The process mediation framework for semantic web services

Roman Vaculín* and Roman Neruda

Institute of Computer Science
Academy of Sciences of the Czech Republic, Czech Republic
E-mail: vaculin@cs.cas.cz
E-mail: roman@cs.cas.cz
*Corresponding author

Katia Sycara

The Robotics Institute
Carnegie Mellon University, USA
E-mail: katia@cs.cmu.edu

Abstract: The ability to deal with the 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 the semantically annotated process models of the service requester and service provider. We propose an Abstract Process Mediation Framework (APMF) 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: (1) when the mediation process has complete visibility of the process model of the service provider and service requester (*complete visibility scenario*) and (2) 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 planning and semantic reasoning with the discovery of appropriate external services such as data mediators. Finally, the Process Mediation Agent (PMA) is introduced, which realises an execution infrastructure for runtime mediation.

Keywords: process mediation; OWL-S; semantic web services; adapter synthesis.

Reference to this paper should be made as follows: Vaculín, R., Neruda, R. and Sycara, K. (2009) 'The process mediation framework for semantic web services', *Int. J. Agent-Oriented Software Engineering*, Vol. 3, No. 1, pp.27–58.

Biographical notes: Roman Vaculín is a PhD candidate in Theoretical Computer Science at Charles University, Czech Republic. He works in the Institute of Computer Science of the Academy of Sciences of the Czech Republic. From 2005 to 2007, he worked in the Robotics Institute at Carnegie Mellon University, USA initially as a Fulbright scholar and later as a visiting research scholar. His research interests are mainly in the field of semantic web services, focusing on areas such as service discovery, composition, mediation and monitoring.

Roman Neruda received his PhD in Theoretical Computer Science in 1998 from the Academy of Sciences of the Czech Republic in Prague. He works as a Researcher in the Institute of Computer Science of the Academy of Sciences of the Czech Republic, and as a Lecturer at Charles University, also in Prague. His research interests include intelligent agents, computational intelligence and hybrid learning algorithms.

Katia Sycara is a Research Professor in the School of Computer Science at Carnegie Mellon University, USA and Director of the Advanced Technology Lab. She holds the Sixth Century Chair in Computing at the University of Aberdeen, UK. She has a PhD from Georgia Institute of Technology, USA and an Honorary Doctorate from the University of the Aegean, Greece. She is a Fellow of the IEEE, a Fellow of AAAI and the recipient of the ACM Agents Research Award. She has authored more than 350 technical papers and has been the principal investigator of multimillion-dollar research awards. She was founding Editor-in-Chief of the *Journal of Autonomous Agents and Multiagent Systems* and is on the editorial board of six other journals.

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. Individual web services serve as building blocks in process models that prescribe control and data flows and thus define functionalities of more complex applications. With changing business needs, processes could get reconfigured, replaced by new providers, or additional components and services may need to be added. As a result, 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 (Christensen *et al*., 2001) allows to declaratively describe operations, the format of messages, and the data structures that are used to communicate with a web service. BPEL4WS (Andrews *et al*., 2003) 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 (Farrell and Lausen, 2007), OWL-S[1] and WSMO (Roman *et al*., 2005) strive to enrich syntactic specifications with rich semantic annotations to further facilitate flexible dynamic web services invocation, discovery and composition (Sycara *et al*., 2004). This is still not enough for achieving service interoperability in dynamic environments, where service providers and requesters do not typically share the same data models and interaction protocols. Therefore, incompatibilities on different levels among service providers and requesters (Vaculín and Sycara, 2007b) may result. Additionally, due to unpredictable changes of business needs, existing web services change very often.

In this paper we address the problem of *automatic mediation* of process models consisting of semantically annotated web services in order to achieve interoperability. We focus 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 that guide the exchange of messages of provider and requester during service invocation. We analyse the problem of process mediation and we propose an *Abstract Process Mediation Framework* (*APMF*) (see Figure 4) that identifies the key functionalities needed for successful process mediation in dynamic environments. In this paper, we focus only on the process mediation, while we discuss the other functional areas such as data mediation, monitoring and recovery only very briefly.

We take into consideration the environment within which mediation takes place. Specifically, we have identified *visibility of involved process models* as an important factor with a direct impact on design of effective process mediation architectures and algorithms. With respect to visibility, for a given process model two extreme cases are:

1 *white box visibility* – the complete process model is visible, including the whole interaction protocol

2 *black box visibility* – only definitions of available operations (*i.e.*, atomic processes in the OWL-S terminology) are visible, while the interaction protocol is not disclosed.

By applying specific visibility options to the mediation scenario of one requester and one provider, we obtain four possible mediation configurations (see Table 1). We analyse these scenarios and propose mediation techniques for two cases:

1 when the mediation process has complete visibility of the process model of the service provider and the service requester (*complete visibility scenario*)

2 when the mediation process has visibility only of the process model of the service provider but not the service requester (*asymmetric scenario*).

In both cases, our solution combines planning to generate appropriate mappings between processes, and semantic reasoning (specifically, matching of services based on their semantically annotated specifications) with the discovery of appropriate external services (serving, *e.g.*, as data mediators) and suitable recovery techniques. 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 rest of the paper if structured as follows. In Section 2 we give a brief overview of OWL-S. In Section 3 we introduce the process mediation problem along with a running example. Section 4 defines the abstract process mediation framework and gives an overview of means of process mediation depending of visibility of participating process models. In Section 5 we describe algorithms for process mediation in the complete visibility scenario (Section 5.1) and for the asymmetric scenario (Section 5.2) together with the analysis of these algorithms (Section 5.3) and an execution infrastructure in the form of a process mediation agent (Section 5.4). Section 6 covers briefly discovery of external services in the process mediation context and Section 7 provides an experimental evaluation of mediation algorithms. Finally, Section 8 overviews the related work and Section 9 concludes the paper.

2 OWL-S

OWL-S[1] is a semantic web services description language, expressed in OWL (Bechhofer *et al*., 2004). OWL-S covers three areas: web services capability-based search and discovery, specification of service requester and service provider interactions, and service execution. The *Service Profile* describes what the service does in terms of its capabilities and it is used for discovering suitable providers, and selecting among them. The *Process Model* specifies (a) how the service works and (b) clients can interact with the service by defining the requester-provider interaction protocol. OWL-S process model pertains mainly to describing service providers. However, its constructs can be used to describe the requester as well. The *Grounding* links the Process Model to the specific execution infrastructure, for example, it maps OWL-S processes to WSDL (Christensen *et al*., 2001) operations and allows for sending messages in SOAP.[2]

For the purposes of the process mediation the Process Model is the most important part. The Process Model specifies ways of how clients can interact with the service. The elementary unit of the process model 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 can be combined into composite processes by using control constructs such as sequence, any-order, choice, if-then-else, split, loops, *etc*.

Processes are specified by means of their Inputs, Outputs, Preconditions, and Effects (IOPEs). We use a dot syntax to access IOPEs of a process, so for example *p.inputs* stands for the set of inputs of a process *p.* Types of inputs and outputs are defined as concepts in an ontology or as simple XSD data-types. We represent inputs and outputs as typed variables, so for example $p.inputs = \{(?v, T) \mid ?v \in Var, T \in Types\}$, where *Var* is a set of variable names (we assume each input and output name to be unique) and *Types* is a set of types. We represent preconditions and effects of processes as expressions in the form of conjunction of description logic atoms enriched with OWL datatypes. Specifically, we use a *DL-Lite* (Calvanese *et al*., 2005) subset of OWL since the basic reasoning tasks are computationally tractable. *DL-Lite* can also be safely combined with an action planning formalism since it is closed with respect to instance level update operation (De Giacomo *et al*., 2006) which is needed when DL instances are used to represent the state of a planner.

