Towards Flexible Interface Mediation for Dynamic Service Invocations

Philipp Leitner, Anton Michlmayr, Schahram Dustdar

Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040, Vienna, Austria
lastname@infosys.tuwien.ac.at

Abstract. One of the main benefits of service-based systems is the loose coupling of components, which increases flexibility during the selection of internal and external business partners. However, currently this flexibility is severely limited by the fact that components have to provide not only the same functionality, but do so via virtually the same interface. Invocation-level mediation may be used to overcome this issue – by using mediation interface differences can be resolved transparently at runtime. In this paper we present the general concepts of invocation-level mediation, and show how these ideas are integrated into our dynamic service invocation framework Daios. To demonstrate the flexibility of our mediation framework we present two fundamentally different mediation strategies, one based on structural similarity and one based on semantically annotated WSDL.

1 Introduction

Systems based on the Service-Oriented Architecture (SOA) paradigm [1] decouple clients from service providers by leveraging standardized protocols and languages (e.g., HTTP, SOAP, WSDL) and a registry (e.g., UDDI or the ebXML registry) as service broker. In theory, this loose coupling allows service clients to roam freely between internal and external business partners, and always select the partner that is most appropriate at any given time. However, in practice this flexibility is limited by the problem that clients rely on specific service interfaces for their invocation. Therefore, services need to adhere to identical WSDL contracts in order to be interchangeable at runtime. The assumption of interface compatibility is not realistic if services are provided by different departments or companies.

Currently, most work in the area focuses on providing an infrastructure to resolve these compatibility issues: ESBs [2] provide an additional bus that decouples clients and services, and integration adapters or mediators [3,4] are used as intermediary to resolve the inherent problems of invocation heterogeneity. The approach that we present in this paper follows a different idea: we use a pure client-side approach to mediation, i.e., we enable the clients themselves to

adapt their invocation to different target services. Specific mediation behavior is introduced in the clients using mediation adapters, which can either be general-purpose or tailored towards specific domains or scenarios. This lightweight approach removes the need for an explicit mediation middleware, and resembles the traditional idea of SOA (where clients and services interact directly) more closely. The practical advantage of client-side mediation is that all context information which may be needed is readily available (e.g., which interface or service the client actually expected, or the format that the client expects the invocation result to be in). Additionally, clients are enabled to construct their individual mediation strategy (by assembling an individual chain of mediators), without relying on any specific support from the service infrastructure.

The contribution of this paper is threefold: firstly, we summarize the general concepts of service mediation; secondly, we present how the existing Daios Web service invocation framework [5] has been extended to include a dynamic mediator interface, and thirdly, we explain the implementation of two example mediators that demonstrate the capabilities of this interface. The rest of this paper is structured as follows: Section 2 clarifies the need for invocation-level mediation based on an illustrative example, Section 3 explains the general concepts of mediation, Section 4 details the Daios mediation interface and the mediators that we have implemented using this interface, and Section 5 shows the results of a preliminary evaluation. Section 6 elaborates on some related work in the field. Section 7 finally summarizes the paper, and provides an outlook on future work.

2 Motivating Example

To illustrate the need for runtime mediation we present a simple motivating example. Consider the problem of building a composite service for *cell phone number portability*. Number porting is a service pushed by the European Union that allows clients to take their mobile telephone number with them if they change their cell phone operator (CPO). The number porting related business process of a CPO may look roughly as sketched in Figure 1 (simplified for clarity).

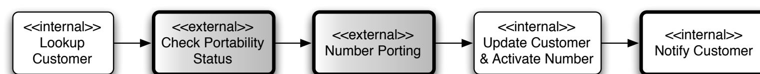

Fig. 1: Number Porting Process

The process starts by looking up the customer using the CPO-internal `Lookup Customer` service. After finding the customer, the process has to send a message to the customer's former CPO to check the portability status. If, for some reason, the porting is not possible, the process is terminated and rescheduled to be executed at a later point (not shown in Figure 1 for brevity). After the portability

check, a request is sent to the old CPO to release the number and transfer it. Afterwards, the account of the customer is updated. Finally, the customer is notified (via SMS, Mail, etc. . .) that the porting is finished.

