# Process Mediation of OWL-S Web Services*

Katia Sycara and Roman Vaculín

The Robotics Institute, Carnegie Mellon University
{katia,rvaculin}@cs.cmu.edu

**Abstract.** The ability to deal with incompatibilities of service requesters and providers is a critical factor for achieving smooth interoperability in dynamic environments. Achieving interoperability of existing web services is a costly process including a lot of development and integration effort which is far from being automated. Semantic Web Services frameworks strive to facilitate flexible dynamic web services discovery, invocation and composition and to support automation of these processes. In this paper we focus on mediation and brokering mechanisms of OWL-S Web Services which we see as the main means to overcome various types of problems and incompatibilities. We describe the process mediation component and the hybrid broker component that present two different approaches to bridge incompatibilities between the requester's and provider's interaction protocols.

## 1 Introduction

The main goal of Web Services is to enable and facilitate smooth interoperation of diverse software components in dynamic environments. Due to the dynamic nature of the Internet and rapid, unpredictable changes of business needs, the ability to adapt to changing environments is becoming important. Existing service providers should be able to communicate with new clients that might use different data models and communication protocols. Furthermore, any changes in the provided services requires intensive modifications of existing interfaces and implementation to maintain interoperability. The possibility of achieving interoperability of existing components automatically without actually modifying their implementation is therefore desirable.

Process mediation services present a possible solution in situations where interoperability of components with fixed, incompatible communication protocols needs to be achieved. A process mediation component resolves all incompatibilities and generates appropriate mappings between different processes. Implementing the mediation component is complicated and costly, since it has to address many different types of incompatibilities. On the data level, components may be using different formats to encode elementary data or data can be represented in incompatible data structures. Furthermore, messages can be exchanged in different orderings, some pieces of information which are required by one process may be missing in the other one, or control flows can be encoded in very different ways.

* This research was supported in part by Darpa contract FA865006C7606 and in part by funding from France Telecom.

T.S. Dillon et al. (Eds.): Advances in Web Semantics I, LNCS 4891, pp. 324–345, 2008.

Current web services standards provide a good basis for achieving at least some level of mediation. WSDL [1] standard allows to declaratively describe operations, the format of messages, and the data structures that are used to communicate with a web service. BPEL4WS [2] adds the possibility to combine several web services within a formally defined process model, thereby allowing one 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 WSDL-S [3], OWL-S [4] and WSMO [5] strive to enrich syntactic specifications with rich semantic annotations to further facilitate flexible dynamic web services discovery and invocation. Tools for reasoning can be used for more sophisticated tasks such as, matchmaking and composition [6].

In this paper, we address the problem of automatic mediation of process models consisting of semantically annotated web services. Processes can act as service providers, service requesters or communicate in peer-to-peer fashion. We are focusing on the situation where the interoperability of two components, one acting as the requester and the other as the provider, needs to be achieved. We assume that both the requester and the provider behave according to specified process models that are fixed, incompatible and that are expressed explicitly. We use the OWL-S ontology for semantic annotations because it provides a good support for the description of individual services as well as explicit constructs with clear semantics for describing process models.

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

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

The above mentioned characteristics of semi-open environments make the mediation a much harder problem. It is simply impossible to assume that various requesters and providers will interoperate smoothly without any mediation or that a one purpose mediation component can be developed for each new service provider or requester. However,

since it is reasonable to assume that each component provides a description of its interaction protocols and since mechanisms of registering components into the system can be controlled, it is possible (1) to analyze in advance if interoperability of some components is possible and if it is (2) to use mechanisms that utilize results of the analysis step to perform an automatic mediation. We will describe mechanisms for an automatic mediation in semi-open environments in Section 4.

*Dynamic open environments* add more levels of complexity to the mediation problem. Since components can appear and disappear completely arbitrarily, it is necessary to incorporate appropriate discovery mechanisms. Also it is not possible to perform an analysis step in advance because requesters and providers are not known in advance. Therefore the automatic mediation process must rely only on *run-time* mediation. In Section 5 we describe a standalone Runtime Process Mediation Service that can be combined with suitable discovery mechanisms. We also show how the Runtime Process Mediation Service can be integrated into a Broker component that performs both, the discovery of suitable service providers and the runtime mediation.

The rest of the paper is structured as follows. In Section 2, we present an overview of OWL-S and componets for discovery and invocation of OWL-S web services. In Section 3, we will discuss categories of problems and mismatches that must be resolved during the mediation process, we specify assumptions of our approach and we also introduce an example that demonstrates several mismatches between the requester's and the provider's process models. In Section 4, we describe an algorithm for process mediation in the semi open world[1]. In Section 5, we discuss the process mediation in the open world. Namely, we first introduce a Runtime Mediation Service that can be used as a standalone component, and next we describe how this component can be embedded into the Broker that combines the discovery with the process mediation. In Section 6, we give an overview of the related work and we conclude in Section 7.

## 2 Overview of Relevant Concepts and Components of OWL-S

OWL-S [4] is a Semantic Web Services description language, expressed in OWL [7]. 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 ways of how clients can interact with the service by defining the requester-provider interaction protocol. The Grounding links the Process Model to the specific execution infrastructure (e.g., maps processes to WSDL [1] operations and allows for sending messages in SOAP [8]). Corresponding Profiles, Process Model and Groundings are connected together by an instance of the Service class that is supposed to represent the whole 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 are specified by means of their inputs, outputs, preconditions, and effects (IOPEs). Types of inputs and outputs are

[1] In the rest of the paper we will use words *environment* and *world* interchangeably.

usually defined as concepts in some ontology or as simple XSD data-types. Processes can be combined into composite processes by using the following control constructs: sequence, any-order, choice, if-then-else, split, split-join, repeat-until and repeat-while. Besides control-flow, the Process Model also specifies a data-flow between processes.