3 Process mediation problem

When requesters and providers use fixed, incompatible communication protocols interoperability can be achieved by applying a *mediation process* which resolves all incompatibilities, generates appropriate mappings between different processes and translates messages exchanged during runtime. 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.[1] We use *R* to stand for the requester's process model and *P* for the provider's process model. Also, we assume that both process models share the same domain ontology and target the same problem. We want to avoid the situation when, for example, the provider is a book selling service and the requester needs a library service.

In order to introduce the problem of process mediation, we use the notion of the provider's *ability to satisfy* the requester. We write *satisfied*(R, P) if the provider's process model P is able to *satisfy* the requester's process model R. Intuitively, *satisfied*(R, P) holds if for each requester's service call the provider can receive that request and return the required outputs and effects. Also we require that both the requester and provider execute their process models according to their execution semantics (Ankolekar *et al*., 2002) and that both initiated processes reach a proper final state. We give a precise definition of the *satisfied* predicate in Section 5.1.1. Clearly, if *satisfied*(R, P) is true there is no need for process mediation between R and P. But since we assumed that R and P are incompatible, there exist mismatches between R and P that do not allow *satisfied*(R, P) to be true. We distinguish *static* and *runtime* mismatches. Static mismatches (we write mis_{static}) are those mismatches which can be identified without the need to execute the processes models. Processes re-orderings or variable types incompatibilities are typical examples of static mismatches. On the contrary, runtime mismatches (we write $mis_{runtime}$) can be identified only during the runtime execution because they depend on actual values returned by service calls. Examples include preconditions evaluation failure, conditionals evaluations, *etc*. Given the existence of mismatches between R and P the problem of process mediation can be represented as a tuple $(R, P, mis_{static}, mis_{runtime})$. The mediation process can be seen as a process model M that translates the identified mismatches and realises all the mappings between P and R. Assuming that M reconciles all static mismatches, the following conditions must hold:

- *satisfied*(R, M), or in other words $(R, M, \emptyset, mis_{runtime})$
- *satisfied*(M, P), or in other words $(M, P, \emptyset, mis_{runtime})$.

In this paper, we focus on the following major problems:

- Identification of static mismatches mis_{static} between R and P and synthesis of mediator's process model M to reconcile R and P. We consider the offline generation of M in the complete visibility scenario and also the runtime generation of M in the asymmetric scenario.
- Development of the runtime mediation mechanisms and infrastructure to deal with runtime mismatches $mis_{runtime}$. On the other hand, we do not address the data mediation problem. We assume that data mediators can be accessed as external web services.

From the procedural point of view, process mediation consist of the following steps:

Step 1 *Communication with the requester* – The requester initiates communication with the mediation process with an initial request corresponding to its own process model. Later, the requester continues executing its process model by sending other request messages to the mediator.

Step 2 *Communication with the provider* – The mediator identifies an appropriate process or processes of the provider's process model that can satisfy (match) the requester's service call.

Step 3 *Analysis of mismatches* – The mediator identifies possible mismatches, such as data incompatibilities or service re-orderings, and computes the *reconciliation plans* that serve to resolve the identified mismatches.

Step 4 *Discovery of external services, such as data mediators* – If some mismatches cannot be resolved, possibly new external services (*e.g.*, a currency conversion service) for bridging identified mismatches might need to be discovered.

Step 5 *Execution of the reconciliation plan* – The mediator executes the computed reconciliation plan, which might include:

- calling the provider's processes
- invoking external services for providing missing pieces of information or performing data translations
- performing internal mediation actions, such as performing data conversions
- returning a response message to the requester.

3.1 Incompatibilities

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

1 *Data level mismatches*:

- *Syntactic/Lexical mismatches* – data are represented as different lexical elements (numbers, dates format, local specifics, naming conflicts, *etc*.)
- *Structural data mismatches* – data are represented in different data structures (arrays, records, sets, *etc*.)
- *Ontology mismatches* – the same information is represented as different concepts:
  a in the same ontology (subclass, superclass, siblings, no direct relationship)
  b or in different ontologies, *e.g.*, (customer vs. buyer)

2 *Service level mismatches*:

- a requester's service call is realised by several providers' services or a sequence of requester's calls is realised by one provider's call
- requester's request can be realised in different ways which may or may not be equivalent (*e.g.*, different services can be used to satisfy requester's requirements)
- reuse of information: information provided by the requester is used in a different step in the provider's process model
- missing information: some information required by the provider is not provided by the requester
- redundant information: information provided by one party is not needed by the other one

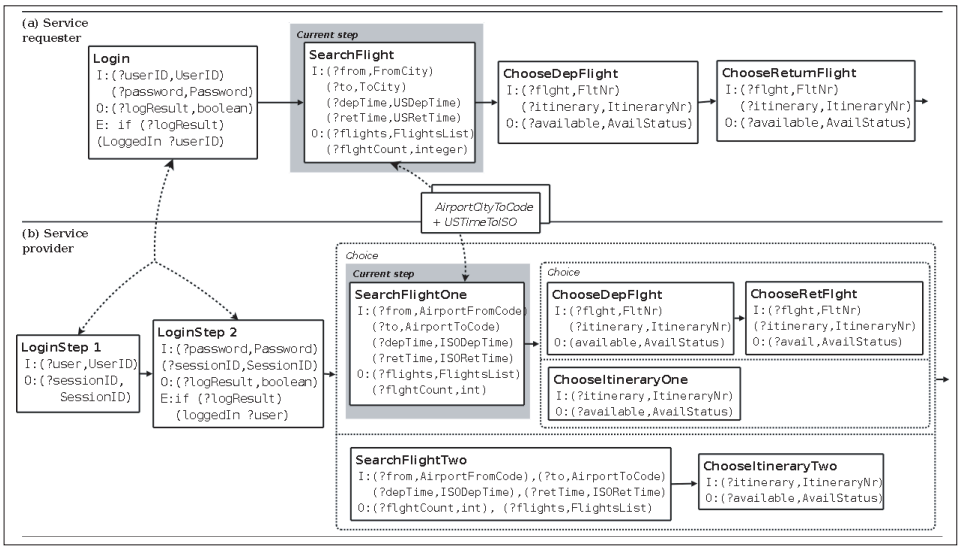
3 *Protocol/Structural level mismatches* – control flow in the requester's process model can be realised in very different ways in the provider's model (*e.g.*, sequence can be realised as an unordered list of steps, service calls might be realised in different ordering, *etc.*).

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. In our work we assume, that we work with web services which already have the grounding provided.

3.2 Example problem

Figure 1 presents an example of the mediation problem between a hypothetical requester and provider from the flights booking domain. Specifically, Figure 1(a) depicts a fragment of the process model of the requester while the provider's process model fragment depicted in Figure 1(b) represents a more elaborate scenario that allows the requester to book either the whole itinerary or to pick the departure and return flights separately.

Figure 1 Process models of the requester and provider and possible mappings between them



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.

Figure 2 Fragment of the flights domain ontology with only concepts and ISA-relations displayed

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

The example demonstrates several types of mismatches and a possible mapping between both process models represented by dashed arrows between parts (a) and (b) of Figure 1. Sometimes, the mapping can be achieved without the use of any help of external services, such as in the case of requester's *Login* atomic process. In this case, the service level mismatch is resolved by mapping the *Login* request into two provider's atomic process, *LoginStep1* and *LoginStep2*, which are able to produce the required outputs and effects. 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 the requester's *SearchFlight* atomic process and the provider's *SearchFlightOne* process. In this particular case, a combination of external services *AirportCityToCode* (for translating instances of *City* to instances of *AirportCode*) and *USTimeToISO* (for translating between US and ISO time formats) can be used to bridge the gap as shown in Figure 1. Although the example is rather simple it illustrates many problems such as parameters incompatibilities, process models branching (choices), or multiple equivalent or compatible parameters (*?from*, *?to*, *etc.*) which all makes finding the correct mappings a hard problem.