In this process, only the activities `Lookup Customer` and `Update Customer & Activate Number` are provided by internal services, which can be assumed to have stable and relatively fixed interfaces. The activities `Check Portability Status` and `Number Porting` have to be carried out by external services provided by the CPO that the number has to be ported from. Lastly, `Notify Customer` is an internal activity, which may be provided by a variety of services made available by different internal departments (e.g., by SMS, e-mail, or mail services). This scenario illustrates how essential dynamic adaption is: in some cases the services to invoke differ between instances of the same business process; in other cases the ability to dynamically exchange service providers simply adds value to the process by increasing overall flexibility. Of course it would be possible in this scenario to use e.g., WS-BPEL `Switch` and `Assign` statements to select the appropriate partner service and explicitly reformat the invocation input and output data according to the respective target service, but this approach unnecessarily complicates the business process by shifting what is essentially an implementation issue (selecting the right service provider in a given process instance) to the business process. Additionally, this approach would only scale to a small and well-known number of alternate service providers – if the number of alternatives is very large, or if the alternatives change frequently this workaround quickly becomes unfeasible. Even worse, if the service to invoke has to be looked up dynamically in a service registry this transformation approach fails at any rate. However, note that the actual process of selecting a target service from a set of possible choices is outside the scope of this paper. We tackle this problem within our SOA infrastructure VRESCo [6, 7].

3 Interface-Level Invocation Mediation

Generally, mediation can happen on two different levels: invocation-level mediation defines the mapping of messages (single invocations) between services, while protocol-level mediation considers resolving incompatibilities in the business protocol (invocation ordering) of services. Similar distinctions have previously been identified by different researchers (among others [3, 4, 8, 9]). Per definition, protocol-level mediation is only important for stateful services, since stateless services do not rely on a specific ordering of invocations. Given that SOA traditionally focuses on stateless services we do not cover protocol-level mediation in this paper. However, others have already provided some interesting work in this area (e.g., [10, 11]).

Before going on to explain our mediation architecture, we need to define a number of general concepts that we are going to use in the forthcoming sections. Where applicable, we will use well-known terms (e.g., from the Semantic Web Services (SWS) community [12]) instead of inventing new ones. First of all, we have to distinguish between two different *formats*, *high-level (domain) concepts*

and *proprietary (low-level) formats*. High-level concepts represent things and ideas that exist in the real world, i.e., which are independent from a concrete service or implementation. High-level concepts may (but do not necessarily need to) be concepts in a Semantic Web ontology [13, 14]. Domain concepts are what domain experts talk about. Proprietary formats, on the other hand, are concrete implementations of high-level concepts. They are optimized towards concrete implementation goals, and are specific to single services. In general, proprietary formats motivate mediation – in the end, invocation-level mediation is the process of mediating between different low-level formats that implement the same domain concepts. Mediation between services is only reasonable, if the services implement the same concepts, even though they are probably using different low-level formats to represent them. The general operation of invocation-level mediation is the *transformation* of one format into another. We can distinguish three different types of transformation: (1) transforming high-level concepts into a low-level format is called *lowering* [15]; (2) the inverse operation, transforming proprietary formats into domain concepts is called *lifting*; and (3) we refer to the direct transformation of one proprietary format into another as *conversion*. These general concepts and their relationships are summarized in Figure 2.

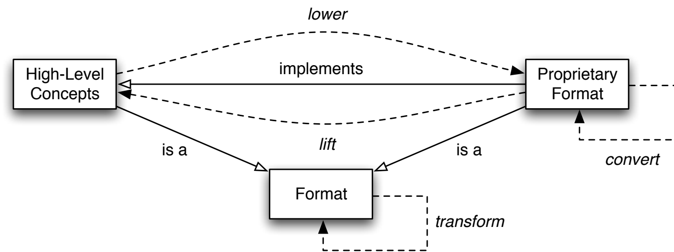

Fig. 2: General Mediation Concepts

We can distinguish three different scenarios for invocation-level mediation (Figure 3). In Scenario (a), a client expects a concrete service interface, but is actually invoking a different one. The invocation is mediated by converting the low-level format provided by the client directly to the format expected by the actual target service. Scenario (b) is similar, but in this scenario mediation is a two-step procedure. Firstly, the client invocation is lifted to domain concepts. Afterwards, this general representation is lowered to the proprietary format expected by the actual target service. The response is processed analogously. Finally, in Scenario (c) the client does not provide the service input in a proprietary format, but already in the conceptual high-level representation. Obviously, this scenario is a special case of Scenario (b) – in this case the processing is simpler since no lifting of the input and no lowering of the response is necessary.

