Towards a Process Mediation Framework for Semantic Web Service

Vaculín, Roman
2008

Institute of Computer Science
Academy of Sciences of the Czech Republic

Towards a Process Mediation Framework for Semantic Web Services

Roman Vaculín, Roman Neruda, Katia Sycara

Technical report No. 1048, November 2008

Pod Vodárenskou věží 2, 182 07 Prague 8, phone: +420 266 051 111, fax: +420 286 585 789,
e-mail:ics@cs.cas.cz

Institute of Computer Science
Academy of Sciences of the Czech Republic

Towards a Process Mediation Framework for Semantic Web Services[1]

Roman Vaculín[2], Roman Neruda, Katia Sycara[3]

Technical report No. 1048, November 2008

Abstract:

The ability to deal with incompatibilities of service requesters and providers is a critical factor for achieving interoperability in dynamic open environments. We focus on the problem of process mediation of semantically annotated process models of the service requester and the service provider. We propose an Abstract Process Mediation Framework identifying the key functional areas that need to be addressed by process mediation components. Next, we present algorithms for solving the process mediation problem in two scenarios: (a) when the mediation process has complete visibility of the process model of the service provider and the service requester (*complete visibility scenario*), and (b) when the mediation process has visibility only of the process model of the service provider but not the service requester (*asymmetric scenario*). The algorithms combine the AI planing and semantic reasoning with recovery techniques and the discovery of appropriate external services such as data mediators. Finally, the Process Mediation Agent (PMA) is introduced, which realizes an execution infrastructure for runtime mediation.

Keywords:
process mediation, semantic web services, agents and web services, automated planning

[1]This research was partially supported by the the Czech Ministry of Education project ME08095 and "Information Society" project 1ET100300517. This research was supported in part by Darpa contract FA865006C7606 and in part by funding from France Telecom.

[2]Roman Vaculín, Roman Neruda – Institute of Computer Science, Academy of Sciences of the Czech Republic ({vaculin,roman}@cs.cas.cz)

[3]Katia Sycara – The Robotics Institute, Carnegie Mellon University (katia@cs.cmu.edu)

1 Introduction

One of the main promises of Web Services standards is to enable and facilitate smooth interoperability of diverse applications and business processes implemented as components or services. The most of existing web services are programmed to exchange data according to some protocol. In addition to that, individual web services serve as building blocks in process models that prescribe control and data flows and thus define functionalities of more complex applications. However, as business needs change, processes could get reconfigured, replaced by new providers, or additional components and services may need to be added. As a result of these changes, the existing processes must become interoperable with the new ones. The possibility of achieving interoperability of existing processes without actually modifying their implementation and interfaces is therefore desirable.

Current Web Services standards provide a good basis for achieving some level of interoperability. WSDL [12] standard allows to declaratively describe operations, the format of messages, and the data structures that are used to communicate with a web service. BPEL4WS [3] adds the possibility to combine several web services within a formally defined process model, and so to define the interaction protocol and possible control flows. However, neither of these two standards goes beyond the syntactic descriptions of web services. Newly emerging standards for semantic web services such as SAWSDL [17], OWL-S [24] and WSMO [21] strive to enrich syntactic specifications with rich semantic annotations to further facilitate flexible dynamic web services invocation, discovery and composition [23]. However, this is not enough for achieving the interoperability in dynamic environments, at least for two reasons.

First, service providers and requesters do not typically share the same data models, interaction protocols and oftentimes even not the basic standards for WS specification. As a result, we have to deal with incompatibilities on different levels among service providers and requesters [29]. Second, due to the dynamic nature of the internet and rapid, unpredictable changes of business needs, also the existing web services change very often. The ability to adapt to changing environments is therefore crucial. In such cases, interoperability can be achieved by applying a process mediation component which resolves all possible incompatibilities and generates appropriate mappings between these processes. Implementing the process mediation component is complicated and costly, since it has to address many different types of incompatibilities on the data, service and process levels. As part of the process mediation, it is also necessary to deal with possible failures and unexpected behaviors of services. In open environments the process mediation component might also need to interact with other middle agents, such as discovery agents.

In this paper we address the problem of automatic mediation of process models consisting of semantically annotated web services. We are focusing on the situation where the interoperability of two components, one acting as the requester and the other as the provider, needs to be achieved. Usually, both the requester and provider adhere to some relatively fixed process models. The process models can either correspond to a particular existing implementation or they can be default (generic) process models that for example generalize a business processes of some specific problem domain (e.g., flights booking requester or provider). We analyze the problem of process mediation and we propose an *Abstract Process Mediation Framework (APMF)* that identifies the key functionalities which need to be involved to allow a successful process mediation in dynamic environments. In addition to *process mediation*, *data mediation* and *service invocation*, we introduce the notion of *semantic monitoring* and *semantic recovery*, which are novel to the process mediation and to the semantic web services area as well. The process mediation context also brings a new perspective and new problems to the *service discovery*. We discuss each of these functional areas from the process mediation perspective and we introduce technical solutions developed in the context of OWL-S semantic web services.

The problem of process mediation as well as the suitable solutions to it depend on various conditions. We study the problem with respect to the openness of the operating environment and with respect to visibility of involved process models to the mediation component. We have identified these two characteristics as important factors with a direct impact on design of effective process mediation architectures and algorithms. With respect to the environment openness, we distinguish three prototypical environment types: (1) a relatively *closed corporate intranet environment* in which development of all components can be controlled by one authority; (2) a *semi-open environment with*

a controlled registration in which software components are controlled and developed by independent authorities but the system maintainer / owner can define policies specifying how components are added and removed; (3) a *dynamic open environment* in which software components of independent vendors can be added, modified and removed arbitrarily.

With respect to visibility of involved process models, for a given process model we distinguish three cases, depending on the amount of information that the mediation component has about the process model:

1. *complete visibility* (*white box model*): the mediation component can see the complete process model
2. *partial visibility* (*gray box model*): the mediation component can see only definitions of available operations (i.e., atomic processes in the OWL-S terminology), while the interaction protocol is not disclosed
3. *no visibility* (*black box model*): the mediation component does not have any information about the process model, other than knowing the address of the endpoint; so that only exchanged messages can be observed during the mediation process

By applying specific visibility options to the mediation scenario of one requester and one provider, we obtain nine possible mediation configurations. We analyze these scenarios and propose mediation techniques and architectures for two cases: (a) when the mediation component has complete visibility of the process model of the service provider and the service requester (*complete visibility scenario*), or (b) when the mediation component has visibility only of the process model of the service provider but not the service requester (*asymmetric scenario*). In both cases, our solution combines the AI planing to generate appropriate mappings between processes, and semantic reasoning with the discovery of appropriate external data mediators and suitable recovery techniques. These two scenarios are representative since they are both very realistic in current conditions and the proposed techniques can be applied to the remaining scenarios as well. The scenario (a) represents a typical situation for closed or semi-open environments of intranet or B2B applications. On the other hand, the scenario (b) represents a mediation problem usual in open environments in which clients are concerned about privacy and therefore do not wish to disclose their process models.

The main idea of this paper is that in dynamic environments agent technologies together with explicitly defined semantics of web service based process models are the key enablers for dynamic interoperability in service oriented architectures. We believe, that software agents — employing techniques such as reasoning and planning combined with approaches like dynamic discovery and recovery from failure — is the best choice available today that can offer an alternative to dealing with problems of incompatibilities and adaptivity manually. It is very likely, that as the web services will be expected to operate in a more and more autonomous fashion, we will see the transition of some of these technologies into conventional WS infrastructure.

The rest of the paper if structured as follows. In Section 2 we introduce the process mediation problem and the characteristics of closed, semi-open and open environments. Section 2.1 starts with introducing the abstract process mediation framework and it continues with description of technical solutions of each of the functional areas of the framework. In Section 3.2 we analyze modalities of the process mediation problem derived from visibility of process models by the process mediation component. In Section 4 we describe two concrete architectures addressing the problem in the complete visibility scenario and the asymmetric scenario. Section 5 discusses discovery of new mediation services. Finally in Section 6 we briefly review the related work and we conclude in Section 7.

2 Process Mediation Problem

When requesters and providers use fixed, incompatible communication protocols interoperability can be achieved by applying a *process mediation agent (component)* which resolves all incompatibilities, generates appropriate mappings between different processes and translates messages exchanged during run-time.

Interoperability of a requester and a provider might be complicated by diverse types of incompatibilities. In the context of process mediation the following types of mismatches can be identified:

1. *Data level mismatches*:
   (a) *Syntactic / lexical mismatches*: data are represented as different lexical elements (numbers, dates format, local specifics, naming conflicts, etc.).
   (b) *Structural Data mismatches:* data are represented in different datastructures (arrays, records, sets, etc.)
   (c) *Ontology mismatches*: the same information is represented as different concepts
      i. in the same ontology (subclass, superclass, siblings, no direct relationship)
      ii. or in different ontologies, e.g., (Customer vs. Buyer)
2. *Service level mismatches*:
   (a) a requester's service call is realized by several providers' services or a sequence of requester's calls is realized by one provider's call
   (b) requester's request can be realized in different ways which may or may not be equivalent (e.g., different services can be used to to satisfy requester's requirements)
   (c) reuse of information: information provided by the requester is used in different place in the provider's process model (similar to message reordering)
   (d) missing information: some information required by the provider is not provided by the requester
   (e) redundant information: information provided by one party is not needed by the other one
3. *Protocol / structural level mismatches*: control flow in the requester's process model can be realized in very different ways in the provider's model (e.g., sequence can be realized as an unordered list of steps, etc.)