From the perspective of necessary tools, there are two main areas that need to be covered: *service search and discovery* and *service invocation*.

In order to make its capabilities known to service requesters, a service provider advertises its capabilities with infrastructure registries, or more precisely middle agents [9], that record which agents are present in the system. UDDI registries [10] are an example of a middle agent, with the limitation that it can make limited use of the information provided by the OWL-S Profile. The OWL-S/UDDI Matchmaker [11,12] is another example, which combines UDDI and OWL-S Service Profile descriptions. The OWL-S/UDDI matchmaker supports flexible semantic matching between advertisements and requests on the basis of ontologies available to the services and the matchmaking engine. After a requester has found the contact details of a provider through matchmaking, then the requester and the provider interact directly with one another. Since in an open environment they could have been developed by different developers, incompatibilities during interoperation may happen.

Brokers present another mechanism of discovery and synchronization [9,13]. Brokers have been widely used in many agents applications such as integration of heterogeneous information sources and Data Bases [14], e-commerce [15], pervasive computing [16] and more recently in coordinating between Web services in the IRS-II framework [17]. A brokering component used as a middle agent between a requester and a provider addresses several problems: a broker can perform discovery and selection of providers incorporating a decision procedure assessing compatibility issues, it can perform mediation and so to allow interactions of otherwise incompatible partners, it can be used to maintain anonymity of a provider and a requester by acting as a proxy and effectively hiding their identities, brokers can perform a range of coordination activities such as load balance between several provider's, etc. In this paper we focus specifically on the broker ability to combine discovery and process mediation functionalities.

A tool for execution of OWL-S web services must be able to interpret the Process Model of the service according to its semantics and provide a generic mechanism for invocation of web services represented as atomic processes in the Process Model. The OWL-S Virtual Machine (OVM) [18] is a generic OWL-S processor that allows Web services and clients to interact on the basis of the OWL-S description of the Web service and OWL ontologies. Specifically, the OWL-S Virtual Machine (OVM) executes the Process Model of a given service by going through the Process Model while respecting the OWL-S operational semantics [19] and invoking individual services represented by atomic processes. During the execution, the OVM processes inputs provided by the requester and outputs returned by the provider's services, realizes the control and data flow of the composite Process Model, and uses the Grounding to invoke WSDL based web services when needed. The OVM is a generic execution engine which can be used to develop applications that need to interact with OWL-S web services.

The architecture of the OVM and its relation with the rest of the Web service is described in Figure 1. On the left side the provider is displayed together with its OWL-S

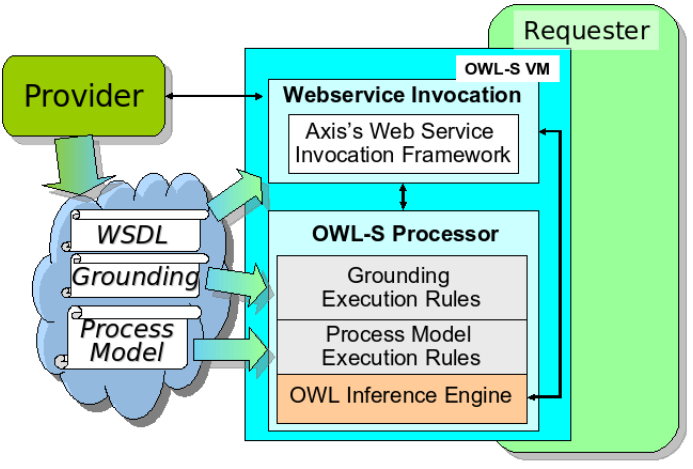


**Fig. 1.** The OWL-S Virtual Machine Architecture

Process Model, Grounding and WSDL description that together define how clients can interact with this service. The OVM is displayed in the center of the picture. It is logically divided in two modules: the first one is the OWL-S Processor which uses the OWL-S Inference Engine and a set of rules implementing the operational semantics of the OWL-S Process Model and Grounding to manage the interaction with the provider. The second component is the Web service Invocation module that is responsible for the information transfer with the provider. Finally, the OVM is shown as a part of the requester which can use it to interact with the provider.

## 3 Mediation Problem

In this section, we analyze the various types of incompatibilities that must be addressed during the mediation, describe how these mismatches can be generally handled in the context of OWL-S, introduce the problem setting and its assumptions and provide an example that demonstrates several mismatches.

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

1. *Data level mismatches*:
   (a) *Syntactic / lexical mismatches*: data are represented as different lexical elements (numbers, dates format, local specifics, naming conflicts, etc.).
   (b) *Ontology mismatches*: the same information is represented as different concepts
      i. in the same ontology (subclass, superclass, siblings, no direct relationship)
      ii. or in different ontologies, e.g., (Customer vs. Buyer)
2. *Service level mismatches*:
   (a) a requester's service call is realized by several providers' services or a sequence of requester's calls is realized by one provider's call

(b) requester's request can be realized in different ways which may or may not be equivalent (e.g., different services can be used to to satisfy requester's requirements)
(c) reuse of information: information provided by the requester is used in different place in the provider's process model (similar to message reordering)
(d) missing information: some information required by the provider is not provided by the requester
(e) redundant information: information provided by one party is not needed by the other one

3. *Protocol / structural level mismatches*: control flow in the requester's process model can be realized in very different ways in the provider's model (e.g., sequence can be realized as an unordered list of steps, etc.)