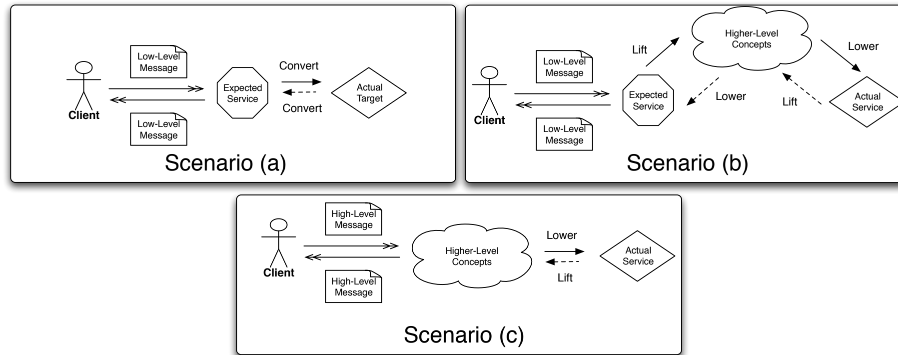

Fig. 3: Mediation Scenarios

These three scenarios are similar from an implementation point of view, but conceptually different. The first two scenarios are typical for legacy clients, or clients that invoke a specific well-known service instance "most of the time", but still need to invoke other services with different contracts from time to time. Speaking in terms of the example from Section 2, one can image that a client for the activity `Notify Customer` was implemented targeting a short message service (since this is the usual way of notifying customers), but still needs to use an e-mail service from time to time. The third scenario is characteristic for clients that have already been built with dynamic binding and runtime service selection in mind. In our example we can assume that clients for the activities `Check Portability Status` and `Number Porting` are implemented in such a way, since there is no "default" service that has to be used more often than others – in these cases, the service to use is entirely dependent on the concrete process instance. In the first scenario, no explicit high-level conceptual representation has to be available. This eases the general mediation model, but scales only to a very small number of possible service alternatives, while the Scenarios (b) and (c) are also applicable to a higher number of alternatives.

4 Mediation Adapters

In this section we will detail how client-side mediation has been implemented within the Daios [5] project. The general idea of Daios is to decouple clients from the services they are invoking by abstracting from service implementation issues such as encoding styles, operations or endpoints. Therefore, clients only need to know the address of the WSDL interface describing the target service and the input message that should be passed to it; all other details of the target service implementation are handled transparently. In Daios, data flowing into and out of services are represented by specific data structures (`DaiosMessage`s). These messages are on a higher level of abstraction than e.g., SOAP messages, and can be represented as an unordered labelled tree structure.

4.1 Daios Invocation Mediation

Even though Daios decouples clients and service providers, the clients still need to know the exact structure of the message that the target service expects. This data coupling is problematic. Services from different providers will usually not rely on the same data model, even if their functionality is equivalent or similar. Therefore, we have extended Daios to include an interface that can be used to hook a *chain of mediators* into the client. The chain of mediators implements a stepwise transformation from the original input (which may be in the proprietary format of a different service, or directly representing high-level concepts) to the proprietary format expected by the target service. Input usually enters the chain encoded as `DaiosMessage`, and the output of the chain is SOAP. Therefore, the mediator chain should at some point contain the "default" mediator, which implements the mapping of the Daios-internal message format to SOAP.

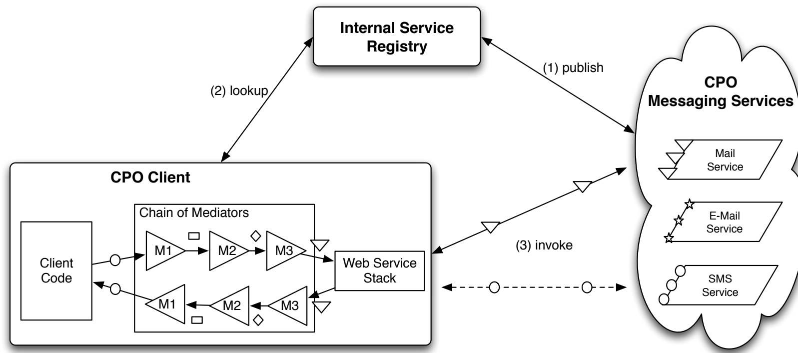

Fig. 4: Client-Side Mediation Architecture

Figure 4 sketches Daios' overall mediation architecture, and how it leverages the standard SOA model of service providers, consumers and registry. In the figure, we exemplify the mediation model based on the activity `Notify Customer` from the process in Figure 1. Additionally, we assume that the client has been developed according to the mediation Scenario (b) from Section 3. (1) A number of different messaging services are published in the service registry. The messaging services all have a similar domain purpose (sending messages to customers), but they are provided by different internal departments of the CPO and are accessible using different interfaces[1]. (2) When the activity `Notify Customer` in the process has to be carried out, the client looks up a messaging service in the registry (according to the preferences of the customer) and constructs a Daios