We assume that both the requester and the provider behave according to specified process models and that both process models are expressed explicitly using OWL-S ontologies [24]. In OWL-S, the elementary unit of process models is an atomic process, which represents one indivisible operation that the client can perform by sending a particular message to the service and receiving a corresponding response. Processes are specified by means of their inputs, outputs, preconditions, and effects (IOPEs). Types of inputs and outputs are defined as concepts in an ontology or as simple XSD data-types. Processes can be combined into composite processes by using control constructs such as sequence, any-order, choice, if-then-else, split, loops, etc. In OWL-S, syntactic and lexical level mismatches (category 1a) are handled by the service Grounding which defines transformations between syntactic representation of web service messages and data structures and the semantic level of the process model. The Grounding provides mechanisms (e.g., XSLT transformations) to map various syntactic and lexical representations into the shared semantic representation. Finally, we assume that both process models share the same domain ontology and target conceptually the same problem.

2.1 Abstract Process Mediation Framework

We have defined an *abstract process mediation framework* (*APMF*) that identifies and separates critical functional areas which need to be addressed by mediation components in order to effectively solve the process mediation problem.

Figure 2.1 shows the abstract process mediation framework. The three key functionalities, namely *process mediation*, *data mediation* and *service invocation*, are displayed as horizontal layers. The process mediation layer, realized by *process mediators*, is responsible for resolving service level and protocol level mismatches (categories 2 and 3 defined in Section 2). The data mediation layer, realized by *data mediators*, is responsible for resolving data level mismatches (category 1 defined in Section 2). Typically, when trying to achieve interoperability, process mediators and data mediators are closely related. A natural way is to use data mediators within the process mediation component to resolve "lower" level mismatches that were identified during the process mediation. Finally, the service invocation layer is responsible for interactions with actual web services, which include the services of the requester, provider and possibly other external services.

In addition to the key functionalities, other areas might need to be covered depending on the environment characteristics. Specifically, since we assume that environments are dynamic and might be open, the *monitoring* and *recovery* functionalities must be included in the APMF. We display these

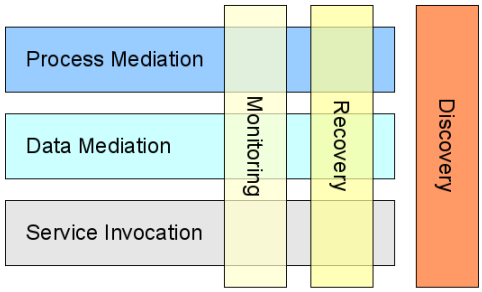

Figure 2.1: Abstract Process Mediation Framework

functionalities as vertical layers intercepting the horizontal layers because monitoring and recovery are intertwined with the key mediation functionalities and both need to be performed on all different levels. Clearly, on different levels requirements on and techniques of monitoring and recovery are different, which we discuss later in this section. Finally, we also consider the *discovery of external services* functionality as closely related to the process mediation problem in dynamic environments. For example, when an incompatibility such as a missing piece of information is identified between two processes, an external service might need to be discovered which is capable of delivering the required information. Interestingly, the requirements for discovery motivated by process mediation [27] are somewhat different when compared to traditional approaches such as [19].

2.2 Process Mediation Overview

The problem of process mediation can be seen as finding an appropriate mapping between requester's and provider's process models. The mapping can be constructed by combining simpler transformations representing different ways of bridging described mismatches. We need to decide if *structural differences* between process models can be resolved. Assuming that the requester starts to execute its process model, we want to show that for each step[4] of the requester the provider with some possible help of intermediate translations represented by *data mediators* or *built-in conversions* can satisfy the requester's requirements (i.e., providing required outputs and effects) while respecting its own process model. If no such a mapping can be found by using the known data mediators, we would like to know what are the reasons, or more specifically, what is the information gap that impedes the mediation.

In general, the process mediation layer has to address three problems: (1) finding mappings between processes, (2) identifying possible information gaps that impede the mediation, and (3) providing suitable mechanisms for the runtime mediation.

Various approaches can be applied to process mappings provision ranging from manual to fully automatic techniques. In some cases the suitable mappings can be computed automatically either prior to the mediation itself or during the runtime. In Sections 4.3 and 4.1 we present mechanisms based on AI planning for computing the mappings. In our approach the developed algorithms are also able to identify the information gaps and transform those gaps into the matchmaker queries.

In the minimalistic scenario, during the runtime mediation process the suitable / discovered mappings (transformations) has to be applied to the interactions between the requester and the provider. In the more elaborate scenario, also the monitoring and recovery mechanisms has to be incorporated to achieve a reliable and robust execution. In other words, the process mediation layer controls and integrates the other remaining layers.

Because the problem of process mediation is complex and extensive, we focus primarily on the process level mismatches while we address the data mediation only in a very limited way. We assume that *data mediators* have the form of external services which can be discovered and used by the process mediation component. Furthermore, in our system data mediators can have a form of converters that

[4]In the following text the word *step* stands for an atomic process executed by the requester. If we refer to the provider's atomic processes, we mention it explicitly.

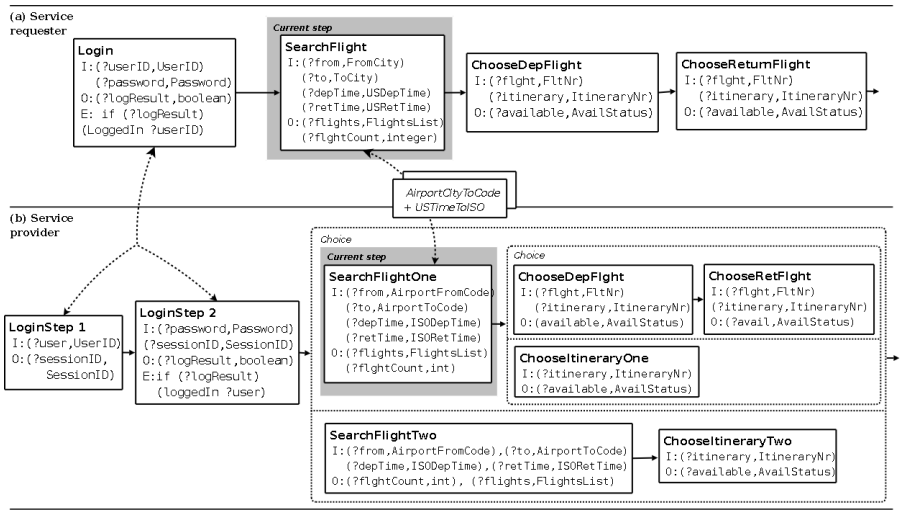

Figure 2.2: Process models of the requester and provider and possible mappings between them.

are built-in to the system. Currently, built-in converters support basic type conversions such as up-casting and down-casting based on reasoning about types of inputs and outputs. By up-casting or down-casting we mean a conversion of an instance of some ontology class to a more generic or more specific class respectively. For details related to data mediation see for example [22, 9, 26].

2.3 Motivating Example

Figure 2.2 presents an example of the mediation problem between a hypothetical requester and provider from the flights booking domain. Specifically, Figure 2.2a depicts a fragment of the process model of the requester while the provider's process model fragment depicted in Figure 2.2b represents a more elaborate scenario that allows the requester to book either the whole itinerary or to pick the departure and return flights separately.

The requester's process model starts with the *Login* atomic process that has two inputs, *userId* which is an instance of the *UserID* class and *password* of *Password* type, one output *logResult* of *boolean* type and a conditional effect expressing that the predicate *(LoggedIn userID)* will become true if the value of *logResult* equals to true. Similarly the process continues by executing other atomic processes. Inputs and outputs types used in process models refer to a simple ontology showed in Figure 2.3.

Concepts	UserID, PasswordCity, FromCity, ToCity, DateTime, USDateTime, USDepTime, USRetTime, ISODateTime, ISODepTime, ISORetTime, FlightsList, FltNr, ItineraryNr, AvailStatus, LoggedIn, AirportCode, AirportFromCode, AirportToCode
ISA-relations	USDateTime $\sqsubseteq$ DateTime, USDepTime $\sqsubseteq$ USDateTime, USRetTime $\sqsubseteq$ USDateTime, ISODateTime $\sqsubseteq$ DateTime, ISODepTime $\sqsubseteq$ ISODateTime, ISORetTime $\sqsubseteq$ ISODateTime, FromCity $\sqsubseteq$ City, ToCity $\sqsubseteq$ City, AirportFromCode $\sqsubseteq$ AirportCode, AirportToCode $\sqsubseteq$ AirportCode

Figure 2.3: Fragment of the flights domain ontology with only concepts and ISA-relations displayed.