3.3 Mappings

We represent the process model *M* of mediator as a set of mappings between process models *R* and *P*. Each mapping is basically a sequence of *reconciliation plans* which realise translations of one requester's atomic process to the provider's model (see Section 5.1.1 for exact definitions). Reconciliation plans are responsible for the following structural transformations: *step re-ordering*, *adding a step* (*e.g.*, executing external service), *suppressing a step* (*e.g.*, ignoring results of some process), *condensing several steps* of requester into one provider's, and *decomposing one requester's step* into several provider's steps.

Internally, we represent reconciliation plans as sequences of commands that can be executed by the mediator execution infrastructure during runtime to realise the actual translations. Specifically, a reconciliation plan consists of commands for executing provider's atomic processes (*execute-provider*), communication with requester (*receive-from-requester*, *send-to-requester*), execution of external services (*execute-external*) and internal translations (*e.g.*, *mediator-explicit-downcast*).

Figure 3 shows part of a mapping consisting of two reconciliation plans generated by algorithms presented in following sections for the requester and the provider in Figure 1. Requester's requests (*receive-from-requester*) show names of input parameters

(*i.e.*, variables received from requester), while for provider calls, external services calls and internal translations also output variables and input bindings are shown. Names of services are underlined. The shown reconciliation plans exactly correspond to dashed arrows in Figure 1. Notice that reconciliation plans do not explicitly represent described transformations such as step re-ordering, step repressing, *etc*. Rather these transformations are represented implicitly, by, *e.g.*, invoking particular service calls with particular variable bindings. For example, in Figure 3 the reconciliation Plan 1 for the *Login* requester's call represents a requester's step decomposition by mapping the *Login* to provider's *LoginStep1* and *Login-Step2*.

Figure 3 Example mapping (reconciliation plans) for possible requester's execution

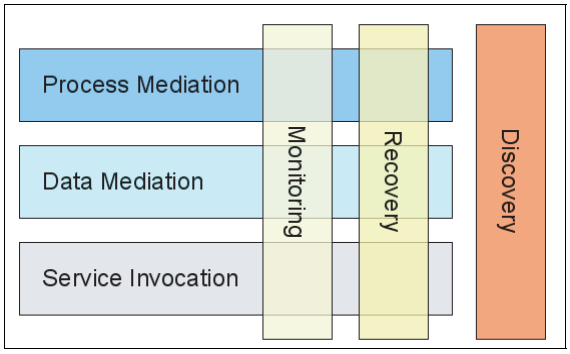
The requester's execution sequence: Login, SearchFlight, ChooseDepFlight, ChooseReturnFlight, ...	
Reconciliation plans computed for first two steps of the requester's execution:	
I. Reconciliation plan for *Login*	*1. receive-from-requester* Login i1-userID i1-password *2. execute-provider* LoginStep1 user=i1-userID Outputs: o2-sessionID *3. execute-provider* LoginStep2 password=i1-password sessionID=o2-sessionID Outputs: o2-logResult *4. send-to-requester* logResult=o2-logResult
II. Reconciliation plan for *SearchFlight*	*1. receive-from-requester* SearchFlight i1-from i1-to i1-depTime i1-retTime *2. execute-external* AirportCityToCode from=i1-from Outputs: o2-apt-code *3. mediator-explicit-down-cast* o2-apt-code AirportToCode *4. execute-external* AirportCityToCode to=i1-to Outputs: o4-apt-code *5. mediator-explicit-down-cast* o4-apt-code AirportToCode *6. execute-external* USTimeToISO time=i1-depTime Outputs: o6-iso-time *7. mediator-explicit-down-cast* o6-iso-time ISODepTime *8. execute-external* USTimeToISO time=i1-retTime Outputs: o8-iso-time *9. mediator-explicit-down-cast* o8-iso-time ISODepTime *10. execute-provider* SearchFlightOne from=o1-apt-code to=o4-apt-code deptTime=o6-iso-time retTime=o8-iso-time Outputs: o10-flights o10-flghtCount *11. send-to-requester* flights=o10-flights flght-count=o10-flghtCount

The 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 Spencer and Liu (2004) for details on analysing casting operations for ontology classes.

4 Abstract process mediation framework

We have defined an *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 4 shows the abstract process mediation framework. The three key functionalities, namely *process mediation*, *data mediation* and *service invocation*, are displayed as horizontal layers.

Figure 4 Abstract process mediation framework (see online version for colours)



The *process mediation layer*, realised by *process mediators*, is responsible for resolving service level and protocol level mismatches. The process mediation layer has to address three problems:

1 finding mappings between processes

2 identifying possible information gaps that impede the mediation

3 providing suitable mechanisms for any needed runtime mediation.

We focus on these problems in this paper.

The *data mediation layer*, realised by *data mediators*, is responsible for resolving data level mismatches. Typically, when trying to achieve interoperability, process mediators and data mediators are closely related. We use data mediators within the process mediation component to resolve 'lower' level mismatches that were identified during the process mediation. 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 the form of converters that 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. Other than that, we do not address the problem of data mediation. For details related to data mediation see for example (Spencer and Liu, 2004; Burstein *et al*., 2003; Vaculín *et al*., 2008a).

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 case of OWL-S web services, the service invocation layer must be able to interpret the Process Model of the service according to its semantics (Ankolekar *et al*., 2002) and provide a generic mechanism for invocation of web services represented as atomic processes in the Process Model. For invocation we use a component developed by Paolucci *et al*. (2003).

Since we assume that environments are dynamic and might be open, the mediation processes make use *of monitoring* and *recovery*. We display these functionalities as vertical layers in Figure 4 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 (for details on monitoring and recovery in the OWL-S

context see Vaculín and Sycara (2007a; Vaculín *et al*., 2008c)). Finally, we also consider the *discovery of external services* functionality as closely related to the process mediation problem in dynamic environments.

4.1 Visibility and means of process mediation

As we mentioned in the introduction, we distinguish two possible degrees of visibility for a given process model. The *white box visibility* means that the mediation process can see the complete process model, including all relevant operations definitions (atomic processes) and the complete interaction protocol (control and data flows). In case of *black box visibility*, the mediation process can see only definitions of available operations, while the process workflow and the interaction protocol is not disclosed. Such a situation is motivated either by privacy concerns or by the fact that no formal specification of the interaction protocol exists. Four scenarios are possible depending on visibility of each participating process model. Table 1 summarises these scenarios in the form of a visibility matrix. The visibility matrix identifies boundary cases and means of mediations in these situations.

Table 1 The visibility matrix

Requester/ Provider	***Black Box Provider*** • Only operations visible • *Environments*: all	***White Box Provider*** • Complete process model visible • *Environments*: all
Black Box Requester • Only operations visible • *Environments*: open	*I. Black Req. - Black Prov.* • runtime: reactive mediation based on matching operations	*II. Black Req. - White Prov.* • runtime: reactive mediation (Sec. 5.4)
White Box Requester • Complete process model visible • *Environments*: semi-open, closed	*III. White Req. - Black Prov.* • offline: requester's execution paths exploration + matching of operations • runtime: employ pre-computed mappings	*IV. White Req. - White Prov.* • offline: requester's execution paths exploration + provider's process simulation (Sec. 5.1) • runtime: employ pre-computed mappings

We distinguish two basic modes of mediation: an *offline analysis,* in which some computations can be performed before the runtime mediation starts, and a *reactive mediation*, in which all computations need to be performed strictly during runtime. The offline analysis can be employed only when the process models are known in advance. This includes for example enterprise and B2B scenarios (such as Case IV in the matrix). During offline analysis, possible mismatches between the process models can be identified and corresponding mappings for bridging these mismatches can be computed either fully automatically or with a human assistance.