[1] Note that we do not assume a public service registry. Instead, we work on the assumption of a company-private service registry containing only well-known services, since such registries are more common in today's service-based systems.

frontend to the service (this is a completely automated process that mainly includes parsing the WSDL description of the service and its XML Schema type system, for more details refer to [5]). (3) Finally, the client constructs a message in the proprietary format of one of the possible alternatives (the SMS service in the example) and commences the invocation. The message is now passed through the mediation chain of this client, and will be lifted to a common domain representation using well-defined transformation rules, and again lowered to the format expected by the actual target service. As a last step, the message is serialized to SOAP. This SOAP message is then passed to a Web service stack and sent to the target service. If a return message is received as a response to the invocation, it travels through the mediator chain in the opposite direction, and is passed back to the client in the proprietary format of the SMS service.

In the end, it is the decision of the service client how to construct the mediation chain for every given invocation (i.e., which mediators should be used, and in which order). To do that, the client can choose from a set of general-purpose and domain-specific mediators. Using domain-specific mediators existing domain or service mapping knowledge can be re-used. That decision involves significant knowledge about the services that are likely to be invoked, therefore, a fully automated solution to the mediator selection problem is rather problematic. Mediators may often be able to judge if their application makes sense in a given invocation scenario (e.g., a semantic mediator can judge if semantic annotations are available), but they cannot decide if their application actually resolves all differences in the best way. In our current solution we rely on semi-automated decision making (including humans) in order to construct the mediation chain for any given client. We leave the problem of automatically constructing mediator chains in an optimal way (a problem which bears some resemblance to automated service composition [16]) for our future work.

In the following sections we will detail the implementation of two very different general-purpose mediators, which demonstrate the flexibility of our approach.

4.2 Structural Mediation

One common source for incompatibilities between services is the structure of information. A simple example is sketched in Figure 5, which shows the service interfaces of two different `check_porting_status` services (in DAIOS notion, i.e., as unordered labelled trees). The core information (telephone number, customer identifier, name, location) is contained in both interfaces, but structured differently. Additionally, both interfaces contain a number of fields which are not used in the other message. Consider the case where the user provides input such as the first one in the figure, and the service provides an interface such as the second one. In this case, the incompatibility can be resolved by stepwise transformation of the original user input (i.e., adding new nodes, removing not used nodes, renaming nodes) until the input has the same structure as the service interface. Obviously, information that is not existent in the original message will be left empty in the result message; information that has been present in the original message but not in the service interface is lost. Therefore, the resulting

message is guaranteed to be structurally valid, however, there is no guarantee that the resulting message does not miss mandatory information.

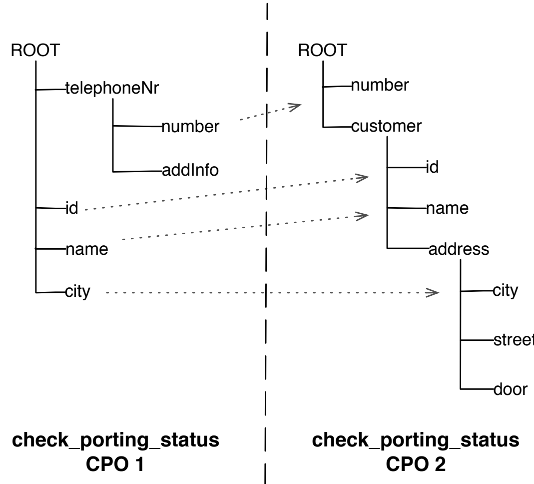

Fig. 5: Structural Interface Mediation

As part of our DAIOS prototype we have developed a general-purpose mediator that implements such a structural mediation. In terms of the concepts introduced in Section 3 this mediator can implement either Scenario (a) or (c). The mediator is universal, since it does not depend on any additional information besides the target WSDL contract and the client input, and its functionality is well suited to resolve "simple" interface differences (e.g., typical service version changes as introduced in [7]). However, the mediator has a few distinct disadvantages: essentially, the problem of finding the optimal changes to transform a given input tree to a given target format results in computing the tree edit distance (and the respective edit script, i.e., an ordered list of changes that have to be applied), which is known to be NP-hard [17]. The efficiency of the transformation can be improved by implementation-level techniques such as sub-result caching or pruning of identical subtrees. We also cache the edit script associated with every given combination of input and output trees to speed up future similar transformations, however, the transformation effort for unknown invocations still increases exponentially with the amount of change necessary.