The example demonstrates several types of inconsistencies that the PMA has to deal with and two possible mappings between both process models.Dashed arrows between parts (a) and (b) of Figure 2.2 represent symbolically possible mappings between requester's and provider's process models that can be found by the process mediation algorithm. Sometimes, the mapping can be achieved without the use of any help of other services, such as in the case of requester's *Login* atomic process. However, often

the identified data incompatibilities or missing pieces of information require external services to be used in order to construct a meaningful mapping between process models. Consider, for example, the requester's *SearchFlight* atomic process and the provider's *SearchFlightOne* process. In this particular case, a combination of external services *AirportCityToCode* and *USTimeToISO* can be used to bridge the gap as shown in Figure 2.2. The process mediation algorithm must be able to identify such gap and possibly be able to transform the the identified gap into a query for the discovery service or agent, which eventually might discover the suitable candidate external services. After that, the process mediation component uses a composition algorithm to construct the sought mapping by employing the newly discovered services.

3 Means of Process Mediation

3.1 Operating Environments Characteristics

Depending on the nature of the environment in which the interoperability has to be achieved, different approaches must be considered. A relatively *closed corporate intranet environment* in which development of all components can be controlled by one authority allows high built-in interoperability of components. An agreement on the syntax and semantics of exchanged data and communication protocols is possible in advance. Typically, either both the client and the provider can be developed to cooperate together or the client is specifically developed to interact with some particular provider service. In a closed environment it is also much easier to develop ad-hoc mediation components for individual pairs of services, because limited number of components can interact with each other and all protocols and all incompatibilities are known.

However, the environment in corporations cannot always be considered as entirely closed because services of many diverse contractors and subcontractors are used as part of organizations' business processes. We call such an environment as *semi-open*. In semi-open environments, software components are *controlled and developed by independent authorities* which engenders both data and protocol incompatibilities. Semi-open environments are *dynamic with a controlled registration*, i.e., new components can be added, removed or replaced, there can be several interchangeable components (from several contractors) that solve the same problem. The system can define policies specifying how components are added and removed. Semi-open environments typically imply some level of *trust* among contractors. Contractors are motivated to publish descriptions of interaction protocols of their components to allow interoperability with other components.

The above mentioned characteristics of semi-open environments make the mediation a much harder problem. It is simply impossible to assume that various requesters and providers will interoperate smoothly without any mediation or that a one purpose mediation component can be developed for each new service provider or requester. However, since it is reasonable to assume that each component provides a description of its interaction protocols and since mechanisms of registering components into the system can be controlled, it is possible (1) to analyze in advance if interoperability of some components is possible and if it is (2) to use mechanisms that utilize results of the analysis step to perform an automatic runtime mediation.

Dynamic open environments add more complexity to the mediation problem. Since components can appear and disappear completely arbitrarily, it is necessary to incorporate appropriate discovery mechanisms. Also it is not possible to perform an analysis step in advance because requesters and providers are not known in advance. In open environments requesters typically concern about their privacy aspects and do not wish to fully reveal their process models. Therefore the automatic mediation process must rely only on *run-time* mediation and on the limited knowledge of requester's process model.

3.2 Visibility and Means of Process Mediation

As we mentioned in introduction, we distinguish three possible degrees of visibility for a given process model. *Complete visibility* (white box model) means that the mediation component can see the complete process model, including all relevant operations (atomic processes) definitions and the complete

interaction protocol (control and data flows). In case of *partial visibility* (gray box model), the mediation component can see only definitions of available operations, i.e., atomic processes in the OWL-S terminology, while the interaction protocol is not disclosed. There might be various motivations for not disclosing the complete interaction protocol, such as privacy concerns, or its non-existence. Finally, in case of *no visibility* (black box model), the mediation component does not have any information about the process model, other than knowing the address of the endpoint. Again, such a situation is motivated either by non-existence of any specifications, or by privacy concerns. During the runtime mediation process, the mediation component can observe the messages exchanged between the requester and the provider. Exchanged messages can be annotated and provide additional semantic information, such as the format of expected response to a given message.

Assuming the problem of process mediation of one requester and one provider, nine scenarios are possible depending on visibility of each participating process model. Table 3.1 summarizes all these scenarios in the form of a visibility matrix. In the heading of each column and each row of the matrix, short characterization of visibility types for the provider and the requester are given. For some less obvious cases, we explicitly describe for what environments a given scenario is relevant. The matrix describes all nine cases of possible visibility settings in terms of a basic characterization, available means of process mediation, and in terms of how realistic and useful given scenario is.

The visibility matrix identifies boundary cases and means of mediations in these situations. If we assume that the interaction sessions with a certain requester or provider might repeat, the mediation component could use machine learning techniques to learn the process models control and data flows. Specifically, learning the process model would allow transition from black and gray box visibility model to the white box model. Various techniques such as workflow mining [1, 14], plans recognition or MDPs can be used for learning the model of provider's or requester's process model. In the following subsections we discuss various means of process mediation.

3.3 Off-line Analysis

We start with the situation, when the mediation component has a perfect information about participating process models of the requester and the provider, which is the case IX in the visibility matrix. Such a situation can be usually encountered in enterprise or B2B integration scenarios, typically in closed or semi-open environments. Usually, both process models are available and visible to the process mediation not only during the runtime, but also before the runtime mediation starts. This allows an *off-line analysis* to be performed before the runtime mediation starts. During the off-line analysis, possible mismatches between the process models can be identified and corresponding mappings using available data mediators for bridging these mismatches can be computed. The mappings can be computed either fully automatically, or a human assistance can be used for resolving possible problems. Techniques such as AI or automatic code synthesis can be used for the off-line analysis. We developed a solution based on analysis of possible requester's execution paths and their mappings to the provider's process model by employing planning techniques [29] which we present in the next section. Discovered mappings are used later during the runtime meditation to perform necessary translations.

Next, the availability of the provider's process model can be utilized for purposes of *recovery analysis*. By this we mean the process of enriching the process model with recovery statements on the level of individual services. In other words, it is possible to compute compensation or undo actions for world affecting services included in the provider's process model and add those statements to the original process model if no recovery statements were available in it. Such technique increases robustness by allowing undoing of executed parts of the provider's process model. For example, Eiter et. al. in [16] describe algorithms for computing reversing actions for existing plans / actions. Recovery analysis is available in scenarios in which at leas some visibility of provider's process model is guaranteed, i.e., cases II, III, V, VI, VIII, and IX.

3.4 Reactive Mediation

In the situation when the interaction protocol of the requester is not visible and only the operations are known in advance (case VI), the off-line analysis can be used as well. However since the protocol of the requester is not known, the mediator has to assume that the requester can call any of the

Requester / Provider	***Black Box Provider*** • No specification available in advance, only the endpoint address • Provider either responds correctly or fails • *Motivation*: privacy; definitions do not exist	***Gray Box Provider*** • Only operations visible, e.g., WSDL, OWL-S atomic processes • *Motivation*: full PM spec. does not exist; does not want to disclose interaction protocol	***White Box Provider*** • Operations and the process model are visible • *Motivation*: enterprise, B2B integration; open environments: providers motivated to publish PMs
Black Box Requester • No specification available in advance • *Environments*: open, e.g., mobile requesters • *Motivation*: privacy; specs. do not exist • Options of request / response: (1) response spec. not available; (2) request annotated with spec. of response format	*I. Black Req. - Black Prov.* • only messages can be observed by the mediator • passive mediation: forward messages + observe results, based on failures interpretation deduce possible actions (type casting, etc.) • useful for pre-defined mediation: format translation, data conversions	*II. Black Req. - Gray Prov.* • matchmaking: mediator might be able to select the best matching operation of the provider for request • possible mediation actions computed during run-time • possibly incomplete information in the request – only inputs (i.e., no mediation on outputs possible) • realistic: WSDL provider, unspec. req.	*III. Black Req. - While Prov.* • reactive mediation: for request mediate request against the PM • e.g. runtime planning • complete analysis of provider's PM possible • very realistic: open environments, mobile clients, commercial providers
Gray Box Requester • Operations are visible, e.g., e.g., WSDL, OWL-S atomic processes • *Motivation*: WSDL exists while a full PM spec. does not; does not want to disclose an interaction protocol entirely	*IV. Gray Req. - Black Prov.* • similar to *I*: messages can be observed, format of responses to the requester is known • mediation: forward messages to provider + automatic runtime mediation of responses	*V. Gray Req. - Gray Prov.* • off-line analysis: find mappings for each requester's operation • analysis possibly based on matchmaking • runtime mediation: similar to *II*, except IO(PE)s of the provider are known • recovery analysis – compensation & recovery actions can be precomputed • very realistic: WSDL on both sides	*VI. Gray Req. - White Prov.* • any request from a specified set of requester's set can come at any time • possibility: calculate mappings for all combinations of requester's & provider's operation (expensive, get a lot of garbage) • solution: similar strategy as in *III*, e.g. reactive mediation • very realistic
White Box Requester • Operations and the process model are visible • *Environments*: semi-open, closed • *Motivation*: enterprise, B2B integration, e.g., semantic web services challenge	*VII. White Req. - Black Prov.* • similar to *IV*: messages can be observed, format of responses to the requester is known • mediation: forward messages to provider + automatic runtime mediation of responses (have better knowledge of provider than in *IV*)	*VIII. Black Req. - Gray Prov.* • off-line analysis: use, e.g., requester's execution paths exploration [29] + matching of provider's operations (provider's PM is unconstrained), or non-deterministic planning [25] • mediation: employ mappings during runtime & local recovery	*IX. White Req. - White Prov.* • perfect knowledge of PMs • off-line analysis: find mappings, e.g., requester's execution paths exploration [29] + provider's PM simulation • runtime mediation: employ mappings & local + global recovery • very realistic: enterprises, B2B integration

Table 3.1: The visibility matrix

specified operations at any time. On the other hand, since the process model of provider is known, the only reasonable solution is to pre-compute mappings for all possible combinations of requester's and provider's operations. Such a solution might be viable for simple processes with small number of operations. An alternative choice is to give up an off-line analysis and to rely only on the analysis during the runtime. We call such a solution a *reactive mediation*. The reactive mediation means that any reconciliations of mismatches are calculated during the runtime only. The reactive mediation presents the only possible solution in cases when the requester's process model is not known. Consider, e.g., the case III where the provider's process model is fully visible while the requester's process model is completely invisible. Only at the run-time the mediation component starts to receive messages from the requester which need to be mapped to the provider's process model on the fly. For example, a restricted version of the run-time planning can be used to find meaningful mappings. Since such a scenario makes a sense mostly in open environments, advanced recovery and discovery mechanisms need to be incorporated.