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.

Mediation components (or mediators) can be used to resolve other types of incompatibilities. In our work we distinguish two types of mediation: *data mediation* and *process mediation*. We assume, that *data mediators* are responsible for resolving data level mismatches (category 1) while *process mediators* are responsible for resolving service level and protocol level mismatches (categories 2 and 3).

Typically, when trying to achieve interoperability, process mediators and data mediators are closely related. A natural way is to use data mediators within the process mediation component to resolve "lower" level mismatches that were identified during the process mediation. As opposed to WSMO methodology [5], OWL-S does not introduce mediators as first class objects. [20] shows that in the OWL-S framework mediators can be naturally represented as web services and described in the same way as any other web service. This is particularly the case for data mediators, which typically work as transformation functions from one domain into the other and hence can be easily described as atomic processes by specifying appropriate input and output types. We comply with this approach since we believe that the view of mediators as web services naturally fits into the Semantic Web Services Architecture [21] and allows us, for example, to use the same discovery and invocation mechanisms for mediators as for any other services.

### 3.1 Process Mediation

When requesters and providers use fixed, incompatible communication protocols interoperation can be achieved by applying a *process mediation component* which resolves all incompatibilities, generates appropriate mappings between different processes and translates messages exchanged during run-time. Figure 2 shows this problem setting.

Implementing the mediation component is complicated and costly, since it has to address all different types of incompatibilities described in the previous section. We describe mechanisms how the process mediation can be solved automatically. In particular, we show how the workflow and dataflow mismatches can be resolved.

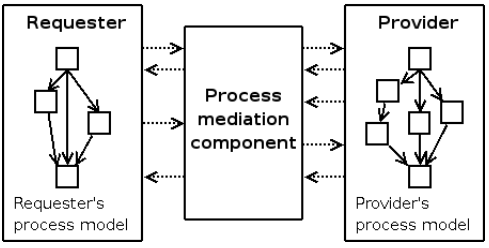


**Fig. 2.** Mediation of process models

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[2]. Because the problem of process mediation is complex and extensive, we address the data mediation (mismatches of type 1) only in a very limited way. For details on the data mediation see, e.g., [22,23,24]. We assume that data mediators are given to the process mediation component as an input. In our system data mediators can have a form of a converter that is built-in to the system or of an external web service [20]. We support basic type conversions 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.

We assume that both process models describe services that belong to the same domain. By this we mean that inputs, outputs, preconditions and effects are defined in the same ontology and that both partners target conceptually 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. Both process models could be using the same ontology but the mediation would not make much sense in this case. This requirement can be easily achieved either by appropriate service discovery mechanisms [25] or simply be a consequence of the real situation when only applications from within the same domain need to be integrated.

### 3.2 Motivating Example

Figure 4 depicts a fragment of the process model of a hypothetical provider from the flights booking domain. The requester's model, presented in Figure 3, represents a straightforward process of purchasing a ticket from some airlines booking web service, while the provider's process model represents a more elaborate scenario that allows the requester, besides booking the flight, to also rent a car or to book a hotel. Boxes in figures represent atomic processes with their inputs, outputs, preconditions, and effects, while ovals stand for control constructs. The control flow proceeds in the top-down and left to right direction. Inputs and outputs types used in process models refer to a very

[2] OWL-S process model pertains mainly to describing service providers. However, its constructs can be used to describe the requester in the same way as if describing the provider. The only conceptual difference in using the OWL-S process model to describe the behavior of a requester is that it describes the behavior the requester expects a provider to have.