In the description above we simplified by assuming that message fields have the same semantics iff they have the same name. In practice, this assumption only holds in rather regulated and controlled environments. In the general case, further data heterogeneity problems arise [18] (such as two fields carrying the same name, but having different semantics). Therefore, it is possible to combine the structural mediator with the semantic mediator (see below) if semantic

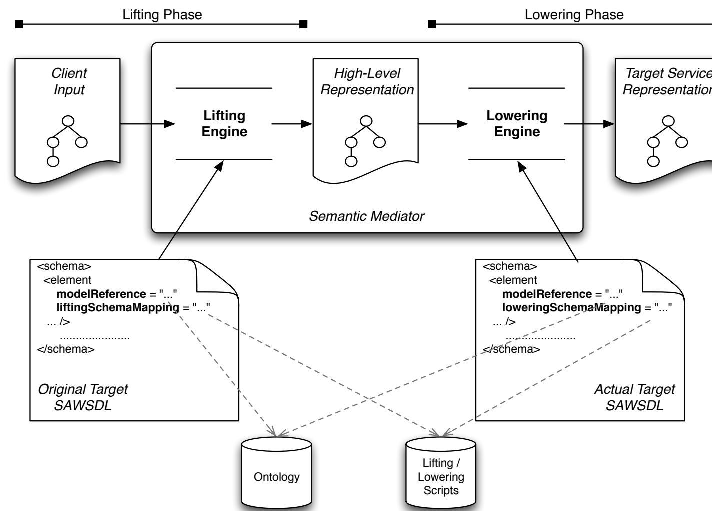

Fig. 6: Semantic Mediation

annotations are available, and use the convention that two message fields have an identical name iff they point to the same ontology concept. Handling these problems without falling back to semantic annotations is part of our future work.

4.3 Semantic Mediation

Currently, most work on Web service incompatibility resolution is carried out within the SWS community. One specifically interesting approach is SAWSDL [19, 20]. SAWSDL provides extensions for WSDL that allows to annotate XML Schema data types, operations and faults with pointers to ontology concepts. Additionally, pointers to scripts that implement lifting and lowering for data types can be added. We have implemented a general-purpose mediator that uses this semantic information in order to implement dynamic mediation between two SAWSDL-annotated Web services. In our implementation, both high-level concepts and proprietary formats are represented using `DaiosMessage`s. Mediation rules are given as transformation scripts.

Figure 6 sketches the general architecture of the Semantic Mediator in a Scenario (b) invocation: the client passes the service input, the SAWSDL description of the original target service and the SAWSDL description of the actual invocation target to the mediator, which retrieves the ontology pointers from the original SAWSDL description, and applies the corresponding lifting scripts; the resulting high-level representation is then transformed to the format expected by the target service by applying the respective lowering scripts (as denoted in the actual target's SAWSDL description). A possible response is processed equivalently (not shown in the figure). Similarly, it is also possible to use the

Semantic Mediator in Scenario (c) situations. In that case, the input provided by the client needs to be annotated with semantic information (i.e., the nodes in the `DaiosMessage` tree need to be annotated with ontology pointers). However, since the input is already provided as a high-level representation, no lifting is necessary and processing starts in the lowering phase (see Figure 6).

In Listing 1, we present an example of a transformation (lowering) script, which leverages the interpreted language Groovy[2]. The script is standard Groovy code, however, we use two special variables (`input` and `output`), which are injected into the interpreter environment (i.e., these variables are already predefined when the execution of the script starts). `Input` refers to the input that is given to the transformation, while `output` represents the transformation result.

```
1  import at.ac.tuwien.infosys.dsg.daiosPlugins.sawsdl.SemanticMessage
2
3  // map telephone number to a complex type in the output
4  ontoUri =
5    "http://infosys.tuwien.ac.at/ontology/nrPorting/data#telephoneNr"
6  newTel = new SemanticMessage()
7  newTel.setString("number", input.getStringByConcept(ontoUri))
8  output.setComplex("telephone_nr", newTel)
9
10 // map date to a split string in the output
11 ontoUri =
12    "http://infosys.tuwien.ac.at/ontology/nrPorting/data#date"
13 merged = input.getStringByConcept(ontoUri)
14 if(merged != null) {
15
16   year = Integer.parseInt(merged.substring(0,4))
17   month = Integer.parseInt(merged.substring(6,7))
18   day = Integer.parseInt(merged.substring(9))
19   newDate = new SemanticMessage()
20   newDate.setInt("day",day)
21   newDate.setInt("month",month)
22   newDate.setInt("year",year)
23   output.setComplex("porting_date", newDate)
24
25 }
```

Listing 1: Lowering Script Example

In the listing, two values are mapped: on the one hand, on lines 4 to 8, a rather simple mapping of a value (identified by the URI `http://infosys.tuwien.ac.at/ontology/nrPorting/data#telephoneNr`) to a slightly nested data structure (`telephone_nr/number`). On the other hand, on lines 11 to 23, a more complex transformation which splits the value of `http://infosys.tuwien.ac.at/ontology/nrPorting/data#date` into three separate message fields, `porting_date/day`, `porting_date/month` and `porting_date/year`.