Reactive mediation addresses situations when offline analysis is not possible. Reactive mediation is suitable especially in open environments, where the cooperating partners (requester and provider) are discovered at runtime (*e.g.*, Cases I and II). Since all calculations need to be performed within restricted amount of time, often with incomplete information about participating process models, incorrect decisions might be made. Therefore advanced recovery mechanisms need to be incorporated.

In Cases I and III, when the provider's interaction protocol is not known, it is important to realise that for the process mediation component it means that *any* operation of the provider can be possibly called 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 possibly identify necessary translations for this operation.

5 Process mediation algorithms

In this section we describe mediation techniques and an execution architecture for a *complete visibility scenario* (Case IV in the visibility matrix), and for an *asymmetric scenario* (Case II in the visibility matrix). By addressing these two scenarios we devise mediation mechanisms covering two most challenging mediation classes (*offline analysis* and *reactive mediation*) which can be later combined, *e.g.*, with matchmaking to deal with the remaining visibility settings. We start with the complete visibility scenario and describe algorithms for computing mappings between process models of the requester and the provider in Section 5.1. We continue with the asymmetric scenario in Section 5.2, which is harder because mappings cannot be precomputed in advance. We analyse properties of developed algorithms in Section 5.3. Finally, we introduce a *Process Mediation Agent (PMA)* in Section 5.4 which presents an execution infrastructure for mediation during runtime.

5.1 Computing mappings in the complete visibility scenario

The problem of process mediation can be seen as finding appropriate mappings between requester's and provider's process models. We need to decide if and how *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[3] of the requester the provider with some possible help of intermediate translations represented by *internal mediation actions* (such as *built-in conversions)* and *external service calls* (such as *data mediators)* can satisfy the requester's requirements (*i.e.*, providing required outputs and effects) while respecting its own process model, control constructs and IOPEs. This can be achieved by exploring possible sequences of steps (execution path) that the requester can execute.

Definition 1

The *requester's execution path* of process model R is any sequence $pathR = (r_1, \ldots, r_n)$ of atomic processes $r_1, \ldots, r_n$ which can be called by the requester in accordance with its process model, starting from the process model first atomic process and ending in 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. 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).

Finding possible mappings means exploring the search space generated by combining allowed execution paths in the provider's process model with available translations. 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, internal mediation actions and external services 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:

- *Generate the minimal set of requester's paths* – based on the process model of the requester and the executional semantics of OWL-S (Ankolekar *et al*., 2002), possible requester's paths are generated (see Section 5.1.2)
- *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* plans library – if no mapping is found for some particular path, the part of the path for which the mapping was not found is conveyed to the user (see Section 5.1.3).

5.1.1 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 2

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 offline 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 1.

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 3

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 $\mathscr{PA}$, 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 6).

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 4

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:

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

Conditions 1–4 of Definition 4 are usual compatibility requirements (see, *e.g.*, Paolucci *et al*., 2002). 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 5

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 anew 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 6

We say that the requester's execution path $pathR = (r_1, \ldots, r_n)$ *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 7

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 8

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

For example, in the running example in Figure 1, if the last executed provider's call was *SearchFlightOne*, the *Next* function would return a set of two atomic processes $\{ChooseDepFlght, ChooseItineraryOne\}$.

5.1.2 Generating the minimal set of requester's execution paths

Possible requester's paths are generated based on the process model of the requester and the executional semantics of OWL-S (Ankolekar *et al*., 2002). The execution paths are generated so that each reachable requester's service call (atomic process) is included in at

least one explored path. This guarantees, that all possible requester's calls will be considered. 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 a given state of execution which depend 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 Vaculín and Sycara (2007b) 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.

5.1.3 Computing 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 process model (atomic process or several atomic processes) with help of *internal mediator actions* and *external services* (*e.g.*, data mediators). The mappings are constructed during the simulation and are represented as sequences of reconciliation plans that can be used during runtime mediation (see Figure 3 for an example of a mapping).

Algorithm 1 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 (Step 3) by calling the *reconcileRequestCall* procedure (defined in Algorithm 2). After that, each found reconciliation plan is simulated in the current simulated execution state by calling the *simulatePlan* procedure (Step 5.1). The *simulatePlan* method returns a new execution state which is a result of simulating a given reconciliation plan in a given state. Next, for the new state the *reconcileSequence* is called recursively in Step 5.2 of Algorithm 1 to calculate the mappings for the rest of the requester's execution path. If also the rest of the requester's execution path can be reconciled, the found reconciliation plan with the corresponding execution state is stored in the *plansLibrary* (Step 5.2.1). As a result, after executing the *reconcileSequence* method only all those reconciliation plans leading to successful mediation will be stored in the *plansLibrary.*

The purpose of the *reconcileRequestCall* 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 execution state of the provider's process model. The reconciliation procedure uses a planning algorithm that tries to combine internal mediation actions and known external services (*e.g.*, data mediators) to find necessary transformations. If no combination of internal mediation actions and known external services can be found, the reconciliation procedure produces an unbound plan associated with query templates which can be used later to discover new external services and to bind the plan, or to inform the user about identified mismatches.

Algorithm 1 *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. **Initialisation:**
   - *request* ← remove first request from *stepsSequence*
   - *returnStatus* ← *false*
3. *newPlans* ← *reconcileRequestCall*(*request*, *S*)
4. **if** *newPlans* = ∅ **then return** *false*
5. **foreach** *plan* ∈ *newPlans*
   - 5.1 *newState* ← *simulatePlan*(*plan*, *S*)
   - 5.2 **if** *reconcileSequence*(*stepsSequence*, *newState*, *plansLibrary*) **then**
     - 5.2.1 store (*request*, *S*, *plan*) in *plansLibrary*
     - 5.2.2 *returnStatus* ← *true*
6. **return** *returnStatus*

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

1. **Initialise:**
   - **foreach** *i* ∈ *r.inputs* **do** add (*Available i*) to *S.V*
   - **foreach** *o* ∈ *r.outputs* **do** add (*RequesterGoal* (*Available o*)) to *S*
   - **foreach** *effect* ∈ *r.effects* **do** add (*RequesterGoal effect*) to *S*
   - *plans* = ∅
2. **foreach** *p* ∈ *Next*(*S*) **do** *reconcileAtomicProcess*(*r*, *p*, *S*, *plans*, ∅)
3. **return** *plans*

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 external services (such as data mediators) and internal mediation actions used as planning operators (see, *e.g.*, Sirin and Parsia (2004) for details about transformation of OWL-S processes into planning operators). During the planning, the operations are only simulated (services are not executed) with respect to the initial execution state *S*.

Algorithm 2 takes care of the planner initialisation (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 a request *r* and an atomic process *p*. First, it guarantees that all inputs and preconditions of *p* are available (Step 1), and afterwards that all outputs and effects required by *r* are produced (Step 2). In both cases the *solveGoals* procedure implementing a backward chaining algorithm is tried first to achieve missing goals by means of known external services (such as data mediators) and internal mediation actions. 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.