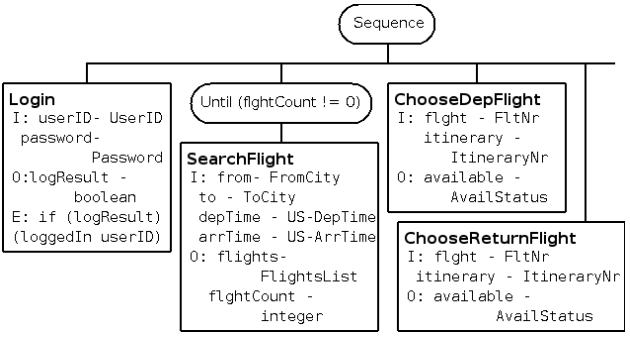


**Fig. 3.** Requester's process model

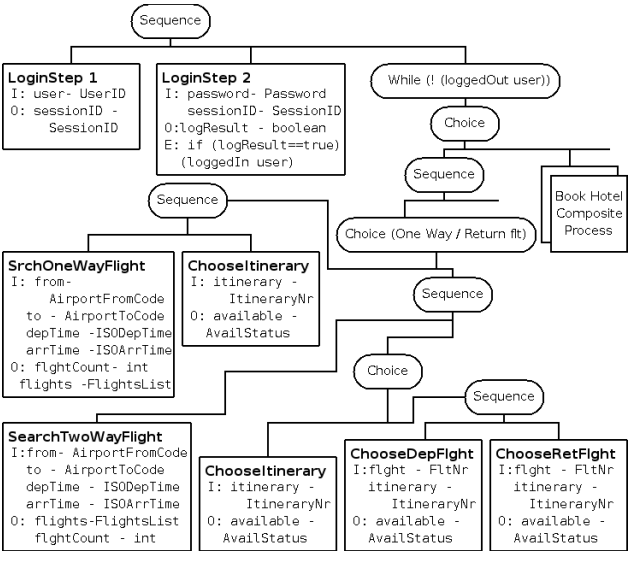


**Fig. 4.** Provider's process model

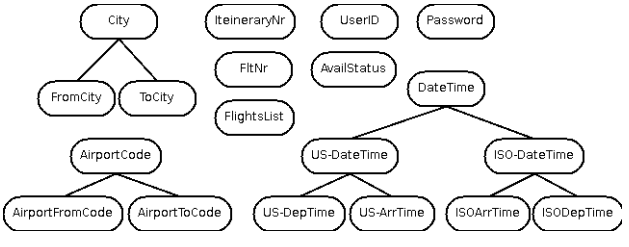


**Fig. 5.** Simple flights domain ontology

simple ontology showed in Figure 5 (ovals represent classes and lines represent subsumption relations). 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 the conditional effect saying that the predicate *(loggedIn userID)* will become true if the value of *logResultOutput* equals to true. In the next step, the *SearchFlight* is executed within the repeat-until loop which is repeated until some flight is found. Similarly the process continues by executing other atomic processes.

This example demonstrates several types of inconsistencies that we have to deal with. The *Login* step in the requester's model is represented by two separate atomic processes in the provider's model (mismatches 2a, 2e, see Section 3). Types of inputs and outputs do not always match exactly, e.g., *AirportFromCode* and *FromCity* are not directly related in the ontology (mismatch 1b), *US-DepTime* and *ISODepTime* based on ISO 8601 are subclasses of the common superclass *DateTime* (mismatch 1b). *SearchFlight* in the requester's process model can be mapped either to *SrchOneWayFlight* or to *SrchTwoWayFlight* (mismatch 2b). Finally, the structure of both processes is quite different and it is not obvious at first glance whether the requester can be mapped into the provider's process model (mismatch 3).

## 4 Process Mediation in the Semi-open World

An environment in corporations that need to dynamically interact with services provided by other trusted partners, e.g., their subcontractors can be usually characterized as a semi-open environment. From the point of view of process mediation several characteristics of semi-open environments are important:

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

These characteristics allow us to perform the process mediation of the given requester and the provider in two steps: (1) during the registration process of the component to the system an analysis of the process models of the requester and the provider is performed to find possible mappings between provider's and requester's process models, or to identify incompatibilities that cannot be reconciled with given set of available data mediators and external services. Results of the analysis for the given requester/provider pair are stored. (2) During run-time, when some requester and provider need to interact, saved mediation mappings for this pair are used in the mediator runtime component.

### 4.1 Execution Paths Analysis Approach

The problem of process mediation can be seen as finding an appropriate mapping between requester's and provider's process models. The mapping can be constructed by combining simpler transformations representing different ways of bridging described mismatches. We need to decide if *structural differences* between process models can be resolved. Assuming that the requester starts to execute its process model, we want to show that for each step of the requester the provider (with some possible help of intermediate translations represented by *data mediators* or *built-in conversions*) can satisfy the requester's requirements (i.e., providing required outputs and effects) while respecting its own process model. This can be achieved by exploring possible sequences of steps (execution path) that the requester can execute.

*Requester's execution path* is any sequence of atomic processes which can be called by the requester in accordance with its process model, starting from the process model first atomic process and ending in 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 (assuming some data translation facility). If there exists a possible requester's execution path which could not be mapped to any part of the provider's process model, we would know that if this path were chosen, the mediation would fail. Thus the existence of a mapping for each possible requester's execution path is a necessary precondition of successful mediation. Indeed, it is only a necessary condition of successful process mediation for the following reason. Since the possible mappings are being searched before actual execution, some of them can turn out not to work during execution (e.g., because of failing preconditions of some steps). Still, by analyzing requester's execution paths and trying to find mappings for them, we can partially answer the question of mediation feasibility.

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

Finally, during the execution we need to decide, what actions should be performed in each given state. Generated mappings are used to decide, if and what services of the provider's process model should be executed, or if a translation (or a chain of translations) is necessary after the requester executes each step.

### 4.2 Mediator Algorithm Overview

The following procedure provides a top-level view of the whole process mediation:
**1. Generate requester's paths:** based on the process model of the requester, possible requester's paths are generated (see Section 4.3)
**2. Filter out those requester's paths that need not be explored:** as the result we get the *minimal set of requester's paths*. (see Section 4.3)

**3. Find all appropriate mappings to the provider's process model for each requester's path from the minimal set of paths and store them in the *mappings repository*:** if for a path no mapping is found, user is notified with pointing out the part of the path for which the mapping was not possible[3]. (see Section 4.4)

During the mediation process following steps are performed to choose the best available action and to execute it:

**1. Retrieve possible actions from the *mappings repository*** that are available in this context
**2. Remove inconsistent actions:** actions that are not consistent with actual variables bindings (e.g., preconditions fail)
**3. When no suitable mediation action is available, fail**
**4. When more actions are available, choose the best:** Having execution paths and mappings precomputed, we can easily figure out, if the suggested mediation action, if chosen, allows to finish all paths that can be taken by the requester from this state of execution. If there is no such an action, we choose the one that allows to finish the most paths.
**5. Execute selected action:** depending on the type of an action either the *OWL-S Virtual Machine* [18] is called to execute the external service or the provider's atomic process, or the built-in converter is called, or a response to the requester is generated by the mediation component.
**6. Update state of the mediator:** *mappings repository* and variables values and valid expressions are updated.

