our metadata model is not intended to compete with approaches used in the Semantic Web services community (SWS) [7], such as OWL-S for describing semantics of services using ontologies. We aim at enterprise develop-

ment where metadata is an important business asset which should not be accessible for everyone, as opposed to the SWS community where domain ontologies should be public to facilitate integration among different providers and consumers. It is therefore important to provide a deep integration of the metadata model with the core services provided by VRESCo.

The remainder of this paper is structured as follows: Section 2 describes an illustrative scenario where a metadata model is important to achieve dynamic service selection within an enterprise scenario. In Section 3 we depict our metadata model that forms the core of the VRESCo runtime. Section 4 presents a detailed mapping example from the VRESCo metadata model to concrete service instances. Section 5 describes the implementation of this model within the VRESCo runtime. In Section 6 we discuss some of the related work and finally Section 7 concludes this work and highlights some future work.

2. Illustrative Example

In this case study we tackle the problem of building a composite service for *cell phone number portability*. Such a service is currently available to customers when they change the cell phone operator (CPO) and want to keep their old number, thus the telephone number has to be ported to the new operator. We assume a simplified process such as the one depicted in Figure 1.

The process itself runs internally within the CPO where the customer recently signed a contract. After signing the contract, the new CPO has to port the customer's old number. Therefore, the CPO has to coordinate with the customer's old provider in order to support this feature.

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

In this process, a number of dynamic service bindings exist: the external `Number Porting Service` that has to be used is an outcome of the result of the `Lookup CPO` activity and cannot be determined statically; the same is true

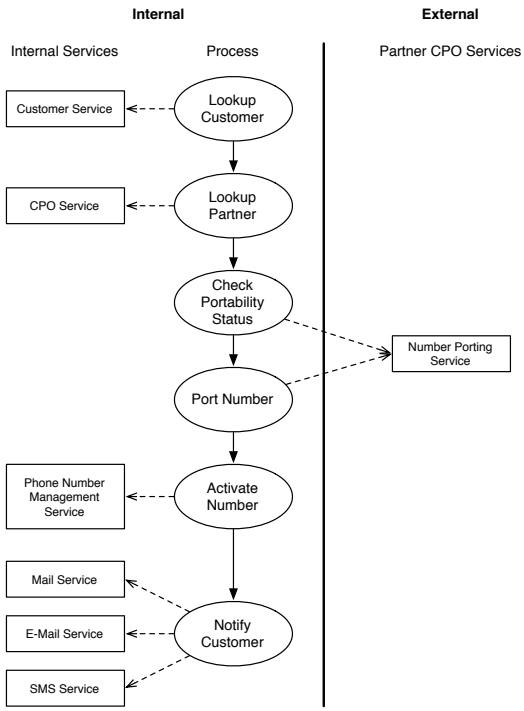

Figure 1. Number Portability Process

for the notification service used to implement the last activity in the process. The possible alternatives for each of these activities are well-known, and their number is relatively small: in this example it is not reasonable to assume that the CPO wants to cooperate with unknown partners, or that a previously unknown notification service (e.g., a public service from the Internet) should be used. However, the possible alternatives are not static, new CPOs may enter the market while others leave, and new notification services may be implemented while others are deactivated. The process itself should not have to be adapted manually as a result of such changes in the environment. Additionally, we cannot assume that each of the services has the same interface, or relies on the same implementation-level data types, i.e., the services selected and bound at runtime may vary significantly in terms of both interfaces and implementation. This complicates the problem of dynamic binding, since mediation between per se incompatible invocations and interfaces may become necessary. Therefore, standard programmatic approaches to handle variability such as the well-known Strategy pattern [3] are not suitable even in this relatively simple illustrative example.

3. VRESCo Metadata Model

In this section we describe the VRESCo metadata model, an abstract, feature-driven model for defining what

functionality is offered by a service. Additionally, we describe the mapping from this model to concrete service instances and discuss mechanisms to express the state of services instances using the proposed metadata model.

3.1. Metadata Model

In Figure 2, we have depicted our basic metadata model for modeling services, their features, pre- and postconditions. We use a slightly relaxed UML notation in the figure. In this model we have to abstract from the technical service implementation to achieve a common understanding what a service does and what it expects and provides. In a typical SOA environment, there may be multiple services that facilitate the same business goal, therefore, we also need a way to group services according to their functionality. In the following, we use *italic* font to represent model elements and `typewriter` to indicate instances of a model element.