3.5 Mediation as Matchmaking

In cases when the provider's interaction protocol is not know (i.e., cases II, V, VIII), it is important to realize that for the process mediation component it means that no operations orderings are specified by the provider. This means that *any* operation can be possibly called by the requester at any time. Thus, given the requester's request, the process mediator needs to select such an operation of the provider that matches the request the best and possible identify necessary translations for this operation. In other words, the process mediation in such cases translates to the service matchmaking problem known from the literature (e.g., [19]). Small variations apply between cases II, V, VIII as described in the visibility matrix. While cases II and V will be more usual for open environments and only the runtime mediation will be possible, case VIII might allow an off-line analysis similar to case IX with the main difference being the lack of the provider's interaction protocol. The solution for case VIII can be based on combining analysis applied in the case IX (white box requester - white box provider) with the matchmaking of for selecting the best available operation of the provider.

3.6 Passive and Pre-defined Mediation

Finally, cases I, IV and VII cover situations in which no specifications of the provider are available in advance (black box provider). If also no information about the requester is available (case I), mediation component can basically only observe the exchanged messages and to employ fault handling to identify possible failures. We call such an approach a *passive mediation*. Assuming the observed failures provide enough information to allow identification of failure causes, next time a similar message is observed, the identified cause can be used for guiding the mediation actions. Semantic monitoring and fault handling introduced in [28, 31] provide a good basis for failures interpretation. However, we did not researched the passive mediations and learning yet. As the amount of available information increases in cases IV and VII, the passive mediation can be enhanced with the reactive mediation mechanisms.

In addition to off-line analysis, passive and reactive mediation, also *pre-defined mediation* is important. By that we mean transformations or translations of exchanged messages which can be defined in advance and which do not require any complicated reasoning during the runtime. Such transformations may include translations between different formats (i.e. the mediator can play a role of an adapter), providing additional security by encrypting or signing the messages, known data format conversions, adding transactions support, etc. The passive mediation corresponds to functionalities know from traditional middleware. Pre-defined mediations do not typically depend on visibility heavily and therefore it can be applied to all scenarios identified in the visibility matrix.

4 Concrete Process Mediation Architecture

In this section we describe a concrete agent architecture and mediation techniques for (1) a *complete visibility scenario* when the mediation component has complete visibility of the process model of the service provider and the service requester (case IX defined in Table 3.1), and for (2) an *asymmetric*

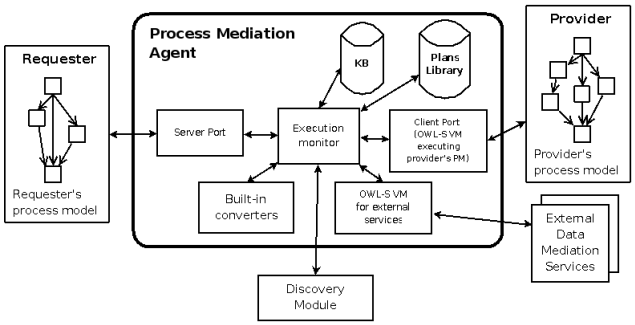

Figure 4.1: Mediation of process models by the Process Mediation Agent (PMA): problem setting and the PMA architecture

scenario when the mediation component has visibility only of the process model of the service provider but not the service requester (case III defined in Table 3.1). These two scenarios stand out in some sense compared to the other possible visibility configurations because they represent two important business integration cases. The complete visibility scenario is a typical situation for closed or semi-open environments of intranet and B2B applications, while the asymmetric scenario represents a mediation problem usual in open environments in which clients are concerned about privacy and therefore do not wish to disclose their process models. In addition to that, the scenario (1) is a perfect candidate for *offline analysis* techniques, while the scenario (2) can be solved by applying *reactive mediation*. Thus, by addressing these two scenarios we devise mediation mechanisms covering two most challenging mediation classes which can be later combined e.g. with matchmaking to deal with the remaining visibility settings.

In both cases, our solution employs a *process mediation middle agent* (*PMA*) for runtime mediation. In case of the complete visibility scenario, an offline analytical module is employed for computing mappings between process models of the requester and the provider. The PMA uses these precomputed mappings during the runtime mediation process. In case of the asymmetric scenario the PMA has to compute available mappings during the runtime, since the requester's process model is not available. The PMA uses external services (external data mediators) to deal with data incompatibilities and missing pieces of data together with planning techniques to find the appropriate mappings by combing available external data mediators. Since the PMA operates in dynamic open environments, it needs to interact with an appropriate discovery service. In our case, the PMA uses an external OWL-S Matchmaker service [19] to take care of the discovery of data mediators. Finally, the PMA integrates comprehensive monitoring and recovery mechanisms which were introduced in [28, 31].

4.1 Runtime mediation: The PMA Overview and Architecture

The architecture of the PMA is designed to support either the mediation in the strictly reactive mode in which all mappings must be computed only during the runtime, or in the mode when it uses mappings precomputed for example by the offline analytical module. Figure 4.1 shows an architecture of the PMA. The *server port* is used for interactions with the requester and the *client port* for interactions with the provider. The *client port* uses the OWL-S Virtual Machine (OVM) [18] to interact with the provider. The OVM is a generic OWL-S processor for execution of OWL-S services with built-in advanced features such as support for execution monitoring [30] and recovery [31]. Specifically, the OVM executes the Process Model of a given service by going through the Process Model while respecting the OWL-S operational semantics [4] and invoking individual services represented by atomic processes. During the execution, the OVM processes inputs and outputs of executed services, realizes the control and data flow of the composite Process Model, and uses the Grounding to invoke WSDL based web services when needed. Another instance of the OVM is used to execute external data mediation services if necessary.

The *Execution Monitor* is the central part of the PMA. It executes the mediation algorithm and

links all the other components together. Specifically, the *Execution Monitor* maintains the execution state and stores information received from the requester and provider in a *Knowledge Base*. Furthermore, the *Execution Monitor* interacts with the *Discovery Service* when new data mediators need to be found during the runtime. The *Plans Library* is used to store reconciliation plans (defined later in this section) that were either provided as an output by the offline analytical module, or that were used successfully in the previous mediation sessions. Also information about discovered data mediators is cached in the *Plans Library*.

The PMA tries to remember and reuse information that it gained from previous interactions with requesters and a discovery service. Only when no historical or precomputed information is available, the PMA explores the search space to find an appropriate mapping by simulating the execution of the provider's process model. Since interactions with external web services can fail, and also some choices made by the PMA can lead to failure, the PMA tries to recover from failures and possibly use backtracking if more mappings are available for a given execution state.

Definitions

Before describing details of analysis algorithms we introduce notions of execution state, reconciliation plans and the provider's ability to satisfy the requester.

Definition 1: The *execution state* for given requester's and provider's process models at a given time is a tuple $S = \langle V, F, RH, EH \rangle$, where V is a set of data (variables with their values and types) received from the requester, provider and external services, F is a set of valid expressions (e.g., produced as effects of service calls), RH is a requester's history (sequence of requests, i.e., atomic processes, executed by the requester), and EH is the execution history (sequence of services executed within one mediation session including atomic process of the provider and mediation services). We use a dot notation to access individual parts of a given state (e.g., $S.V$ stands for data available in S).

The execution state can be used during the mediation execution or during the simulated execution which is part of the off-line analysis. Since during simulation services are not executed, we cannot use values of inputs and outputs in the set V. Therefore, during the simulation we only names and types of available variables are stored in V while during the execution values are stored as well. For example, an expression $(Available(?userId, UserID))$ means that a variable $?userId$ of type $UserID$ is available and can be used as an input of some process. Similarly, preconditions and effects cannot be always fully evaluated during the simulation because of missing variable values. However, if an effect or a precondition does not depend on variable values, it can be evaluated, such as, e.g., in case of the effect $(LoggedIn?userId)$ of the *LoginStep2* atomic process from Figure 2.2.

We represent mappings between provider's and requester's processes as *reconciliation plans*. The *reconciliation plan* for a request r and execution state S specifies which actions in what order need to be executed to perform the translations between r and provider's process in the state S. Furthermore, the reconciliation plan can also represent what queries need to be asked to the discovery service to find new data mediators so that the plan can be executed.