### 4.3 Generating the Minimal Set of Requester's Execution Paths

When generating requester's execution paths we potentially have to deal with combinatorial explosion caused by chains of branching in the requester's process model. We want to find out what reconciliation actions are available or necessary in given state of execution which depends on possible combinations of available variables and valid expressions in this state. Because the current state depends on actions performed preceding this state, we might be in principle interested in every possible requester's path. In [26] we describe some 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.

### 4.4 Finding Mappings for the Requester's Path

In order to find all the mappings for a given requester's path we simulate the execution of the provider's process model and try to map each step of the requester's path to some part of the provider's model (atomic process or several atomic processes) with help of *data mediators*. If some step of the requester's path cannot be mapped to the provider's process model, the simulation backtracks to the last branching (e.g., *choice* or *any-order*). The mapping is constructed during the simulation and is represented as

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

a sequence of actions that the mediation component should execute during the runtime mediation (see Figure 6 for an example of a mapping).

The *reconciliation algorithm* for the given requester's path works as follows:

**Input:** requester's path *requesterStepsSequence*

1. Initialize the simulator state by adding *requesterStepsSequence* to it
2. Call *executeNextRequesterCalls* method
3. Simulate the execution of the provider's process model until no requester's steps need to be reconciled, or the provider finishes, or reconciliations fails
   - when the atomic process $P$ is reached during simulation, call the *reconciliation method for P*

The *executeNextRequesterCalls* is a simple method that removes the first call from the requester's path *requesterStepsSequence* and adds inputs of this call to the simulator's state and for each output and effect that are expected to be produced creates an appropriate goal.

The *reconciliation method for a provider's atomic process P* first tests if all inputs of process $P$ are available in the simulator's state and all preconditions are satisfied. In such a case it simulates the execution of the atomic process $P$. If some inputs or preconditions of $P$ are missing, a backward chaining algorithm is used to find a *combination of data mediators* which can provide missing inputs or preconditions. If some of the outputs or effects required by the requester are missing after $P$ is executed, the same backward chaining algorithm is used to find a *combination of data mediators* which can translate generated outputs or generate missing effects. After the process $P$ is reconciled, the *executeNextRequesterCalls* is called to simulate next requester's call. Details of the reconciliation algorithm can be found in [26].

**The requester's path:**
Login, SearchFlight, ChooseDepFlight, ChooseRetFlight, ...
**A possible mapping for first two steps:**
*requester-Login* s1-userID s1-password
*provider-LoginStep1* s1-user sessionID
*provider-LoginStep2* s1-password sessionID logResult
*mediator-prepare-to-send* logResult
*mediator-send*
*requester-SearchFlight* s2-from s2-to s2-depTime s2-arrTime
*external-AirportCityToCode* s2-from apt-code-gener1
*mediator-explicit-down-casting* apt-code-gener1 AirportToCode
*external-AirportCityToCode* s2-to apt-code-gener2
*mediator-explicit-down-casting* apt-code-gener2 AirportToCode
*external-USTimeToISO* s2-depTime iso-time-gener1
*mediator-explicit-down-casting* iso-time-gener1 ISODepTime
*external-USTimeToISO* s2-arrTime iso-time-gener2
*mediator-explicit-down-casting* iso-time-gener2 ISODepTime
*provider-SearchReturnFlight* apt-code-gener1 apt-code-gener2 iso-time-gener1 iso-time-gener2 flights flght-Count
*mediator-prepare-to-send* flights
*mediator-prepare-to-send* flghtCount
*mediator-send*

**Fig. 6.** Example solution for a requester's path

### 4.5 Example Mapping

Figure 6 shows part of one mapping generated for a requester's execution path that can be executed by a requester as defined in Figure 3 in Section 3.2. The mapping was generated for a provider's process model defined in Figure 4. This example assumes that we have provided the system with the *AirportCityToCode* external web service for translating instances of *City* to instances of *AirportCode*, and the service *USTimeToISO* for translating between US and ISO time formats. Each step name is prefixed by *requester*, *provider* and *external* to indicate to which component it is related. Requester's steps show names of inputs parameters, while for the provider, translators and external services also output variables are included. This example also illustrates implicit up-casting of types and explicit down-casting which is enforced by the fact, that *AirportCityToCode* and *USTimeToISO* are defined to work with more generic types than those provided by requester and requested by the provider. Due to the requirement for explicit down-casting, the user is prompted whether the chosen casting is allowed or not. In this example all the castings are allowed. See [22] for details on analyzing casting operations for ontology classes.

## 5 Process Mediation in the Open World

Dynamic open world of the Internet imposes some restrictions on the mediation process. Since components can appear and disappear completely arbitrarily, it is necessary to incorporate appropriate discovery mechanisms. It is not possible to perform an analysis step in advance because requesters and providers are not known in advance. Therefore the automatic mediation process must rely only on the run-time mediation. Also reasoning possibilities are quite restricted during runtime, since the mediation component must respond instantly and it cannot afford to delay the execution of the requester.

In Section 5.1 we first describe a standalone Runtime Process Mediation Service (RPMS). This component works as an ordinary web service. Therefore, it can for example, be registered with discovery registries and discovered later. In this scenario we assume that the requester first contacts a discovery registry (e.g., the OWL-S/UDDI matchmaker [11]) to find an appropriate provider and then it uses the the RPMS (which also can be discovered) to mediate interactions with the discovered provider.

Next, we show how the Process Mediation Service can be integrated into a Broker component that performs both, the discovery of suitable service providers and the runtime mediation (Section 5.2).