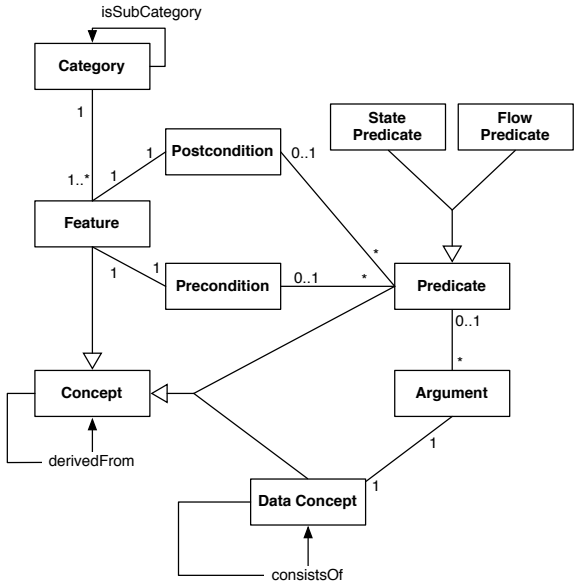

Figure 2. VRESCo Metadata Model

The main building blocks of the VRESCo metadata model are *Concept*s. A *Concept* is the definition of an entity in the domain model (an example of a domain model is shown in Figure 3 below). We distinguish between three different types of *Concept*s:

- *Feature*s represent activities in the domain that perform a concrete action, e.g., `Check_Portability_Status`, `Port_Number` or `Notify_Customer` from the example process in Figure 1.
- *Data Concept*s represent concrete entities in the domain (e.g., customers, addresses or bills) which are defined using other *Data Concept*s (e.g., the concept `Customer` might consist of `Customer_Id`, `Customer_Name`, and `Address`) and/or atomic elements such as strings or numbers.
- *Predicate*s represent domain-specific statements (propositional functions in a mathematical sense) that either return `true` or `false`. Each *Predicate* can have a number of *Argument*s that express their input. For example, a (state) predicate for a *Feature* `Port_Number` could be `Portability_Status_Ok(Phone_Number)`, expressing the portability status of a given phone number.

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

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

- *Flow Predicate*: this type of predicate can be used in pre- and postconditions to indicate constraints related to the data flow, such as data required by a feature or produced by a feature. This is expressed by using two special variants of flow predicates called `requires` and `produces`. An example from our process in Section 2 would be a *Postcondition* having a predicate `requires(Customer)`, expressing that a concept `Customer` is needed as an input for feature `Check_Portability_Status`. In case of a *Postcondition*, the predicate `produces(Portability_Status)` can be used to express that the aforementioned feature produces the data concept `Portability_Status` as output.
- *State Predicate*: this type of predicate expresses some global behavior that is valid either before (for a *Precondition*) or after invoking a feature (for a *Postcon-*

dition). For example, a postcondition can be added to the `Notify_Customer` feature expressing the global state change after a successful notification, e.g., `notified(Customer)`.

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

3.2. Mapping to Service Level

In the following section we substantiate the VRESCo metadata model as described in Section 3 by explaining the mapping of concrete Web services to the domain model.

Service Model. The service model that is used for the mapping basically follows the Web service based notation as introduced by WSDL. Concrete services represent a collection of service operations. Every operation may have a number of input parameters, and may return one or more output parameters. Additionally, operations may depend on a certain state in order to function as expected, and may result in a state change on successful execution. These state changes are modeled by using state predicates as part of pre- and postconditions.

Mapping. The elements of the service model can now be mapped to our metadata model: Services are grouped into categories (*Category*), where every service may belong to several categories at the same time. Services within the same category provide at least one feature of this category.

Service operations are mapped to features (*Feature*). Currently we assume a 1:1 mapping between features and operations; every feature is implemented in exactly one service operation, and every operation implements exactly one feature of a category. The input and output parameters of the service operations map to data concepts (*Data Concept*). Every parameter is represented by one or more concepts in the domain model. This means that all data that a service accepts as input or passes as output is well-defined using data concepts and annotated with the flow predicates `requires` (for input) and `produces` (for output).