Definition 2: The *reconciliation plan* M for a request r and execution state S is a tuple $M = \langle P, Q \rangle$, where P is a partially ordered plan consisting of atomic processes from $\mathcal{PA}$, internal mediation actions (such as built-in converters), external service calls (such as data mediators), and unbound *abstract processes*, and Q is a possibly empty set of *query templates*. The reconciliation plan $M = \langle P, Q \rangle$ is called *executable* if there are no abstract processes in P, and Q is empty. Otherwise, the plan is called *unbound*.

An *abstract process* in the plan is a place-holder for another plan which needs to be specified later. Abstract process is associated with some query template in Q which can be used for finding the executable plan (such as a sequence of atomic processes) that will be used in the place of the abstract process.

A query template q is a tuple $q = \langle I, O, E, S \rangle$ where I is a set of available inputs, O is a set of required outputs, E is a set of required effects and S is the execution state. We use query templates to represent discovery requirements for the discovery service (details in Section 5).

Next, we introduce the notion of the provider's *ability to satisfy* the requester. We start on the level of atomic processes and subsequently we extend the definition to requester's execution paths and to whole process models.

Definition 3: We say that the request r *can be satisfied by the atomic process* p in the execution state S and write $satisfied(r, p, S)$ if all inputs of p are either provided by r or are available in S, all preconditions of p are satisfied in S and all outputs and effects required by r are produced by p or are available in S. Formally, this can be expressed as the following conditions:

1. $\forall(?i_p, T_{i_p}) \in p.inputs \;\; \exists(?i_r, T_{i_r}) \in (r.inputs \cup S.V)$ such that T_{i_p} *subsumes* T_{i_r}
2. $\forall(?o_r, T_{o_r}) \in r.outputs \;\; \exists(?o_p, T_{o_p}) \in (p.outputs \cup S.V)$ such that T_{o_r} *subsumes* T_{o_p}
3. $\forall p_p \in p.preconditions \;\; S.F \models p_p$
4. $\forall e_r \in r.effects \;\; \exists e_p \in p.effects$ such that $S.F \models e_p \Rightarrow e_r$ or $S.F \models e_r$

Conditions 1–4 of Definition 3 are usual compatibility requirements (see e.g. [19]). In conditions 1 and 2 *subsumes* relation is a standard subsumption from description logic.

When incompatibilities do not allow a request r to be satisfied directly by provider's process p, a reconciliation plan can be used.

Definition 4: We say that the request r *can be satisfied by the reconciliation plan* M in the execution state S and write $satisfied(r, M, S)$ if M can be executed in S and if all outputs and effects required by r are produced by M, i.e. are available in the execution state S', where $S' = simulatePlan(M, S)$. The *simulatePlan* function returns a new execution state which is a result of simulating a given reconciliation plan in a given state.

The previous definition can be easily extended to requester's paths and whole process models. The requester's execution path can be satisfied by the provider's process model P if all steps of the path can be satisfied during a proper execution of both process models by possibly applying available translations.

Definition 5: We say that the requester's execution path $path_R = (r_1, \ldots, r_n)$ *can be satisfied by* the provider's process model P and write $satisfied(path_R, P)$ if there exists a set of reconciliation plans $\{M_1, \ldots, M_n\}$ and a sequence of execution states $(S_0, S_1, \ldots, S_n)$ such that $satisfied(r_i, M_i, S_i)$ holds, $S_i = simulatePlan(M_i, S_{i-1})$, for all $i = 1, \ldots, n$, where S_0 is an initial state of P and S_n is a final state of P.

Definition 6: If $satisfied(path_R, P)$ holds for all requester's execution paths $path_R$ of model R, we say that the requester's process model R *can be satisfied by the provider's process model* P and write $satisfied(R, P)$.

We use definitions of the *satisfied* predicates in our algorithms and in the discussion of their correctness. Finally, we define the *Next* function for getting the set of provider's atomic processes that can be executed at a given execution state.

Definition 7: Let *Next* be a function $Next : \mathcal{S} \rightarrow \mathbb{P}(\mathcal{PA})$, where $\mathcal{S}$ is the set of possible execution states, $\mathcal{PA}$ is the set of atomic processes in the provider's process model and $\mathbb{P}$ stands for a power set of a given set.

For example, in the running example in Figure 2.2, if the last executed provider's call was *SearchFlightOne*, the *Next* function would return a set of two atomic processes $\{ChooseDepFlght, ChooseItineraryOne\}$.

Runtime Mediation Procedure

The logic of the mediation procedure is the following. For each requester's request the PMA calls the *processRequest* procedure until requester or provider finishes successfully or the execution fails. Algorithm 1 shows high-level steps of the *processRequest* procedure. In this procedure, the least time consuming mediation options are considered first, and only if they fail other possibilities are considered.

Algorithm 1 Procedure *processRequest*

1. Receive a requester's atomic process call $requesterCall$ via the *server port*
2. Store inputs of $requesterCall$ in the state S and required results in the KB
3. Find the best available mediation actions:

3.1 **if** $A = \{a \mid a \in Next(S) \wedge satisfied(requesterCall, a, S)\} \neq \emptyset$ **then**

// Exact match: no data mediation needed

- **foreach** $a \in A$ **do**
  - **if** $execute(a, S)$ fails **then** $localRecover(a, S)$ and **continue**
  - **else return success**

3.2 **if** exists reconciliation plan P in the *Plans Library* for $requesterCall$ and S

// Reusing existing plan

- **if** $execute(P, S)$ fails **then** $localRecover(P, S)$ and **continue**
- **else return success**

3.3 **if** $reactiveMode \wedge reconciliationPlans = reconcileRequestCall(requesterCall, S) \neq \emptyset$ **then**

// Planning was used to find the mapping

- **foreach** executable reconciliation plan $P \in reconciliationPlans$ **do**
  - **if** $execute(P, S)$ fails **then** $localRecover(P, S)$ and **continue**
  - **else** store P in the *Plans Library* and **return success**
- **foreach** unbound reconciliation plan $P \in reconciliationPlans$ **do**
  // If no directly executable plan found, use discovery service
  - **if** $P' = bindPlan(P)$ succeeds **then**
    - **if** $execute(P')$ fails **then** $localRecover(P', S)$ and **continue**
    - **else** store P' in the *Plans Library* and **return success**

3.4 **if** $globalRecover(S)$ fails **then** *// Nothing worked, undo and pick a another branch*

- **return failed**

After the PMA receives the request it first tries to match it to some provider's atomic process available in the given execution state (step 3.1).

If no such process exists or execution of all of them fails, reconciliation plans from the *Plans Library* are considered (step 3.2). Notice, that plans can be either remembered from previous mediation sessions or pre-computed from the off-line analysis phase.

If the PMA is running in the reactive mode, as the next step (3.3), the planning algorithm is used (*reconcileRequestCall* procedure, for details see Section 4.2) that tries to find new reconciliation plans by combining known data mediators or proposing queries to the discovery service which would allow to bridge gaps (mismatches) identified during the reconciliation. If some executable plan is found and successfully executed it is stored in the *Plans Library*. Otherwise, if only unbound plans are found, the discovery service needs to be used to bind the plan in the *bindPlan* procedure (see details in Section 5).

Finally, if none of the mediation alternatives succeeds, there still might be a chance that in some previous phase a wrong branch in the provider's process model was taken which does not allow the mediation to be finished while another branch might work. The *globalRecover* procedure in step 3.4 tries to deal with such a situation.

The $execute(P, S)$ procedure executes the reconciliation plan P by executing every action $a \in P$. If a is a provider's process the client port is used, while if a is a data mediator the OVM for external services is used.

4.2 Computing Reconciliation Plans In PMA (Open World)

The purpose of the runtime reconciliation procedure for a given requester's request r and an execution state S is to find possible reconciliation plans that can be used to reconcile the mismatches between the request r and the current state of the provider's process model. The reconciliation procedure uses

a planning algorithm that tries to combine known data mediators to find necessary transformations. If no combination of known data mediators can be found, the reconciliation procedure produces an unbound plan associated with query templates which can be used later to discover new data mediators and to bind the plan.

In principle, the reconciliation procedure transforms missing pieces of information (inputs, outputs, preconditions and effects) into goals that need to be satisfied. Then a classical backward chaining planning algorithm is employed with data mediators and providers atomic processes used as planning operators. In order to guarantee timely termination of the planning algorithm, the maximal length of the plan is constrained externally. During the planning, the operations are only simulated (services are not executed) with respect to the initial execution state S.