### 5.1 Runtime Process Mediation Service

Similar to the approach in the semi-open world, the Runtime Process Mediation Service needs to select an appropriate mediation action after it receives a request from the requester. In the run-time scenario, however, the mediation service cannot use any analysis of process models. The only things that the mediation service can see are the current request message received from the requester, the provider's process model, the state of the execution of the provider and a set of available data mediators. Available mediation actions include (1) executing an appropriate service (atomic process) of the provider's

process model, (2) applying an appropriate translation (i.e., executing a *data mediator* or a *built-in conversion*) to some inputs provided by the requester or outputs returned by the provider, and (3) preparing required results and sending them to the requester.

Figure 7 shows an architecture of the *Runtime Process Mediator Service*. The *server port* is used for interaction with the requester and the *client port* for interaction with the provider. The *client port* uses OWL-S Virtual Machine to interact with the provider. Another instance of the OVM is used to execute external data mediation services if it is necessary. The *Execution Monitor* is the central part of the RPMS. It executes the mediation algorithm and links all the other components together. It uses the *Knowledge Base (KB)* to store information about received inputs and produced outputs.

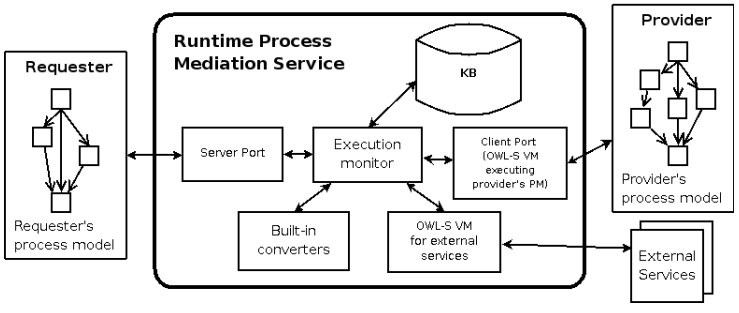


**Fig. 7.** Runtime Process Mediation Service architecture

The following high-level steps represent a mediation procedure executed by the *Execution Monitor* in the *Runtime Process Mediation Service*:

**for each** requester's request
**do** *processRequest*
**until** requester or provider finishes successfully or the execution fails

Procedure *processRequest:*
1. Receive a requester's atomic process call *requesterCall* via the *server port*
2. Store inputs of *requesterCall* and required results in the *KB*
3. **While** required results (outputs and effects) of *requesterCall* are not available
   3.1 Choose the best available mediation action *A*
   **if** some provider's atomic process that can be executed and that matches inputs provided by the requester and produces required results[4] **then** select the atomic process as the mediation action *A*
   **else** find a *data mediator* or a *combination of available data mediators* that are able to translate between the *requesterCall* atomic process and some of the possible provider's atomic processes

[4] In [27] we described extensions of the OVM that support monitoring and execution introspection which allows an easy tracking of the current execution context. It is for example possible to use the OVM during the execution to get a list of atomic process that are allowed by the process model in the given execution context.

**if** some combination is found **then** use it as the mediation action *A*
**else** fail

3.2 Execute action *A*
- **if** *A* is an atomic process of the provider's process model **then** use the OVM of the *client port* to execute it
- **elseif** *A* is a data mediator represented by an external service **then** use OVM for external services to execute it
- **elseif** *A* is a built-in conversion **then** use built-in converters
- **elseif** *A* is return results to requester **then** use Requester's *server port*

3.3 Update the *KB*

4. Return required results of *requesterCall* to the requester

The most obvious shortcoming of the run-time approach is the fact that it uses only current state information without analyzing requester's and provider's process models. While this local reasoning is efficient, in cases when more mediation actions are available it can lead to choosing a wrong mediation action that eventually causes the failure of the whole mediation process.

Consider the situation of the requester from Figure 3 in which it executes the *SearchFlight* atomic process. Let us assume that this step can be mapped either to the *SearchOneWayFlight* process or the *SearchReturnFlight* process of the provider. If the mediator used only the current state information (as, e.g., in [28]), these two options would appear as indistinguishable since there is no difference in their IOPEs. 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.

In the previous example the runtime mediation failed because *SearchOneWayFlight* process and the *SearchReturnFlight* process have exactly the same set of inputs and outputs with the same types, and are therefore indistinguishable for the runtime mediation service. Even though such a situation is possible in real-life process models, we believe that in well formed process models it is rather an anomaly than a common situation.

### 5.2 Broker Hybrid Discovery and Mediation

Although the Runtime Process Mediation Service can be used as a standalone web service, it can be convenient to embed it into the Broker component. The Broker combines the mediation with the discovery process and thus simplifies the interaction process of the provider because it does not have to contact a discovery and mediation service separately. Also the Broker can maintain anonymity of the provider and the requester.

In this paper we build on the broker system described in [29] and extend it by incorporating the process mediation component. We adopt the definition of the Broker protocol based on [13], as graphically summarized in Figure 8. Any transaction involving Broker requires three parties. The first party is a *requester* that initiates the transaction by requesting information or a service to the Broker. The second party is a *provider* which is selected among a pool of provider as the best suited to resolve the problem of the requester. The last party is the *Broker* itself.