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

1 **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*

2 **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., (*RequesterGoalgoal*) => (*Goalgoal*)
  - **if** *solveGoals*(*Goals*, S, *plan*, *maxPlanLength*) **then** add *plan* to *plans*

    else
    * *duplicate the plan and continue with another process in the provider's model*
      · *newPlan = plan*
      · **foreach** $a \in Next(S)$ **do**
      reconcileAtomicProcess($r, a, S, plans, newPlan$)
    * *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*

An important remark is related to plans ranking. Generated reconciliation plans are ranked depending on their quality and length. The quality of a plan is derived from the degree of match between individual plan steps and required information that the step is supposed to provide in a given execution state. Specifically, the matching combines three factors:

1 A classical semantic matching based on parameter types known from WS matchmaking literature (Paolucci *et al.*, 2002) is used as a primary criteria. Matching type of every parameter can be either *exact*, *plugin*, *subsume*, or *fail* with a decreasing quality.

2 Next, the matchmaking function disadvantages parameter substitutions that assign same values for different parameters. The motivation stems from an insight that when the WS designer intentionally introduced several parameters to a service call, each of these parameters has very likely a specific purpose and therefore a substitutions using different values for different parameters is more likely a better match than the one using same value for different parameters. Consider for example a *SearchFlightOne* provider's atomic process in Figure 1 which has among other parameters the *(?from, FromCity)* and *(?to, ToCity)* inputs. In the planning process a substitution which will assign different locations to *?form* and *?to* will be preferred.

3 Finally, the matching functions prefers substitutions that are using recently provided values – *i.e.*, data provided by the current call are preferred. An insight motivating this heuristics is that when a provider provides some data to the requester in a current call it wants the requester to use these data rather that using data from previous calls. Given the matching degree of a step, we assign *a price* to it: the better the degree of match of the step, the lower is its price. The price of the overall plan is gained as a sum over all its steps. During the planning the best-first search strategy is used and when it comes to the execution, the plans are considered in the descending order according to their price. This means that short plans with good quality are preferred. We discuss this issue in the evaluation in Section 7.

5.2 Computing mappings in the asymmetric scenario (reactive mediation)

Similarly to computing the reconciliation plans during the offline analysis, the purpose of the reactive 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. However, compared to mediation in the complete visibility scenario, means of analysis are restricted by the black box visibility of the requester's process model and by the fact that all analysis must be performed during runtime only.

Consider the situation of the requester from Figure 1(a) in which it executes the *Search-Flight* atomic process (emphasised by the grey background in the figure). This step can be mapped either to the *SearchFlightOne* process (as illustrated by dashed arrows in Figure 1) or to the *SearchFlightTwo* process of the provider. In the complete visibility case the mediation algorithm was able to select the right reconciliation plan (*i.e.*, the one for *SearchFlightOne*) for which the mapping exists also for subsequent requester's steps (*Choose-DepFlght*, *ChooseRetFlght*) since possible executions of the requester and provider could have been explored and possibly wrong choices could have been avoided before the actual execution. However, such a deep analysis is not possible in the runtime reactive mediation, because for a given execution state the possible subsequent requester's step are not known. Thus, for the reactive mediation the *SearchFlightOne* and *SearchFlightTwo* processes are indistinguishable since there is no difference in their IOPEs, and the subsequent requester's step (*ChooseDepFlight* and *ChooseReturnFlight*) are not known. Therefore the mediator could choose the *SearchOneWayFlight* which would be wrong since no mapping exists for following two steps (*ChooseDepFlight* and *ChooseRetFlight*) in this context, while in case of selecting the *SearchTwoWayFlight* the mapping exists.

Since the requester's process model and thus also the possible requester's execution sequences are not known to the mediation process in the asymmetric setting, the reactive reconciliation procedure is reduced only to mediation between the current request r and the provider's execution state S. We developed the *reconcileRequestRuntime* procedure for computing the reconciliation plans which uses basically the same logic as the *reconcileRequestCall* procedure for offline analysis introduced in Algorithm 2. In order to guarantee timely termination of the planning algorithm, the maximal length of the reconciliation plan is constrained externally. Our rationale behind constraining the maximal plan length is motivated by an insight, that reasonably working reconciliation plans are very likely to be rather short (for example simple data translations), while long reconciliation plans will tend to be unreliable and quite unrealistic.

To compensate for local decision making which might lead to selection and execution of a wrong reconciliation plan, the recovery mechanisms in the form of undoing and compensation are employed (see Section 4). Thus, if the wrong choice is made, the undo mechanisms are tried to recover and to possibly allow other choices to be explored.

5.3 Algorithms properties

The correctness and the completeness of introduced algorithms can be defined in terms of the *satisfied* predicates defined in Section 5.1.1. We discuss these properties with respect to the single request r reconciliation (*i.e.*, $satisfied(r, M, S)$ property in Definition 5), with respect to the requester's path $path_R$ reconciliation (*i.e.*, $satisfied(path_R, P)$ property in Definition 6), and with respect to the whole process models reconciliation (*i.e.*, $satisfied(R, P)$ property in Definition 7).

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).[4] 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 5.1 is sound and complete with respect to the $satisfied(R, P)$ predicate, since the set of requester's paths generated as described in Section 5.1.2 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.

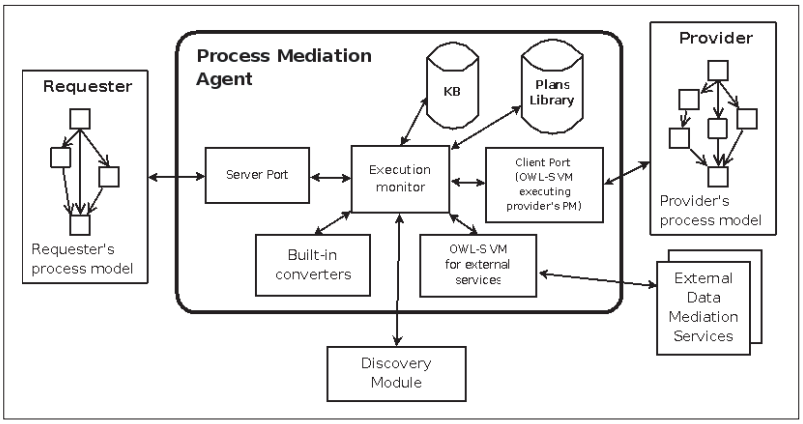
Another important property of generating the process mediator protocol is whether it guarantees a *deadlockfree* communication of the requester and the provider (assuming that both process models are deadlock free). According to Brand and Zafiropulo (1983) a communication is deadlock free if it ends 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.

5.4 Execution infrastructure: the process mediation agent

In this section we introduce the PMA for runtime mediation. The architecture of the PMA is designed to support either the mediation in the complete visibility mode when it uses mappings precomputed mappings, or in the strictly reactive mode in which all mappings are computed only during runtime. Also the PMA addresses runtime mismatches ($mis_{runtime}$).

Figure 5 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) (Paolucci *et al.*, 2003) 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 (Vaculín and Sycara, 2008) and recovery (Vaculín *et al.*, 2008c). Another instance of the OVM is used to execute external data mediation services if necessary.

Figure 5 Mediation of process models by the Process Mediation Agent (PMA): problem setting and the PMA architecture



The *Execution Monitor* is the central part of the PMA. It executes the runtime mediation procedure described in the next subsection 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 Module* when new data mediators need to be found during runtime. The *Plans Library* is used to store reconciliation plans that were either provided as an output by the offline analysis, or that were used successfully in previous mediation sessions. The primary purpose of the plans library is to improve efficiency by avoiding redundant planning in future mediation sessions. Also information about discovered data mediators is cached in the *Plans Library*. When no historical or precomputed information is available, the PMA explores the search space to find an appropriate reconciliation plan.

5.4.1 Runtime mediation procedure

The logic of the runtime 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 4 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.