Since transformation rules are defined using standard Groovy scripts arbitrary complex transformations may be defined. However, it is not mandatory to use Groovy: since our semantic mediator is based on the the Java 6 scripting

[2] `http://groovy.codehaus.org/`

engine (`javax.script`), transformation scripts can be written in many different interpreted languages (however, we use Groovy by default due to its tight integration with the Java runtime environment). If a different scripting language should be used a new `TransformationFactory` needs to be introduced. The transformation factory implements the loading of the correct scripting engine, and the execution of the script. Listing 2 exemplifies how a new Jython[3] script interpreter can be introduced into the semantic mediation engine.

```
1  TransformationFactory.transformation.put(
2    "application/jython",  // transformations are bound to MIME types
3    JythonTransformer.class  // subclasses TransformationFactory
4  );
```

Listing 2: Integrating New Transformation Engines

The semantic mediator is powerful, however, it depends on the availability of semantically annotated WSDL descriptions, which are not widespread today. Additionally, the script-based approach utilized in our semantic mediator introduces a certain amount of processing overhead (see Section 5). We argue that semantic mediation is best used in cases which demand for extensive semantic transformations, e.g., the integration of existing legacy applications into service-based systems.

5 Evaluation

We consider runtime performance to be one of the most critical factors of a Web service mediation framework. Therefore, we have carried out a number of performance tests to learn about the processing overhead introduced by the two mediators presented in Section 4. We have compared the performance of unmediated invocations and invocations using the structural and semantic mediator. For the semantic mediator we have evaluated two different scenarios, one resembling Scenario (b) (both lifting and lowering of messages is necessary) and one resembling Scenario (c) (only lowering is necessary). All tests have been carried out on a 2.4 GHz Intel Core 2 Duo machine, with both test clients and services running on the same machine. Every test has been repeated 100 times, and results have been averaged. We summarize the outcomes of this evaluation in Figure 7.

Figure 7a depicts how the invocation time increases with increasing SOAP message payload. Interpreting the figure, we can see that the invocation time increases proportionally with the payload for unmediated invocations (this finding is in line with the results we have already presented in [5]) and all types of mediators. This means that the actual mediation overhead of all mediators is relatively

[3] `http://www.jython.org/Project/`

independent of the payload size. However, we can also see that specifically the semantic mediator introduces a sizable overhead.

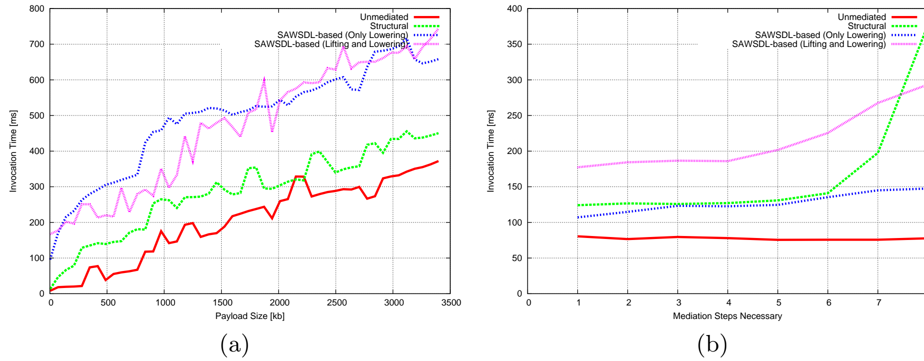

Fig. 7: Runtime Performance Comparison

Obviously, the concrete overhead introduced depends on the amount of mediation necessary. This is depicted in Figure 7b. Here, we plot the invocation time depending on the number of incompatibilities that have to be resolved (e.g., adding one field to an input message). The figure shows that the overhead introduced by the semantic mediator is increasing rather slowly – the bigger part of the overhead is the constant time necessary to load the lifting and lowering scripts, and launch the respective scripting engine. Here, additional tuning might significantly improve the overall performance. The overhead of the structural mediator, on the other hand, varies significantly with the number of incompatibilities: for a small number the mediator introduces practically no overhead, but starting with a certain amount of change the invocation time rises exponentially. This is due to the NP-hardness of the underlying calculation of the tree edit script (see Section 4). In the future, we plan to improve this specific mediator using a heuristic tree edit distance algorithm instead of the currently used deterministic one. However, keep in mind that these results are only valid for the "first" invocation using a given input – because of caching the structural mediator does not inflict a noteworthy delay on any following invocation with the same structure of this client.