Algorithm 2 $reconcileRequestCall(r, S)$, r a request call, S execution state

1. **Initialize:**
   - **foreach** $i \in r.inputs$ **do** add $(Available\ i.name\ i.type)$ $(Value\ i.name\ i.value)$ to S
   - **foreach** $o \in r.outputs$ **do** add $(RequesterGoal\ (Available\ o.name\ o.type))$ to S
   - **foreach** $effect \in r.effects$ **do** add $(RequesterGoal\ effect)$ to S
   - $plans = \emptyset$
2. **foreach** $p \in Next(S)$ **do** $reconcileAtomicProcess(r, p, S, plans, \emptyset, maxPlanLength)$
3. **return** $plans$

Algorithm 2 takes care of the planner initialization (step 1) and starting the reconciliation for every provider's atomic process available in S (step 2). The core of the reconciliation algorithm is performed by the *reconcileAtomicProcess* procedure displayed in Algorithm 3. This procedure is trying to find a plan for reconciliation of request r and process p. First, it must guarantee that all inputs and preconditions of p are available (step 2), and afterwards that all outputs and effects required by r were produced (step 3). In both cases the *solveGoals* procedure implementing a backward chaining algorithm is tried first to achieve missing goals by means of known data mediators. If *solveGoals* does not succeed (i.e., some goals cannot be satisfied), the query template q for the discovery service is suggested that would allow a discovery of new data mediators needed for finishing the plan.

4.3 Computing Mappings in the Complete Visibility Scenario (Semi-open Environment)

In the complete visibility scenario both process models are completely visible. Such a scenario is usual for example in semi-open corporate environments. From the point of view of process mediation several characteristics of semi-open environments are important:

1. Components are not controlled by one authority.
2. Components can be added to the system, removed or replaced dynamically.
3. There can be several interchangeable components (from several contractors) that solve the same problem.
4. The registration of a new component to the system can be controlled.
5. Typically some level of trust among contractors is necessary which allows us to assume that all components can publish descriptions of their interaction protocols to allow interoperability with other components.

These characteristics allow us to perform the offline analysis to find possible mappings between provider's and requester's process models, or to identify incompatibilities that cannot be reconciled with given set of available data mediators and external services. Mappings computed during the offline analysis are later used in the PMA for the runtime mediation.

Execution paths analysis approach

The off-line analysis to find possible mappings between provider's and requester's process models can be achieved by exploring possible sequences of steps (execution path) that the requester can execute. *Requester's execution path* is any sequence of atomic processes which can be called by the requester in accordance with its process model, starting from the process model first atomic process and ending in

Algorithm 3 $reconcileAtomicProcess(r, p, S, plans, plan, maxPlanLength)$, r a request call, p provider's atomic process, S execution state, $plans$ a set of all plans, $plan$ current plan, $maxPlanLength$ maximal allowed length of generated plans

1. **if** $|plan| \geq maxPlanLength$ **then return**
2. **Reconcile inputs and preconditions of** p**:**
   - **if** ($\forall i \in p.inputs$ available in S) and ($\forall prec \in p.preconditions$ satisfied in S) **then**
     - simulate p (add outputs and effects of p to S); add p to $plan$
   - **else**
     - $Goals$= transform missing inputs of p and unsatisfied preconditions of p to goals
       // e.g., missing toCode input =¿ $(Goal(Available\ toCode\ \ AirportToCode))$
     - **if** $solveGoals(Goals, S, plan, maxPlanLength)$ **then** simulate p; add p to $plan$
       else
       - create query template $q = \langle I, O, E, S\rangle$, and corresponding abstract process p_q where $I =$ $r.inputs$, O =missing inputs of p, E =unsatisfied preconditions of p
       - simulate p_q in S; add q and $p_q$to $plan$
3. **Reconcile outputs and effects of** r**:**
   - **if** ($\forall o \in r.outputs$ available in S) and ($\forall effect \in p.effects$ satisfied in S) **then**
     - add $plan$ to $plans$ *// the reconciliation of* r *and* p *is finished*
   - **else**
     - $Goals$= transform missing outputs & effects to goals
       // e.g., $(RequesterGoal\ goal)$ *=¿* $(Goal\ goal)$
     - **if** $solveGoals(Goals, S, plan, maxPlanLength)$ **then** add $plan$ to $plans$
       else
       - *duplicate the plan and continue with another process in the providers model*
         - $newPlan = plan$
         - **foreach** $a \in Next(S)$ **do**
           $reconcileAtomicProcess(r, a, S, plans, newPlan, maxPlanLength)$
       - *create an unbound plan*
         - create query template $q = \langle I, O, E, S\rangle$, and corresponding abstract process p_q where I =outputs produced by $plan$, O =missing outputs of r, E =unsatisfied effects of r
         - add q and $p_q$to $plan$; add $plan$ to $plans$

one of the last atomic processes of the process model. An atomic process is last in the process model if there is no next atomic process that can be executed after it (respecting the control constructs, as e.g. loops).

Since any of all possible requester's execution paths can be chosen by the requester, we need to show that each requester's execution path can be mapped into the provider's process model with help of intermediate translations represented by *data mediators* or *built-in conversions* . If there exists a possible requester's execution path which could not be mapped to any part of the provider's process model, we would know that if this path were chosen, the mediation would fail. Thus the existence of a mapping for each possible requester's execution path is a necessary precondition of successful mediation. Indeed, it is only a necessary condition of successful process mediation for the following reason. Since the possible mappings are being searched before actual execution, some of them can turn out not to work during execution (e.g., because of failing preconditions of some steps). Still, by analyzing requester's execution paths and trying to find mappings for them, we can partially answer the question of mediation feasibility.

Finding possible mappings means to explore the search space generated by combining allowed execution paths in the provider's process model with available translations (data mediators in our case). We explore the search space by simulating the execution of the provider's process model with possible backtracking if some step of the requester's path cannot be mapped or if more mappings are possible. During the simulation, data mediators are used to reconcile possible mismatches.

The following procedure provides a top-level view of our offline analysis approach to compute mappings between process of the requester and the provider:
1. Generate requester's paths: based on the process model of the requester, possible requester's paths are generated (see Section 4.3)
2. Filter out those requester's paths that need not be explored: as the result we get the *minimal set of requester's paths.* (see Section 4.3)
3. Find all appropriate mappings to the provider's process model for each requester's path from the minimal set of paths and store them in the *mappings repository*: if for a path no mapping is found, user is notified with pointing out the part of the path for which the mapping was not possible[5]. (see Section 4.3)

Generating the minimal set of requester's execution paths

When generating requester's execution paths we potentially have to deal with combinatorial explosion caused by chains of branching in the requester's process model. We want to find out what reconciliation actions are available or necessary in given state of execution which depends on possible combinations of available variables and valid expressions in this state. Because the current state depends on actions performed preceding this state, we might be in principle interested in every possible requester's path. In [29] we describe heuristics for pruning those paths that provide no additional information. The path pruning also reduces the number of requester's execution paths for which appropriate mappings to the provider's process model need to be found.

Finding mappings for the requester's path

In order to find all the mappings for a given requester's path we simulate the execution of the provider's process model and try to map each step of the requester's path to some part of the provider's model (atomic process or several atomic processes) with help of *data mediators.* The mappings are constructed during the simulation and are represented as a sequence of reconciliation plans (as defined in Section 4.1) that the PMA can execute during the runtime mediation (see Figure 4.2 for an example of a mapping).

Algorithm 4 presents the recursive *reconcileSequence* method which constructs mappings for one requester's execution path. This method simulates step by step all requests in the given requester's execution path and for each of them finds all possible reconciliation plans. After executing the *reconcileSequence* method, only all those reconciliation plans leading to successful mediation are

[5]At this point service discovery could be used to find a service capable of resolving the mismatch.

stored in the *plansLibrary*. We use the same methods as those introduced for the runtime reconciliation planning. In addition to that we also introduced method *simulatePlan* that returns a new execution state which is a result of simulating a given reconciliation plan in a given state. Compared to the runtime reconciliation planning the main difference is that in the offline analysis we are not constrained by the time. Therefore, we do not constraint the maximal length of reconciliation plan during the offline analysis in the *reconcileAtomicProcess* method. This gives us a guarantees of completes.

Algorithm 4 $reconcileSequence(stepsSequence, S, plansLibrary)$, *stepsSequence* a requester's execution path, S execution state, *plansLibrary* a reconciliation plans library

1. **if** $|stepsSequence| = 0$ **then return** *true* // *termination condition: all steps reconciled*
2. **Intitialization:** $request \leftarrow$ remove first request from *stepsSequence*; $returnStatus \leftarrow false$
3. $newPlans \leftarrow reconcileRequestCall(request, S)$
4. **if** $newPlans = \emptyset$ **then return** $false$
5. **foreach** $plan \in newPlans$
   - $newState \leftarrow simulatePlan(plan, S)$
   - **if** $reconcileSequence(stepsSequence, newState, plansLibrary)$ **then**
     - store $(request, S, plan)$ in *plansLibrary*
     - $returnStatus \leftarrow true$
6. **return** *returnStatus*

4.4 Algorithms Properties

The correctness and the completeness of introduced algorithms can be defined in terms of the $satisfied$ predicates defined in Section 4.1. We discuss these properties with respect to the single request r reconciliation (i.e., $satisfied(r, M, S)$ property in Definition 4), with respect to the requester's path $path_R$ reconciliation (i.e., $satisfied(path_R, P)$ property in Definition 5), and with respect to the whole process models reconciliation (i.e., $satisfied(R, P)$ property in Definition 6).