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). As the next step (Step 3.3), the planning algorithm is used (*reconcileRequestRuntime* procedure, for details see Section 5.2) that tries to find new reconciliation plans. 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 6). 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 (as demonstrated in Section 5.2). The *globalRecover* procedure in Step 3.4 tries to deal with such a situation by backtracking to the nearest such a branching point in the provider's process model while compensating the executed actions in the execution history. If the backtracking with compensation succeeds, an alternative branch and an alternative reconciliation plan is chosen if it exists. Otherwise the backtracking continues to the next branching point.

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.

Since interactions with external web services can fail, and also some choices made by the PMA can lead to failure (mostly because the reconciliation plans focus on static mismatches mis_{static}, while the runtime mismatches $mis_{runtime}$ such as preconditions failures need to be handled during the runtime), the PMA tries to recover from failures. We already mentioned the *globalRecover* procedure which serves as a last resort if no other solution works. In addition to that the runtime mediation procedure incorporates the *reconciliation plans level recovery* realised by the *localRecover*(P, S). When some part of the reconciliation plan fails the plan as a whole is recovered by *compensating* the effects

of the plan[5] and possibly an alternative reconciliation plan is tried. Finally, at the lowest level, failures of individual service calls are recovered by attaching fault handlers to them that either retry the service or possibly try an alternative service. Generally, the recovery mechanisms are based on applying fault handler templates employing recovery actions such as *replace*, *retry*, *replaceByEquivalent*, and *compensate* which were introduced as generic recovery mechanisms for OWL-S services in Vaculín *et al*. (2008c), and on employing mediation specific recovery strategies such as global and local recovery.

Algorithm 4 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** $reactiveMode \wedge reconciliationPlans = reconcileRequestRuntime(requesterCall, S) \neq \emptyset$ **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**

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

- **return failed**

6 Data mediators discovery

The reconciliation procedure defined in Algorithms 2 and 3 is able to identify incompatibilities that cannot be resolved by using any of known mediation methods. We use *query templates* defined in Section 5.1.1 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 is 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 (Paolucci *et al*., 2002). In case of process mediation 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. This is demonstrated in the running example in Figure 1 in which the reconciliation plan between the *SearchFlight* request and the *SearchFlightOne* process can be realised only by combining two services, *AirportCityToCode* and *USTimeToISO*.

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*. (2003) 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* (see Vaculín *et al*., 2008b for details). 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.

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.

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** $M' = \langle P, \emptyset \rangle$

7 Experimental evaluation

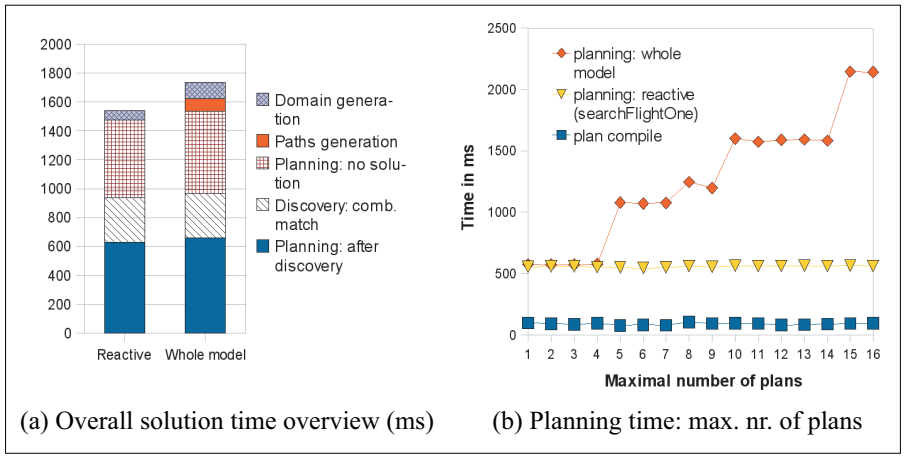
In this section we provide an experimental evaluation of presented process mediation algorithms. Primarily we focus on the algorithms for finding mappings between process requester's and provider's models (*i.e.*, algorithms described in Sections 5.1 and 5.2) since these parts are most critical and possibly most time consuming. The evaluation is based on implementation of described algorithms which is partly done as a standalone component and partly as part of the SHOP[6] planner that we use for the planning part of process mediation. The evaluation is based on an example scenario presented in Section 3.2. Although the scenario seems to be simple in terms of presented incompatibilities, it is complex enough for basic evaluation and for identifying problems that mediation algorithms have to deal with.

In our experiments, we assume that the mediation component does not have any prior knowledge about external services and that no existing plans are available in the plans library by which we enforce interactions with matchmaker and using the planning algorithm. We evaluate both the reactive mediation and the offline analysis. The analysis is performed for the whole process models as displayed in Figure 1. In case of the reactive mediation, the reconciliation plans are always computed for one particular client's request at time (the current one). For evaluation purposes we are using the *SearchFlight* client's atomic process which enforces discovery, usage of external services and types conversions.

In the first scenario, we evaluated the overall duration of a mediation process. Figure 6(a) shows overall durations in milliseconds for both the reactive mediation and the offline mediation with durations of mediation phases showed with different filling patterns. From proportions of each column in the graph it can be seen that the most of the time is needed for the reconciliation planning itself (notice that planning was used first before discovering external services and later continued with new external services added as new planning operators) which is followed by interactions with the matchmaker. In this particular scenario we simply used the first plan returned. However, as illustrated in Figure 6(b), even for our simple scenario, there exist altogether 16 plans. This surprisingly high number of plans is caused by the fact that several parameter substitutions exist due to parameter similarities (such as *?from*, *?to*, or *?depTime*, *?retTime*) and available ontology axioms and reasoning.

Figure 6(b) shows planning and domain generation times as a function of maximal number of plans retrieved (out of total 16 plans). In case of reactive mediation the time is almost constant which might be attributed to some internal planner (SHOP) optimisation. In case of the offline analysis the time for planning phase increases quite rapidly with the increasing number of retrieved plans. This can be explained by presence of two branching points (choices) in the provider's process model which make finding all the plans harder due to backtracking. As this scenario illustrates, one of issues we have to deal with is an explosion of possible plans count. Although many of them are correct plans with respect to definitions in Section 5.1.1, only a small fraction will present a working solution with reasonable parameter bindings. Therefore we introduced plans ranking (Section 5.1.3) which allows us to sort plans according to their quality. One strategy how to deal with a high number of solutions is to retrieve only best top-k plans instead of all plans which we demonstrate in the following experiment.

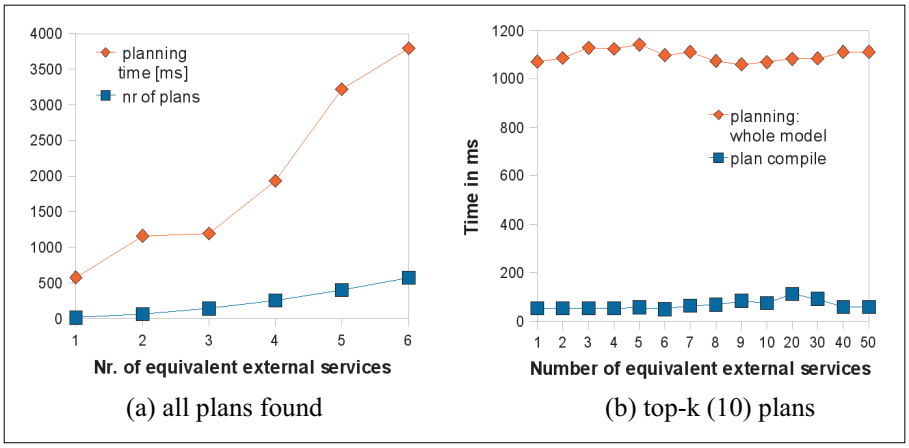
Figure 6 Mediation times for an example scenario in Figure 1 (see online version for colours)