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

3.3. Modeling with State Predicates

There is still a fruitful debate in academia and industry whether services should be designed in a stateful or stateless way. The former implies that service operations have to be invoked in a well-defined order to work correctly, because they keep some internal state between two or more service invocations. We refer to this message sequence as the protocol of a stateful service. In the latter case, every service operation is fully transparent to any previous or later invocation. Stateless services need to be given the full context of any invocation as input, and do not inflict state changes (often referred to as side effects) on usage. Stateful services pose a clear problem for the purposes of this paper because the requirements and outcomes of a service invocation are in that case, unlike with stateless services, not sufficiently described through the service contract.

In our approach *State Predicate*s are used to capture the state of stateful services. *State Predicate*s are semantically well-defined concepts in the domain model. Like all *Predicate*s in our model *State Predicate*s can have arguments, which are used to further describe a state. *State Predicate*s can be used in the precondition as well as in the postcondition of *Feature*s. If a *State Predicate* is used in the precondition then the service consumer has to ensure the validity of this state before using the *Feature*; if a *State Predicate* is used in the postcondition then the provider of the *Feature* ensures this state as a result of successful service execution.

Using these *Predicate*s one can model the protocol of stateful services. However, note that VRESCo does not enforce the *State Predicate*s – there is no way how VRESCo can supervise whether a certain state is actually mandatory for the success of a *Feature*, or whether an execution really led to the expected state change. VRESCo relies on the service publisher to define these *Predicate*s correctly.

4. Mapping Example

After a detailed discussion of the metadata model itself, we now demonstrate a small mapping example taken from the number porting process from Figure 1. We use two activities that are mapped to concrete service instances, namely, `Check_Portability_Status` and `Port_Number`. We use the UML class diagram notation to visualize the service metadata model for our example. Thus, we apply stereotypes for each UML model element to

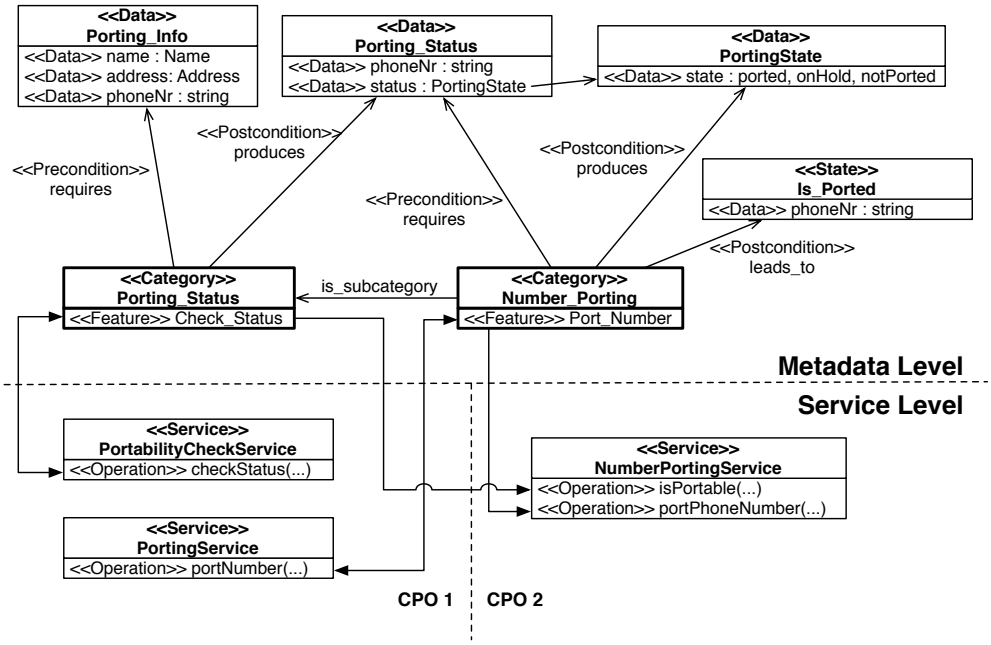

Figure 3. Mapping Example

Concept	PortabilityCheckService	Mapping Operator	NumberPortingService	Mapping Operator
	Input Mapping			
Porting_Info.name.firstname	PortingRequest.firstname (string)	==	firstname	==
Porting_Info.name.lastname	PortingRequest.lastname (string)	==	lastname	==
Porting_Info.address.street	PortingRequest.street	concat	–	–
Porting_Info.address.apt			–	–
Porting_Info.address.zip	PortingRequest.zipcode (int)	stringToInt	–	–
Porting_Info.address.country	PortingRequest.country (string)	==	–	–
Porting_Info.phoneNr	PortingRequest.number (string)	==	phoneNr	==
	Output Mapping			
Porting_Status.phoneNr	PortingRequest.number (string)	==	–	==
Porting_Status.status	PortingRequest.result (boolean)	(Porting_Status.status == "ported") ? true : false	boolean	(Porting_Status.status == "ported") ? true : false

Table 1. Check_Status Feature Mapping

indicate the respective type (e.g., *Category*, *Feature*, *Data*, etc.) according to our metadata model.