6 Related Work

Most related work in the area of resolving interface incompatibilities promotes adapter-based approaches [9, 21]. These adapters are conceptually similar to our mediators, but are more decoupled from the actual clients. We consider these approaches valuable, but think that our approach is more in line with the traditional idea of SOA where clients and providers interact directly. Additionally, our

work allows for simple integration of more complex mediation scenarios, such as mediation based on service semantics (which is hard to accomplish independently from the client). Within the grid computing community, a syntactic mediation approach similar to ours has been proposed [22]. This work uses an ontology-based mediation approach for grid services. Integration of domain-specific mediation knowledge, or structural mediation without semantic information is not covered. To the best of our knowledge no flexible integrated interface mediation framework for general Web services environments like ours has been presented so far.

Other related work has studied mediation on business protocol level. In [3], a number of protocol mismatch patterns are identified, and possible solutions are proposed. In [10], Dumas et al. propose a visual notation and a set of operators that can be used to resolve business protocol incompatibilities. In current industry solutions, service mediation is often handled at ESB level [2]. However, the mediation capabilities of current ESBs such as Apache ServiceMix[4] are limited: from the scenarios presented in Section 3 only Scenario (a) is supported (direct transformation, e.g., applying an XSLT stylesheet to SOAP messages). Service mediation is an often discussed use case for semantic Web services. However, most related work in this community focuses on business protocol incompatibilities. One example is the WSDF framework [23], which uses an RDF model to resolve protocol incompatibilities. Similar research has also been presented in [11]. As part of their work on WSMX, Cimpian et al. have employed the Web Service Modeling Ontology (WSMO) for service mediation [8]. Unlike most other related work, they consider semantic mediation on both interface and process level. These semantic approaches rely on the existence of shared ontologies and explicit semantic information. Even though the same restriction applies for the semantic mediator presented here, our general mediation interface can perfectly be used with other (e.g., domain- or service-specific) mediators when no additional semantic information is available.Other authors have considered mediation on service composition level, e.g., in [24], WS-BPEL processes are adapted by exchanging service bindings at runtime, and compatibility between services is ensured using XSLT-based transformation. Others use annotated WSDL with context information to mediate semantic Web service compositions [25].

7 Conclusion

Currently, dynamic selection of services in SOA-based systems is severely limited by incompatibilities in the interfaces of these services. Enterprise integration solutions such as ESBs or mediation middleware can be used to resolve these problems, but these solutions add additional layers and complexity to the systems built. In this paper we have presented a flexible mediation architecture that enables clients themselves to adapt to varying service interfaces. We have explained the general concepts of interface-level mediation, and how these concepts have been implemented within our existing Daios project. The implementation of two

[4] `http://servicemix.apache.org/`

conceptionally different mediators has been used to demonstrate the flexibility of our approach. Additionally, we have discussed the performance overhead introduced by these mediators. Furthermore, we have critically assessed the benefits and limitations of these mediators. Given that there is no single mediator that is able to handle every situation, the incorporation of domain-specific mediators is possible. Additionally, it is possible to combine a number of different approaches in a chain of mediators.

As part of our future work, we plan to extend the work presented in this paper in various directions: firstly, we envision to extend our approach also to mediation on protocol level; secondly, we are aiming at aligning Daios more closely with our SOA runtime environment VRESCo [6, 7], in order to provide an end-to-end SOA environment to clients, and to allow the usage of the VRESCo metadata [26] model for invocation mediation; thirdly, we want to improve the implementation of our general-purpose mediators, e.g., develop a heuristic algorithm to speed up the structural mediator; and lastly, we plan to carry out some work in the area of automated mediator chain construction, i.e., automated determination of a sequence of mediators that is best suited to resolve a given set of incompatibilities.