(a) Overall solution time overview (ms) (b) Planning time: max. nr. of plans

Another source of problems is related to the number of possible branches or choices in the plan. We evaluate such a scenario by increasing the number of available external services which are the least constrained planning actions (can be used anywhere in process models of the requester and provider if they can contribute to the solution). To evaluate this effect we step-wisely added external services to the system which are equivalent with the *AirportCityToCode* and *USTimeToIso* services. By doing so, we allowed the planner to chose out of several candidate services and thus increased the possible number of plans. Such a scenario is very realistic since in presence of ontologies and flexible matching typically more candidate services might be available to bridge a particular gap. Figure 7(a) shows the total number of plans and the total planning time as a function of number of equivalent services (for the offline mediation scenario). Already with a very small number of services (6) there are around 500 plans available and with adding one another service our system ran out of memory. Figure 7(b) illustrates the same scenario in which only best ten plans are retrieved. As it can be seen, the planning time remains almost constant even when the number of equivalent services grows to tens of services.

Figure 7 Effect of an increasing number of equivalent services (see online version for colours)



(a) all plans found (b) top-k (10) plans

The experiments identify two important issues. First, we can observe a rather fast growth of time complexity with respect to the number of choices in the provider process model in the offline analysis scenario. It can be argued that the example problem is extreme in the sense that it allows all the choices to be considered (the signatures of processes in all branches are almost identical) and that we can afford a long time for an analysis during the offline. At the same time, however, we think that implementation of our algorithm can be improved. The second issue – a big number of potential reconciliation plans – is a more principled one. Although the problem can be partially addressed by using a sophisticated ranking of plans based on semantic matching, which is the solution we use and that works quite well for our scenario, there is no guarantee that in some other scenario it will not work that well. A traditional solution to this problem is to employ process dependent rules (such as in Yellin and Strom, 1997 or Brogi *et al*., 2004) which serve as a high level specification of desirable (correct) mediation process (in literature more often referred as an adapter) for a given pair of the requester and the provider.

8 Related work

Our work is related to several areas of research such as protocols and software adapters synthesis, component-based software engineering and recently also (semantic) web services, which all traditionally deal with interoperability issues.

In the area of protocol synthesis, Zafiropulo *et al*. (1980) introduced one of the first methods using set of production rules for the interactive synthesis of protocols which were modelled as finite state machines. The follow-up works on communication protocols explored many directions involving variations such as synthesising a missing entity that would enable communication of several specified protocols (Merlin and Bochmann, 1983), or including error recovery mechanisms when unreliable communication is assumed (Ramamoorthy *et al*., 1986). Typically, protocols are represented as variations of FSMs or Petri Nets. Probert and Saleh (1991) present a survey of synthesis methods of communication protocols together with a systematic assessment.

In the object oriented community the problem of interoperability of interfaces was addressed for example by Thatté (1994) where adapters are synthesised to achieve an exact fit between interfaces. However, in this work, protocols were not included. Yellin and Strom (1997) enriches application interfaces specification with sequencing constraints and formally defines interface compatibility based upon protocols. A method for checking protocols compatibility and a method for synthesising a software adapters for bridging the identified gaps are provided. Adapters are synthesised based on a high-level description, called an interface mapping, which is a main difference compared to our work.

In the context of software adapters, Wiederhold *et al*. (1997) provide a conceptual underpinning and a general framework for automatic mediation. Primarily, data mediation is addressed. More recent works addressing data mediation include (Spencer and Liu, 2004), where 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 Burstein *et al*. (2003) while Burstein and McDermott (2005) argue for published ontology mapping to facilitate automatic translations.

In the context of Component-Base Software Engineering a lot of effort was invested to formal specification of behavioural aspects of component based systems. For example, Plasil *et al*. (1998) introduce the SOFA architecture and use the behavioural specification in the context of automated updating and reconfiguration. In their later work (Plasil and Visnovsky, 2002) description of component behaviour, formal reasoning about the correctness of the specification refinement and also about the correctness of an implementation is developed. Bracciali *et al*. (2005) develop a formal methodology for adapting components with mismatching interaction behaviour based on high-level notation for expressing adapter specifications and an automated procedure to derive adapters from these given specifications. Later, in Brogi *et al*. (2004), the same method is applied to the formalisation of web service choreography proposal (WSCI) and it is showed how to check whether two or more web services are compatible to interoperate or not, and, if not, whether the specification of adapters can be automatically generated. In Brogi and Popescu (2006) an automated generation of web service adapters for BPEL process is proposed. Adapter is capable of solving behavioural mismatches among BPEL processes. Adapter is synthesised by generating so called service execution trees and their dual forms which are later transformed into YAWL. Our work is different in using flexible matching based on ontologies and in possibility of using external translation services.

A remarkable effort is being done in the context of the Semantic Web Services Challenge.[7] The goal of this initiative is an automation of web services mediation, choreography and discovery. To mention a few examples, Cimpian and Mocan (2005) solve 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. Brambilla *et al*. (2006) apply a model-driven approach based on WebML language. Mediator is designed in the high-level modelling language which supports semi-automatic elicitation of semantic descriptions in WSMO.

Another body of work was done in the agent community. Aberg *et al*. (2005) describe an agent called sButler for mediation between organisations' 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 (Paolucci *et al*., 2005) 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 Cabral *et al*. (2006) and Domingue *et al*. (2005) 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).

9 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 with behavioural specifications of protocols and automatic synthesis of the mediator process (compare, *e.g*., with Brogi and Popescu, 2006). Also, compared to other approaches (such as the one of Cimpian and Mocan, 2005) 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.

Acknowledgements

This research was partially supported by 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.

References

Aberg, C., Lambrix, P., Takkinen, J. and Shahmehri, N. (2005) 'sButler: a mediator between organizations' workflows and the semantic web', *The World Wide Web Conference Workshop on Web Service Semantics: Towards Dynamic Business Integration*, Chiba City, Japan.

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., *et al*. (2003) *Business Process Execution Language for Web Services*, Version 1.1.

Ankolekar, A., Huch, F. and Sycara, K.P. (2002) 'Concurrent semantics for the web services specification language DAML-S', in F. Arbab and C.L. Talcott (Eds.) *COORDINATION*, Vol. 2315 of *Lecture Notes in Computer Science*, Springer, pp.14–21.

Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F. and Stein, L.A. (2004) 'OWL web ontology language reference', 10 February, W3C Recommendation, http://www.w3.org/TR/owl-ref/.

Benatallah, B., Hacid, M-S., Rey, C. and Toumani, F. (2003) 'Request rewriting-based web service discovery', in D. Fensel, K.P. Sycara and J. Mylopoulos (Eds.) *International Semantic Web Conference*, Vol. 2870 of *Lecture Notes in Computer Science*, Springer, pp.242–257.

Bracciali, A., Brogi, A. and Canal, C. (2005) 'A formal approach to component adaptation', *The Journal of Systems and Software*, January, Vol. 74, No. 1, pp.45–54.

Brambilla, M., Celino, I., Ceri, S., Cerizza, D., Valle, E.D. and Facca, F.M. (2006) 'A software engineering approach to design and development of semantic web service applications', *International Semantic Web Conference*, Vol. 4273 of *Lecture Notes in Computer Science*, Springer, pp.172–186.

Brand, D. and Zafiropulo, P. (1983) 'On communicating finite-state machines', *Journal of the ACM*, April, Vol. 30, No. 2, pp.323–342.