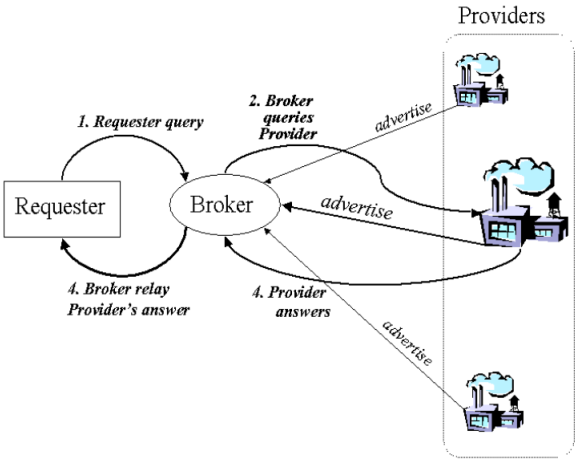


**Fig. 8.** The broker protocol

The protocol in Figure 8 can be divided in two parts: the *advertisement protocol*, and the *mediation protocol*. In the advertisement protocol, the Broker first collects the advertisements of Web service providers that are available to provide their services. These advertisements, shown in Figure 8 by straight thin lines, are used by the Broker to select the best provider during the interaction with the requester. The mediation protocol, shown in Figure 8 using thick curve lines, requires (1) the requester to query the Broker and wait for a reply while the Broker uses its discovery capabilities to locate a provider that can answer the query. Once the provider is discovered, (2) the Broker reformulates the query for that provider, and finally queries it. Upon receiving the query, (3) the provider computes and sends the reply to the Broker and finally (4) the Broker replies to the requester.

In general, the execution of the protocol may be repeated multiple times. For example, the requester may have asked the Broker to book a flight from Pittsburgh to New York. Since there are multiple flights between the two cities, the provider may ask the Broker, and in turn the requester, to select the preferred flight. These interactions are resolved with multiple loops through the protocol. For example, the Broker translates the list of flights retrieved by the provider for the requester, through steps (3) and (4) of the protocol, and then translates the message with the selected flight from the requester to the provider, via steps (1) and (2) of the protocol. The only exception is that step (1) does not require any discovery since the provider is already known.

The protocol described above shows that the Broker needs to perform a number of complex reasoning tasks for both the discovery and mediation part of its interaction. The discovery task requires the Broker to use the query to describe the capabilities of the desired providers that can answer that query, and then match those capabilities with the capabilities advertised by the providers. During the mediation process, the Broker needs to interpret the messages that it receives from the requester and the provider to

decide how to translate them, and whether it has enough information to answer directly the provider without further interaction with the requester. If the requester and provider use fixed, incompatible protocols, the process mediation component as described in the previous section can be used in the broker to mediate between the requester and the provider. In the next two sections, we will analyze these reasoning tasks in more detail.

**Discovery of Providers.** The task of discovery is to select the provider that is best suited to reply to the query of the requester. Following the protocol, providers advertise their capabilities using a formal specification of the set of capabilities they posses, i.e. the set of functions that they compute. These capability specifications implicitly specify the type of queries that the provider can answer.

The discovery process requires two different reasoning tasks. The first one is to abstract from the query of the requester to the capabilities that are required to answer that query. The second process is to compare the capabilities required to answer the query with the capabilities of the providers to find the best provider for the particular problem.

The first problem, the abstraction from the query to capabilities, is a particularly difficult one. Capabilities specify what a Web service or an agent does, or, in the case of information providing Web services, what set of queries it can answer. For example, the capability of a Web service may be to provide weather forecasting, or sell books, or register the car with the local department of transportation. Queries instead are requests for a very specific piece of information. For example, a query to a weather forecasting agent may be to provide the weather in Pittsburgh, while a query to a book-selling agent may be to buy a particular book. The task of the Broker therefore is to abstract from the particular query, to its semantics, i.e. what is really asked. Finally, the Broker must identify and describe in a formal way the capabilities that are needed to answer that query. The abstraction mechanism is described in detail in [29].

As an alternative to abstracting from the query, the requester could specify its request directly in terms of required provider's capabilities before sending the specific initial query. This would allow to skip the abstraction step in the broker. Such a solution is appropriate, for example, in a situation when the initial broker query does not provide enough information to select an appropriate provider.

The second task of the discovery process is to match the capabilities required to answer the query with the advertisements of all the known providers. Since it is unlikely that the Broker will find a provider whose advertisement is exactly equivalent to the request, the matching process can be very complicated, because the Broker has to decide to what extent the provider can solve the problems of the requester. The matching of the service request against the advertised capabilities was implemented using the OWL-S matching engine reported in [11] and [12].

**Management of Mediation.** Once the Broker has selected a particular provider, the second reasoning task that the Broker has to accomplish is to transform the query of the requester into the query to send to the provider. This process of mediation has two aspects. The first one is the efficient use of the information provided by the requester to the Broker; the second one is the mapping from the messages of the requester to messages to the provider and vice versa.

Since the requester does not a priori know when it issues the initial query which is the relevant provider, the (initial) query it sends to the Broker and the query input that the (selected) provider may need in order to provide the service may not correspond exactly. The requester may have appended to the query information that is of no relevance to the provider, while the provider may expect information that the requester never provided to the Broker. In the example above, we considered the example of a requester that asks to book the cheapest flight from Pittsburgh to New York. However, besides the trip origin and destination, the selected provider may expect date and time of departure. In the example, the requester never provided the departure time, and the provider has no use for the "cheapest" qualifier. It is the task of the Broker to reconcile the difference between the information that the requester provided and the information that the provider expects, by (1) recognizing that the departure time was not provided, and therefore it should be asked for, and (2) finding a way to select the cheapest flight among the ones that the provider can find.