In Figure 3, we have depicted the *metadata level* on top defining metadata for the aforementioned services in the process. The lower level expresses the *service level* showing concrete service instances with the mapping to the *metadata level*. In our case we have two concrete service providers, CPO 1 and CPO 2. In the *metadata level*, we have depicted two categories (with thicker border). The first category, `Porting_Status`, defines one feature called `Check_Status` that checks the status of a phone number porting operation. This feature has one precondition that consists of one predicate `requires` expressing that a `Porting_Info` data concept is required as an input, and it produces a `Porting_Status` concept as an output which is expressed using the `produces` predicate in the postcondition. For reasons of readability, the data concepts `Address` and `Name` (both composite types) are not shown in the figure. The second category, `Number_Porting`, is a subcategory of `Porting_Status` and it defines one feature called `Port_Number` to port the phone number from one provider to a another one. The subcategory relationship expresses an inheritance relation, meaning that every feature is inherited (similar to object-oriented programming concepts). Again, this feature has one precondition that contains one flow predicate expressing that the concept `Porting_Status` is taken as input. In addition, the `Number_Porting` feature has a postcondition (modeled as state predicate), namely `IsPorted(phoneNr)`, expressing that a given `phoneNr` is now ported.

The mapping from both categories to concrete service instances is indicated by the arrows from each feature to the concrete service operation. As it can be seen on the left side of the *service level*, CPO 1 provides two concrete services where each feature implements one feature from the *metadata level* whereas CPO 2 only provides one concrete service implementing both features. The VRESCo metadata model does not enforce any particular category and feature design, thus it is flexible enough to handle various mappings. Figure 3 does not show the mappings from the data concepts to concrete data types used in the services. To do so, we have illustrated both service interfaces with their data types and their mappings below in Listing 1 (we use Java with JAX-WS annotations for the code). Please note that the data type `PortingRequest` is not specified in the listing, it can be seen in detail in the mapping in Table 1.

```
1  /** CPO 1 Services */
2  @WebService
3  public class PortabilityCheckService {
4    @WebMethod
5    public PortingStatus
6        checkStatus(PortingRequest r) { ... }
7  }
8
9  @WebService
10 public class PortingService {
11   @WebMethod
12   public bool portNumber(PortingRequest r) { ... }
13 }
14
15 /** CPO 2 Service */
16 @WebService
17 public class NumberPortingService {
18   @WebMethod
19   public bool isPortable(String phoneNr,
20       String firstname, String lastname) { ... }
21
22   @WebMethod
23   public void portPhoneNumber(String phoneNr) { ... }
24 }
```

Listing 1. Example Services

After having defined the basic metadata for the CPO-related services, the mapping of features and concepts to concrete services and data types needs to be defined. The mapping process has to be done upon deployment time of a Web service to the VRESCo runtime (details about that are provided in Section 5). In Table 1, we have summarized the mapping for the feature `Check_Status`, separated in input mapping (for data types used as an input to a service operation) and output mapping (for data types that are returned by a service operation). The goal is to define a mapping for every concept to the input and output data structures that are used within the service operation. The mapping itself is fairly straightforward because most rules require a simple 1:1 mapping from the concept to the data type (using the == operator). The table contains a "–" for every concept element that does not need to be mapped to a concrete data type of a service operation, e.g., the `Address` concept is not needed for the `NumberPortingService`. For every other mapping an operator is given that has to be applied during the execution of a service request at runtime. For example, the `street` and the `apt` elements in the `Porting_Info.Address` concept have to be concatenated because the address property of `PortingRequest` only has a street property that expects also the apartment number. The mapping for the feature `Port_Number` can be done similarly but is not shown due to space restrictions.

5. Implementation

The metadata model presented in this paper was implemented as part of the VRESCo (Vienna Runtime Environment for Service-Oriented Computing) project. VRESCo has been introduced in [9] and aims at addressing some of the current challenges in Service-oriented Computing research [12] and practice. Among others, this includes topics related to service discovery and metadata, dynamic binding and invocation, service versioning and QoS-aware service composition. Besides this, another goal is to facilitate engineering of service-oriented applications by reconciling some of these topics and abstracting from protocol-related issues. The basic architecture of VRESCo is shown in Figure 4.