Acknowledgements

We would like to thank Florian Rosenberg for many helpful discussions and reviews of this paper. 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. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. IEEE Computer **11** (2007)
2. Schmidt, M.T., Hutchison, B., Lambros, P., Phippen, R.: The Enterprise Service Bus: Making Service-Oriented Architecture Real. IBM Systems Journal **44** (2005)
3. Benatallah, B., Casati, F., Grigori, D., Nezhad, H.R.M., Toumani, F.: Developing Adapters for Web Services Integration. In: Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE). (2005)
4. Stollberg, M., Cimpian, E., Mocan, A., Fensel, D.: A Semantic Web Mediation Architecture. In: CSWWS. Volume 2 of Semantic Web And Beyond Computing for Human Experience. (2006)
5. Leitner, P., Rosenberg, F., Dustdar, S.: Daios – Efficient Dynamic Web Service Invocation. to appear in IEEE Internet Computing (2009)
6. Michlmayr, A., Rosenberg, F., Platzer, C., Dustdar, S.: Towards Recovering the Broken SOA Triangle – A Software Engineering Perspective. In: Proceedings of the International Workshop on Service Oriented Software Engineering (IW-SOSWE). (2007)
7. Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: End-to-End Versioning Support for Web Services. In: Proceedings of the International Conference on Services Computing (SCC). (2008)

8. Cimpian, E., Mocan, A., Stollberg, M.: Mediation Enabled Semantic Web Services Usage. In: Proceedings of the Asian Semantic Web Conference (ASWC). (2006)
9. Lin, B., Gu, N., Li, Q.: A Requester-Based Mediation Framework for Dynamic Invocation of Web Services. In: Proceedings of the International Conference on Services Computing (SCC). (2006)
10. Dumas, M., Spork, M., Wang, K.: Adapt or Perish: Algebra and Visual Notation for Service Interface Adaptation. In: Proceedings of the International Conference Business Process Management (BPM). (2006)
11. Williams, S.K., Battle, S.A., Cuadrado, J.E.: Protocol Mediation for Adaptation in Semantic Web Services. In: Proceedings of the European Semantic Web Conference (ESWC). (2006)
12. McIlraith, S.A., Son, T.C., Zeng, H.: Semantic Web Services. IEEE Intelligent Systems **16** (2001)
13. Maedche, A., Staab, S.: Ontology Learning for the Semantic Web. IEEE Intelligent Systems **16** (2001) 72–79
14. Pulido, J.R.G., Ruiz, M.A.G., Herrera, R., Cabello, E., Legrand, S., Elliman, D.: Ontology Languages for the Semantic Web: A Never Completely Updated Review. Knowledge-Based Systems **19** (2006) 489–497
15. Kopecky, J., Roman, D., Moran, M., Fensel, D.: Semantic Web Services Grounding. In: Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT-ICIW '06), Washington, DC, USA, IEEE Computer Society (2006) 127
16. Jinghai Rao and Xiaomeng Su: A Survey of Automated Web Service Composition Methods. In: Proceedings of First International Workshop on Semantic Web Services and Web Process Composition, Springer-Verlag (2004) 43–54
17. Bille, P.: A Survey on Tree Edit Distance and Related Problems. Theoretical Computer Science **337** (2005)
18. Kim, W., Seo, J.: Classifying schematic and data heterogeneity in multidatabase systems. Computer **24** (1991) 12–18
19. World Wide Web Consortium (W3C): Semantic Annotations for WSDL and XML Schema. (2007) `http://www.w3.org/TR/sawsdl/` (Last accessed: April 15, 2008).
20. Kopecky, J., Vitvar, T., Bournez, C., Farrell, J.: SAWSDL: Semantic Annotations for WSDL and XML Schema. IEEE Internet Computing **11** (2007) 60–67
21. Cavallaro, L., Di Nitto, E.: An Approach to Adapt Service Requests to Actual Service Interfaces. In: Proceedings of the International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). (2008)
22. Szomszor, M., Payne, T.R., Moreau, L.: Automated Syntactic Medation for Web Service Integration. In: Proceedings of the IEEE International Conference on Web Services. (2006)
23. Eberhart, A.: Ad-hoc Invocation of Semantic Web Services. In: Proceedings of the International Conference on Web Services (ICWS). (2004)
24. Moser, O., Rosenberg, F., Dustdar, S.: Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In: Proceedings of the 17th International Conference on World Wide Web (WWW). (2008)
25. Mrissa, M., Ghedira, C., Benslimane, D., Maamar, Z., Rosenberg, F., Dustdar, S.: A Context-Based Mediation Approach to Compose Semantic Web Services. ACM Transactions on Internet Technology **8** (2007)
26. Rosenberg, F., Leitner, P., Michlmayr, A., Dustdar, S.: Integrated Metadata Support for Web Service Runtimes. In: Proceedings of the Middleware for Web Services Workshop (MWS'08), co-located with the 12th IEEE International EDOC Conference. (2008)