For the complete visibility scenario the *reconcileAtomicProcess* procedure defined in Algorithm 3 is sound and complete with respect to the $satisfied(r, M, S)$, i.e., for every executable reconciliation plan the predicate $satisfied(r, M, S)$ holds and the procedure finds all executable reconciliation plans satisfying this predicate. The soundness and the completeness are derived from the fact that the *reconcileAtomicProcess* procedure is designed to make the $satisfied(r, M, S)$ predicate true by employing the planning procedure *solveGoals* which is sound and complete (basically it uses a backward chaining best first search algorithm for achieving goals

Consequently, the *reconcileSequence* procedure is sound and complete with respect to the $satisfied(path_R, M)$ (all possible plans are considered). Finally, the whole mediation procedure as described in Section 4.3 is sound and complete with respect to the $satisfied(R, P)$ predicate, since the set of requester's paths generated as described in Section 4.3 is minimal and complete. It is important to notice, that in all cases soundness and completeness are defined with respect to static mismatches only (mis_{static}) while the runtime mismatches are not considered.

For the asymmetric scenario, the *reconcileRequestRuntime* procedure is sound, but incomplete with respect to $satisfied(r, M, S)$ because the maximal length of searched reconciliation plans is constrained. We are aware that constraining the plan length externally is not an ideal solution and we are working on an anytime version of the runtime reconciliation algorithm. With respect to the reconciliation of the requester's path $path_R$, i.e., $satisfied(path_R, P)$, the *reconcileRequestRuntime* might select a wrong solution as we discussed in the previous section, and thus it is not guaranteed to be correct. This problem cannot be avoided since the runtime mediation algorithm does not have enough information to be able to make the correct choice. The algorithm becomes correct only if we assume that all reconciliation plans can be undone. This result has no negative effect on the correctness of the offline algorithm.

Another important property of generating the process mediator protocol is whether it guarantees a *deadlock free* communication of the requester and the provider (assuming that both process models

The requester's path: Login, SearchFlight, ChooseDepFlight, ChooseReturnFlight, ...	
A mapping discovered for first two steps of the requester's path:	
I. Reconciliation plan for *Login*	*1. requester-Login* s1-userID s1-password *2. provider-LoginStep1* s1-user sessionID *3. provider-LoginStep2* s1-password sessionID logResult *4. mediator-prepare-to-send* logResult *5. mediator-send*
II. Reconciliation plan for *SearchFlight*	*1. requester-SearchFlight* s2-from s2-to s2-depTime s2-retTime *2. external-AirportCityToCode* s2-from apt-code-gener1 *3. mediator-explicit-down-casting* apt-code-gener1 AirportToCode *4. external-AirportCityToCode* s2-to apt-code-gener2 *5. mediator-explicit-down-casting* apt-code-gener2 AirportToCode *6. external-USTimeToISO* s2-depTime iso-time-gener1 *7. mediator-explicit-down-casting* iso-time-gener1 ISODepTime *8. external-USTimeToISO* s2-retTime iso-time-gener2 *9. mediator-explicit-down-casting* iso-time-gener2 ISODepTime *10. provider-SearchReturnFlight* apt-code-gener1 apt-code-gener2 iso-time-gener1 iso-time-gener2 flights flghtCount *11. mediator-prepare-to-send* flights *12. mediator-prepare-to-send* flghtCount *13. mediator-send*

Figure 4.2: Example solution for a requester's path

are deadlock free). According to [7] a communication is deadlock free if it ends with both protocols in final states, or the collaboration can continue at any time. In our case, a deadlock in communication of the requester and the provider can occur only in a situation when one of the partners is waiting for a message (or data) that the other partner has not sent yet and is not going to send (e.g., because the seconds partner is also waiting for data which the first partner has not provided). However, such a situation is identified by the *reconcileAtomicProcess* procedure at some point as a missing piece of information (input, output, precondition or effect). As an outcome, the problem can either be resolved if some of external service is able to produce the required piece of information, or the analysis identifies such a situation as the one which cannot be resolved and the user is informed about the problem. This means that the algorithms guarantee a deadlock free communication between the requester and the provider.

Example mapping

Figure 4.2 shows part of one mapping generated for a requester's execution path that can be executed by a requester as defined in Figure 2.2a in Section 2.2. The mapping was generated for a provider's process model defined in Figure 2.2b. This example assumes that we have provided the system with the *AirportCityToCode* external web service for translating instances of *City* to instances of *AirportCode*, and the service *USTimeToISO* for translating between US and ISO time formats. Each step name is prefixed by *requester*, *provider* and *external* to indicate to which component it is related. Requester's steps show names of inputs parameters, while for the provider, translators and external services also output variables are included. This example also illustrates implicit up-casting of types and explicit down-casting which is enforced by the fact, that *AirportCityToCode* and *USTimeToISO* are defined to work with more generic types than those provided by requester and requested by the provider. Due to the requirement for explicit down-casting, the user is prompted whether the chosen casting is allowed or not. In this example all the castings are allowed. See [22] for details on analyzing casting operations for ontology classes.

5 Data Mediators Discovery

Discovering new data mediators requires two questions to be answered. First, the specific *discovery requirement* needs to be gained, and second, this requirement must be translated into *concrete queries* that will be sent to the discovery service. We use query templates defined in Section 4.1 to capture discovery requirements. The query template is derived from the specific mismatch between

the reconciled requester's request r and the process p encountered in the execution state S.

Based on the query template, concrete queries which will be sent to the discovery service need to be formulated in the *bindPlan* procedure. A straightforward idea would be to use the query template as it is. However, the vast majority of discovery services implement a matching algorithm in which only one service is considered as a suitable candidate satisfying a service request while service combinations are not allowed [19]. Such an assumption makes sense when standalone services need to be discovered. In our case, however, this assumption is too restrictive since we are not necessarily looking for one service only. On the contrary, often in the mediation scenario one specific gap identified by the process mediation algorithm can be bridged only by using a combination of several services.

Consider, for example, a situation in the running example in Figure 2.2 in which the *reconcileAtomicProcess* procedure (in step 2) is trying to find mapping between the *SearchFlight* request and the *SearchFlightOne* process. Assuming that no external services are known to the PMA yet, the query template would have the following form: $q = \langle I = \{$ *from - FromCity, to - ToCity, depTime - USDepTime, retTime - USRetTime*$\}, O = \{$*from - AirportFromCode, to - AirportToCode, depTime - ISODepTime, retTime - ISORetTime* $\}, E = \emptyset, S\rangle$. The query template q expresses the fact that inputs of the requester's *SearchFlight* call are the most likely candidates that can be used as input data for possible data mediators, while the inputs of the provider's *SearchFlightOne* call are the most likely output candidates that need to be produced. Finally, the state S included in q can be used to access all remaining data available at the given execution state. Clearly, it is very unlikely that there would ever exist one single service satisfying such a requirement. However, if combinations of services are allowed to be matched, the chances of a successful match are much higher. In our particular case, a combination of external services *AirportCityToCode* and *USTimeToISO* can be used as a match bridging the gap as shown in Figure 2.2.

To deal with the problem the matching assumptions need to be relaxed to allow a combination of several services as an acceptable match for a given service request. Benatallah et. al. in [5] propose an approach that allows a combination of several services to satisfy the service request. Their algorithm based on request rewriting guarantees that an optimal combination covering the request will be found but it is NP-hard. We have decided to go a similar direction by allowing the combination of services satisfying the request to be returned as a relevant match — we call it a *combined match*. In the combined match we do not strictly insist on optimality in order to prevent hard computations. We prefer the coverage instead, since we assume that, if needed, the composition or planning algorithms can find the optimal combination in next steps after discovery is done.

When answering a combined match query, the discovery service first finds a set of services that together produce the required outputs and effects (i.e., any service producing some of required outputs or effects is a good candidate). In the next step, out of these candidates, if more candidates are available producing the same outputs, those are preferred that use only inputs specified in the query. Since no real composition is involved such an approach is very efficient (assuming that appropriate index structures were precomputed during the service registration with the discovery service). We implemented the combined match as an extension to the OWL-S Matchmaker. For details see [27].

Assuming the discovery service supports a combined match, the PMA discovers new mediators in the *bindPlan* procedure in two steps as shown in Algorithm 5. It starts with an exact query request in the form of the query template (step 1.1). If some service matching the query is returned, PMA just binds it in the plan in the place of the corresponding abstract process. Otherwise, the combined match query in the same form is sent to the discovery service (step 1.2). Returned data mediators are transformed into planning operators and the *solveGoal* planning method is re-run with the state S saved in the query template. The produced plan is plugged in the place of the abstract process.