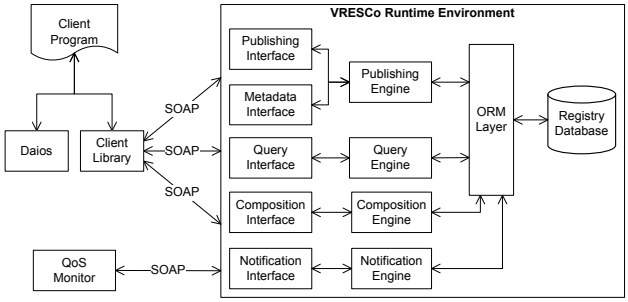

Figure 4. VRESCo Overview

To be interoperable and platform-independent, the VRESCo services are provided as Web services which can be accessed either directly using the SOAP protocol, or via the client library that provides an API for accessing these services. Services and associated metadata are stored in the registry database that is accessed using the object-relational mapping (ORM) layer. The services are published and found in the registry using the publishing and querying engine. Metadata can be accessed using a Metadata Interface. This interface allows clients to retrieve existing categories, features and concepts. Furthermore, new metadata can be inserted using this interface. The VRESCo runtime uses a QoS monitor as described in [14], which continuously monitors the QoS values of services, and keeps the QoS infor-

mation in the registry up to date. Furthermore, the composition engine aims at providing support for QoS-aware service composition which is part of our ongoing work. The event notification engine presented in [8] is responsible for notifying subscribers (e.g., using email or Web service notifications) when certain events of interest occur (e.g., new services are published, services are invoked, QoS values change, etc.). Invocations in VRESCo are issued using the integrated DAIOS dynamic Web service invocation framework [6].

When publishing services in VRESCo, clients need to specify the concrete mapping of operations to features, and of parameters to data concepts (as exemplified in Section 3.2). Furthermore, the rules that are used to transform service parameters to data concepts and vice versa have to specified. The concrete mapping engine that interprets rules and performs transformations at invocation-time (along with the language that we use to describe transformation rules) is currently work in progress.

Besides the specification of metadata upon service deployment, an important aspect is the retrieval of metadata information about a service. In addition to the Metadata Interface (described above), the Query Interface provides the ability to query services and their metadata (e.g., features, categories as well as QoS properties) using an expressive query language which uses the Metadata Interface in the backend. A static view of the metadata for a service is currently not available, but can be added easily by serializing the metadata stored in our relational data model and adding it to a service description (e.g., using WS-MetadataExchange).

Loose coupling is an ultimate goal within VRESCo, therefore, concrete services should not be invoked directly. The potential of having expressive metadata allows clients to query category(s) and feature(s) that the service needs to implement. Such queries return a DAIOS proxy that decouples clients from the services providers to be invoked by abstracting from service implementation issues such as encoding styles, operations or endpoints. This proxy is then bound to a particular service instance that implements a given feature. Re-binding of the proxy to another service that implements the same feature is then easily achieved by applying mediation between these two services. This approach is completely transparent to the client. Additionally, querying can also include the current state (expressed using state predicates). The state is used by VRESCo to determine if a given concrete service implementation is functional in a given situation (by checking the current state as specified by the client against the preconditions specified by the service implementations). Finally, clients may query for services that expect a certain input and produce a certain output (in terms of both flow and state predicates).

6. Related Work

There are several specifications which aim at integrating semantics into Web services (e.g., OWL-S [15], SAWSDL [16], etc.). OWL-S represents an ontology for services which consists of three main parts: the service profile is responsible for advertising and discovering services; the process model gives a detailed description of the service's operation; and the grounding defines how to interoperate with these service using messages. SAWSDL is an extension to WSDL that aims at integrating the semantics of services directly into service description using semantic annotations. This is done by adding an extension attribute to WSDL (*modelReference*) that defines the relation between a WSDL component and a concept in some semantic model, and lifting and lowering mappings that define how to transform concrete types into concepts and vice versa. Most of these specifications follow the idea of Semantic Web, assuming a Web of publicly available Web services with semantic information based on a global ontology. The service descriptions then point to this shared ontology. In contrast to this, the metadata model of our approach has to be queried using the querying and metadata interface, whereas the WSDL documents do not contain any semantic information. This has the advantage that metadata is not accessible for all service consumers which we believe to be an important aspect in enterprise scenarios.