Brogi, A., Canal, C., Pimentel, E. and Vallecillo, A. (2004) 'Formalizing web service choreographies', *Electr. Notes Theor. Comput. Sci.*, Vol. 105, pp.73–94.

Brogi, A. and Popescu, R. (2006) 'Automated generation of BPEL adapters', in A. Dan and W. Lamersdorf (Eds.) *ICSOC*, Vol. 4294 of *Lecture Notes in Computer Science*, Springer, pp.27–39.

Burstein, M., McDermott, D., Smith, D.R. and Westfold, S.J. (2003) 'Derivation of glue code for agent interoperation', *Autonomous Agents and Multi-Agent Systems*, May, Vol. 6, No. 3, pp.265–286.

Burstein, M.H. and McDermott, D.V. (2005) 'Ontology translation for interoperability among semantic web services', *The AIMagazine*, Vol. 26, No. 1, pp.71–82.

Cabral, L., Domingue, J., Galizia, S., Gugliotta, A., Tanasescu, V., Pedrinaci, C. and Norton, B. (2006) 'IRS-III: a broker for semantic web services based applications', in I.F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold and L. Aroyo (Eds.) *International Semantic Web Conference*, Vol. 4273 of *Lecture Notes in Computer Science*, Springer, pp.201–214.

Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M. and Rosati, R. (2005) 'DL-lite: tractable description logics for ontologies', in M.M. Veloso and S. Kambhampati (Eds.) *AAAI*, AAAI Press/The MIT Press, pp.602–607.

Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S. (2001) *Web Services Description Language*.

Cimpian, E. and Mocan, A. (2005) 'WSMX process mediation based on choreographies', *Business Process Management Workshops*, pp.130–143.

De Giacomo, G., Lenzerini, M., Poggi, A. and Rosati, R. (2006) 'On the update of description logic ontologies at the instance level', *AAAI*, AAAI Press.

Domingue, J., Galizia, S. and Cabral, L. (2005) 'Choreography in IRS-III – coping with heterogeneous interaction patterns in web services', in Y. Gil, E. Motta, V.R. Benjamins and M.A. Musen (Eds.) *International Semantic Web Conference*, Vol. 3729 of *Lecture Notes in Computer Science*, Springer, pp.171–185.

Farrell, J. and Lausen, H. (2007) 'Semantic annotations for WSDL and XML schema', http://www.w3.org/TR/sawsdl/.

Merlin, P. and Bochmann, G.V. (1983) 'On the construction of submodule specifications and communication protocols', *ACM Trans. Programming Languages and Systems*, January, Vol. 5, No. 1, pp.1–25.

Paolucci, M., Ankolekar, A., Srinivasan, N. and Sycara, K.P. (2003) 'The DAML-S virtual machine', *International Semantic Web Conference*, Springer, pp.290–305.

Paolucci, M., Kawamura, T., Payne, T.R. and Sycara, K.P. (2002) 'Semantic matching of web services capabilities', in I. Horrocks and J.A. Hendler (Eds.) Vol. 2342 of *Lecture Notes in Computer Science*, Springer, pp.333–347.

Paolucci, M., Soudry, J., Srinivasan, N. and Sycara, K. (2005) 'A broker for OWL-S web services', in L. Cavedon, Z. Maamar, D. Martin and B. Benatallah (Eds.) *Extending Web Services Technologies: The Use of Multi-Agent Approaches*, Kluwer.

Plasil, F., Balek, D. and Janecek, R. (1998) *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*, 4 March.

Plasil, F. and Visnovsky, S. (2002) 'Behavior protocols for software components', *IEEE Trans. Software Eng*, Vol. 28, No. 11, pp.1056–1076.

Probert, R.L. and Saleh, K. (1991) 'Synthesis of communication protocols: survey and assessment', *IEEE Trans. Computers*, Vol. 40, No. 4, pp.468–476.

Ramamoorthy, C.V., Yaw, Y., Aggarwal, R. and Song, J. (1986) 'Synthesis of two-party error-recoverable protocols', *SIGCOMM*, ACM, pp.227–235.

Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., *et al.* (2005) 'Web service modeling ontology', *Applied Ontology*, Vol. 1, No. 1, pp.77–106.

Sirin, E. and Parsia, B. (2004) 'Planning for semantic web services', *Semantic Web Services Workshop at 3rd International Semantic Web Conference (iswc2004)*, http://www.mindswap.org/papers/SWS-ISWC04.pdf.

Spencer, B. and Liu, S. (2004) 'Inferring data transformation rules to integrate semantic web services', *International Semantic Web Conference*, pp.456–470.

Sycara, K., Paolucci, M., Ankolekar, A. and Srinivasan, N. (2004) 'Automated discovery, interaction and composition of semantic web services', *Journal of Web Semantics*, Vol. 1, No. 1, pp.27–46.

Thatté, S. (1994) 'Automated synthesis of interface adapters for reusable classes', *POPL*, pp.174–187.

Vaculín, R. and Sycara, K. (2007a) 'Specifying and monitoring composite events for semantic web services', *The 5th IEEE European Conference on Web Services*, IEEE Computer Society, 26–28 November.

Vaculín, R. and Sycara, K. (2007b) 'Towards automatic mediation of OWL-S process models', *2007 IEEE International Conference on Web Services*, IEEE Computer Society, 9–13 July, pp.1032–1039.

Vaculín, R. and Sycara, K. (2008) 'Semantic web services monitoring: an OWL-S based approach', *41st Hawaii International Conference on System Sciences*, IEEE Computer Society Press, 7–10 January.

Vaculín, R., Chen, H., Neruda, R. and Sycara, K. (2008a) 'Modeling and discovery of data providing services', *2008 IEEE International Conference on Web Services*, pp.1032–1039, IEEE Computer Society, 23–26 September.

Vaculín, R., Neruda, R. and Sycara, K. (2008b) 'Service discovery in the composition and process mediation context in open environments', *The 6th IEEE European Conference on Web Services*, IEEE Computer Society, 12–14 November, under review.

Vaculín, R., Wiesner, K. and Sycara, K. (2008c) 'Exception handling and recovery of semantic web services', *Fourth International Conference on Networking and Services*, IEEE Computer Society Press.

Wiederhold, G. and Genesereth, M.R. (1997) 'The conceptual basis for mediation services', *IEEE Expert*, Vol. 12, No. 5, pp.38–37.

Yellin, D.M. and Strom, R.E. (1997) 'Protocol specifications and component adaptors', *ACM Transactions on Programming Languages and Systems*, March, Vol. 19, No. 2, pp.292–333.

Zafiropulo, P., West, C., Rudin, H., Cowan, D. and Brand, D. (1980) 'Towards analysing and synthesizing protocols', *IEEE Transactions on Communication*, Vol. 28, No. 4, pp.651–661.

Notes

1 The OWL Services Coalition, *Semantic Markup for Web Services (OWLS)*, http://www.daml.org/services/owl-s/1.1/.

2 SOAP, Simple Object Access Protocol (SOAP 1.1), http://www.w3.org/TR/SOAP.

3 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.

4 By using the *DL-Lite* (Calvanese *et al*., 2005) subset of OWL the planning combined with ontology axiomatisation with the planner state represented as a DL A-Box remains computationally tractable (De Giacomo *et al*., 2006).

5 The compensation of the whole plan might not be possible if it is not possible to undo the effects of each step in the plan. However, undo operations must be provided only for *world affecting* actions, *i.e.*, actions with some effects, while the *information gathering* actions do not need to be undone. Since only the minority of web service calls are world affecting, there is a big chance that the compensation of the whole reconciliation plan will succeed even when the undo operations are not known for every action in the plan.

6 http://www.cs.umd.edu/projects/shop/

7 http://sws-challenge.org/