6 Related Work

The work in [32] provides a conceptual underpinning for automatic mediation. In [13] Cimpian at. al. solve the runtime mediation between two WSMO based processes. Besides structural transformations (e.g., change of message order) also data mediators can be plugged into the mediation process, however, recovery and discovery are not addressed at all. Aberg et. al. [2] describe an agent called sButler for

Algorithm 5 $bindPlan(M)$, $M = \langle P, Q \rangle$ unbound reconciliation plan

1. **foreach** $p_q \in P$, p_q abstract process, $q = \langle I, O, E, S \rangle$ query template associated with p_q **do**

   1.1 **if** $mediators = askDiscoveryExact(q) \neq \emptyset$ **then**
   replace p_q in P with best matching $a \in mediators$

   1.2 **elseif** $mediators = askDiscoveryCombined(q) \neq \emptyset$ **then**

   - transform $mediators$ to planning operators and add them to *Plans Library*
   - $Goals$ = transform q to goals; $newPlan = \emptyset$
   - $solveGoals(Goals, S, plan, maxPlanLength)$
   - replace p_q in P with $newPlan$

   1.3 **else return failed**

2. **return** $M' = \langle P, \emptyset \rangle$

mediation between organizations' workflows and semantic web services. The mediation is more similar to brokering, i.e., having a query or requirement specification, the sButler tries to discover services that can satisfy it. The requester's process model is not taken into considerations. OWL-S broker [20] also assumes that the requester formulates its request as query which is used to find appropriate providers and to translate between the requester and providers. In [11] and [15] authors describe the IRS-III broker system based on the WSMO methodology. IRS-III requesters formulate their requests as goal instances and the broker mediates only with providers given their choreographies (explicit mediation services are used for mediation). Brambilla at. al. [6] apply a model-driven approach based on WebML language. Mediator is designed in the high-level modeling language which supports semi-automatic elicitation of semantic descriptions in WSMO. In [22], data transformation rules together with inference mechanisms based on inference queues are used to derive possible reshapings of message tree structures. An interesting approach to translation of data structures based on solving higher-order functional equations is presented in [9] while [10] argues for published ontology mapping to facilitate automatic translations.

7 Conclusions and Further Work

In this paper we dealt with process mediation mechanisms of two OWL-S process models operating in dynamic open environments. We described algorithms based on the analysis of provider's and requester's process models for finding mappings between them, and for performing runtime mediation. The main advantage of our approach, besides enabling the interoperability of requesters and providers, is the capability of the process mediation agent to operate in conditions where failures and changes of the environment must be taken into account. Due to recovery mechanisms employing dynamic recovery and built-in heuristics, the PMA is able to recover if possible and its performance degrades gracefully when the environment changes or no simple recovery is possible. Compared to other relevant recent work, our approach is unique in using ontologies for service specification and matching together together with behavioral specifications of protocols and automatic synthesis of the mediator process (compare e.g., with [8]). Also, compared to other approaches (such as the one of [13]) our dynamic discovery of external (data mediation) services is unique. However, our experiments pointed out some issues that need to be addressed. Namely the efficiency of our offline analysis algorithm in case of presence of more branching points in the provider's process model, and the issue with identifying the right reconciliation plan if more plans are available.

In the paper, we focused on the process mediation problem itself and we did not discuss many practical issues such as security, hosting of the PMA, etc. Let us discuss briefly the hosting question. In general, depending on the particular application domain, the PMA can be deployed either as part of the provider's infrastructure, requester's infrastructure or as part of the middle layer in between. In all cases, there are very strong incentives for hosting the PMA related to achieving interoperability. Hosting the PMA on the side of provider might allow new partners to interact with the provider. From the requester's perspective, hosting the PMA makes a good sense when some application needs

to be extended by adding a new provider or when an existing provider needs to be replaced by a new one. In such a case the PMA can be used on the requester's side as a smart adapter to bridge the possible incompatibilities. Finally, the PMA can find its role in the infrastructure of enterprises such as mobile operators which provide access to services of third parties to their final customers.

Bibliography

[1] W.M.P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data and Knowledge Engineering*, 47(2):237–267, 2003.

[2] Cecile Aberg, , Patrick Lambrix, Juha Takkinen, and Nahid Shahmehri. sButler: A Mediator between Organizations' Workflows and the Semantic Web. The World Wide Web Conference workshop on Web Service Semantics: Towards Dynamic Business Integration, May 2005.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business Process Execution Language for Web Services, Version 1.1. 2003.

[4] Anupriya Ankolekar, Frank Huch, and Katia P. Sycara. Concurrent semantics for the web services specification language DAML-S. In Farhad Arbab and Carolyn L. Talcott, editors, *COORDINATION*, volume 2315 of *Lecture Notes in Computer Science*, pages 14–21. Springer, 2002.

[5] Boualem Benatallah, Mohand-Said Hacid, Christophe Rey, and Farouk Toumani. Request rewriting-based web service discovery. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2003.

[6] Marco Brambilla, Irene Celino, Stefano Ceri, Dario Cerizza, Emanuele Della Valle, and Federico Michele Facca. A software engineering approach to design and development of semantic web service applications. In *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2006.

[7] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983.

[8] Antonio Brogi and Razvan Popescu. Automated generation of BPEL adapters. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2006.

[9] Mark Burstein, Drew McDermott, Douglas R. Smith, and Stephen J. Westfold. Derivation of glue code for agent interoperation. *Autonomous Agents and Multi-Agent Systems*, V6(3):265–286, May 2003.

[10] Mark H. Burstein and Drew V. McDermott. Ontology translation for interoperability among semantic web services. *The AI Magazine*, 26(1):71–82, 2005.

[11] Liliana Cabral, John Domingue, Stefania Galizia, Alessio Gugliotta, Vlad Tanasescu, Carlos Pedrinaci, and Barry Norton. IRS-III: A broker for semantic web services based applications. In Isabel F. Cruz, Stefan Decker, Dean Allemang, Chris Preist, Daniel Schwabe, Peter Mika, Michael Uschold, and Lora Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 201–214. Springer, 2006.

[12] Erik Christensen, Francisco Curbera, and Greg Meredith Sanjiva Weerawarana. Web Services Description Language, 2001.

[13] Emilia Cimpian and Adrian Mocan. WSMX process mediation based on choreographies. In *Business Process Management Workshops*, pages 130–143, 2005.

[14] Ana Karla Alves de Medeiros, Carlos Pedrinaci, Wil M. P. van der Aalst, John Domingue, Minseok Song, A. Rozinat, Barry Norton, and Liliana Cabral. An outlook on semantic business process mining and monitoring. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops (2)*, volume 4806 of *Lecture Notes in Computer Science*, pages 1244–1255. Springer, 2007.

[15] John Domingue, Stefania Galizia, and Liliana Cabral. Choreography in IRS-III - coping with heterogeneous interaction patterns in web services. In Yolanda Gil, Enrico Motta, V. Richard Benjamins, and Mark A. Musen, editors, *International Semantic Web Conference*, volume 3729 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2005.

[16] Thomas Eiter, Esra Erdem, and Wolfgang Faber. On reversing actions: Algorithms and complexity. In Manuela M. Veloso, editor, *IJCAI*, pages 336–341, 2007.

[17] Joel Farrell and Holger Lausen. Semantic annotations for WSDL and XML schema, 2007. http://www.w3.org/TR/sawsdl/.

[18] Massimo Paolucci, Anupriya Ankolekar, Naveen Srinivasan, and Katia P. Sycara. The DAML-S virtual machine. In Dieter Fensel, Katia P. Sycara, and John Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2003.

[19] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic matching of web services capabilities. In Ian Horrocks and James A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2002.

[20] Massimo Paolucci, Julien Soudry, Naveen Srinivasan, and Katia Sycara. A broker for owl-s web services. In David Martin Lawrence Cavedon, Zakaria Maamar and Boualem Benatallah, editors, *Extending Web Services Technologies: The Use of Multi-Agent Approaches*. Kluwer, 2005.

[21] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubn Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77 – 106, 2005.

[22] Bruce Spencer and Sandy Liu. Inferring data transformation rules to integrate semantic web services. In *International Semantic Web Conference*, pages 456–470, 2004.

[23] Katia Sycara, Massimo Paolucci, Anupriya Ankolekar, and Naveen Srinivasan. Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, 1 (1):27–46, 2004.

[24] The OWL Services Coalition. *Semantic Markup for Web Services (OWL-S)*. http://www.daml.org/services/owl-s/1.1/.

[25] Paolo Traverso and Marco Pistore. *Automated Composition of Semantic Web Services into Executable Processes*. 2004.

[26] Roman Vaculín, Huajun Chen, Roman Neruda, and Katia Sycara. Modeling and discovery of data providing services. In *2008 IEEE International Conference on Web Services*, pages 1032–1039. IEEE Computer Society, September 23-26 2008.

[27] Roman Vaculín, Roman Neruda, and Katia Sycara. Towards extending service discovery with automated composition capabilities. In *The 6th IEEE European Conference on Web Services*, pages 3–12. IEEE Computer Society, November 12-14 2008.

[28] Roman Vaculín and Katia Sycara. Specifying and monitoring composite events for semantic web services. In *The 5th IEEE European Conference on Web Services*, pages 87–96. IEEE Computer Society, November 26-28 2007.

[29] Roman Vaculín and Katia Sycara. Towards automatic mediation of OWL-S process models. In *2007 IEEE International Conference on Web Services*, pages 1032–1039. IEEE Computer Society, July 9-13 2007.

[30] Roman Vaculín and Katia Sycara. Semantic web services monitoring: An OWL-S based approach. In *41st Hawaii International Conference on System Sciences*, page 313, Waikoloa, Hawaii, January 7-10 2008. IEEE Computer Society Press.

[31] Roman Vaculín, Kevin Wiesner, and Katia Sycara. Exception handling and recovery of semantic web services. In *Fourth International Conference on Networking and Services*, pages 217–222. IEEE Computer Society Press, 2008.

[32] Gio Wiederhold and Michael R. Genesereth. The conceptual basis for mediation services. *IEEE Expert*, 12(5):38–47, 1997.