Curbera and Mukhi [2] discuss the use of metadata in SOA environments. The authors introduce three usage models for metadata: multiprotocol services, dynamic discovery and negotiation, and cooperative specialization. The main focus of their work is on flexible configuration and optimization of the middleware environment (e.g., communication protocol and QoS attributes). Mukhi et. al. [10] further propose an architecture for this approach. Metadata are represented as XML documents that conform to the WS-Policy specification. The architecture is based on several message interceptors (e.g., metadata exchange, offloading negotiation, dependency negotiation, etc.) which are used to dynamically expose and negotiate service behavior between a service consumer and a service provider. Our work is complementary to theirs, since our metadata approach aims at adaptability and autonomic behavior, such as dynamic (re-) binding, selection, invocation, and composition of services.

In [13], Parastatidis and Webber propose SSDL (SOAP Service Description Language) as a language to describe Web services based on the SOAP protocol following message-orientation as the architectural paradigm. SSDL features a number of different frameworks to specify service-based systems (e.g., the Message Exchange Pattern Framework). In contrast to SSDL, our work focuses on associating metadata to services by abstracting from its concrete implementation technology. Our approach can be seen

complementary to SSDL, as it extends the SSDL concepts with additional semantics for service messages and interaction protocols.

7. Conclusions

In this paper, we have introduced a metadata model for enterprise SOA runtime environments. Our model allows the mapping of low-level Web service entities, such as Web services, WSDL operations and Web service parameters, to entities in the enterprise's domain model. Using this metadata, applications can implement functionality that allows for dynamic exchange of services, or semi-automated composition. Our model uses features to express domain activities, and data concepts to represent domain objects. Data flow is modeled using flow predicates, while state predicates are used to represent stateful Web services. Based on an illustrative example from the telecommunications domain we have shown how concrete Web services are mapped onto our metadata model. Finally, the implementation of our model within the VRESCo project has been sketched.

We think that the metadata model proposed in this paper could also be adopted by existing SOA based programming models such as the Service Component Architecture (SCA) [11] or Java Business Integration (JBI) [4]. These approaches provide models and runtimes to develop service based systems but both do not sufficiently address service metadata to achieve a higher degree of flexibility and adaptivity.

As part of our future work, we plan to develop a graphical user interface that eases metadata specification and the mapping procedure. Additionally, we will evaluate the model thoroughly regarding publishing and querying performance, and verify its applicability in real-life SOA scenarios. Our future plans also include the application of the metadata model described in this paper in areas such as semi-automated service composition and service invocation mediation.

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 Andreas Huber and Thomas Laner for a first implementation of this metadata model within VRESCo.

References

[1] B. H. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee. Software Engineering for Self-Adaptive Systems: A Research Road Map. In *Dagstuhl Seminar Proceedings*, 2008. http://drops.dagstuhl.de/opus/volltexte/2008/1500/pdf/08031.SWM.Paper.1500.pdf (Last accessed: July 28, 2008).

[2] F. Curbera and N. Mukhi. Metadata-driven middleware for web services. In *4th International Conference on Web Information Systems Engineering (WISE'03), Rome, Italy*, pages 278–286, 2003.

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

[4] Java Business Integration (JBI). http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html (Last accesssed: July 28, 2008).

[5] 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, July 2008.

[6] P. Leitner, F. Rosenberg, and S. Dustdar. DAIOS – Efficient Dynamic Web Service Invocation. *IEEE Internet Computing*. To appear.

[7] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2), 2001.

[8] 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.

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

[10] N. Mukhi, R. B. Konuru, and F. Curbera. Cooperative middleware specialization for service oriented architectures. In *Proceedings of the 13th International Conference on World Wide Web - Alternate Track Papers & Posters (WWW'04), New York, NY, USA*, pages 206–215, 2004.

[11] Open SOA. Service Component Architecture (SCA). http://www.osoa.org (Last accesssed: July 28, 2008).

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

[13] S. Parastatidis and J. Webber. SSDL – The SOAP Service Description Language. http://www.ssdl.org (Last accessed: July 28, 2008).

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

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

[16] World Wide Web Consortium (W3C). *Semantic Annotations for WSDL and XML Schema*, 2007. http://www.w3.org/TR/sawsdl/ (Last accesssed: July 28, 2008).