The approach to mediation proposed in [29] assumes that the requester is flexible enough to be able to provide missing pieces of information that are required by the provider. In our example, the requester must be able to provide the departure time. The problem occurs, when the requester is not flexible because it uses a fixed protocol. It could be perfectly possible that the requester would not be able to provide the departure time immediately after the first query was processed because its own protocol dictates the provider to do something else, as e.g. providing the arrival time first and the departure time afterwards. In such a case the broker must take also the requester's process model into considerations to accomplish mediation.

In general, during mediation, the broker has to deal with exactly the same problems that we identified in Section 3 on process mediation and therefore incorporating a process mediation component into the broker is a natural extension of the original broker implementation.

**The Broker Architecture.** The overall architecture of the Broker incorporating the mediation component is shown in Figure 9. It is based on the broker architecture described in [29]. To interact with the provider and the requester the Broker instantiates two ports: a *server port* for interaction with the requester (since the Broker acts as a provider vis a vis the requester) and a *client port* for interaction with the provider (since the Broker acts as a client vis a vis the provider).

The *client port* uses the *OWL-S Virtual Machine* (OVM) [18] to interact with the provider. The *server port* which is used to communicate with the requester exposes the process model that is equivalent with the requester's process model. Specifically, the *server port* receives messages sent by the requester and replies to them in the format that corresponds to definitions of the provider's Process Model.

The reasoning of the Broker happens in the *Execution Monitor / Query Processor* that is responsible for translation of messages between the two parties and for the implementation of the mediation algorithms. Specifically, the *Execution Monitor / Query Processor* stores information received from the requester in a *Knowledge Base* that is instantiated with the information provided by the requester. Furthermore, the *Execution Monitor / Query Processor* interacts with the *Discovery Engine,* which provides the

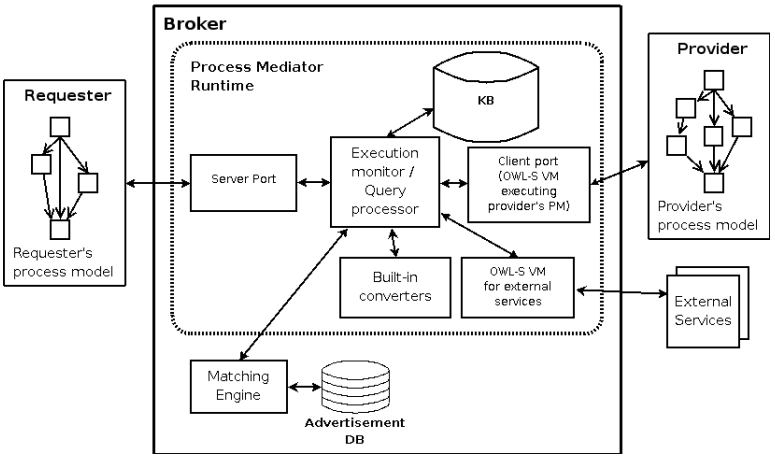


**Fig. 9.** The hybrid broker architecture

storage and matching of capabilities, when it receives a capability advertisement and when it needs to find a provider that can answer the query of the requester.

## 6 Related Work

The work in [30] provides a conceptual underpinning for automatic mediation. The work closest to ours is in [28]. Mediation between two WSMO based process is performed strictly during runtime. Besides structural transformations (e.g., change of message order) also data-mediators can be plugged into the mediation process. Aberg et. al [31] describes an agent called sButler for mediation between organizations' workflows and semantic web services. The mediation is more similar to the 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 [29] 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. The work in [32] and [33] 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). Authors in [34] apply a model-driven approach based on WebML language. Mediator is designed in the high-level modeling language which supports semi-automatic elicitation of semantic descriptions in WSMO. In [22], data transformation rules together with inference mechanisms based on inference queues are used to derive possible reshapings of message tree structures. An interesting approach to translation of data structures based on solving higher-order functional equations is presented in [23] while [24] argues for published ontology mapping to facilitate automatic translations.

## 7 Conclusions and Further Work

In this paper we described mediation mechanisms of two OWL-S process models that can be applied in different environments. Specifically, we considered a semi-open environment that frequently occurs for inter and intra corporation business process. We described an algorithm based on the analysis of provider's and requester's process models for deciding if the mediation is possible and for performing the runtime mediation in the semi-open environment. Next, we described a Runtime Process Mediation Service that can be used in an open environment of the Internet. Finally we demonstrated how discovery can be combined with the process mediation within the Broker component. Embedding discovery and process mediation into the Broker facilitates interactions between requesters and providers because requesters do not have to contact discovery and mediation services separately and Broker further allows to maintain anonymity of the requester and the provider.

In future work, we will address some of the current limitations such as a better support for data mediation and improve efficiency. We want to explore a user assisted mediation and the top-down analysis of process models to allow more selective exploration of requester's paths and so to better deal with possible combinatorial explosion. Also we want to explore the possibility of discovering data mediators during runtime.

## References

1. Christensen, E., Curbera, F., Weerawarana, G.M.S.: Web Services Description Language (2001)
2. Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., et al.: Business Process Execution Language for Web Services, Version 1.1 (2003)
3. Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Schmidt, M.T., Sheth, A., Verma, K.: Web Service Semantics - WSDL-S (2005), http://www.w3.org/Submission/WSDL-S/
4. The OWL Services Coalition: Semantic Markup for Web Services (OWL-S), http://www.daml.org/services/owl-s/1.1/
5. Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., Fensel, D.: Web Service Modeling Ontology. Applied Ontology 1(1), 77–106 (2005)
6. Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N.: Automated discovery, interaction and composition of semantic web services. Journal of Web Semantics 1 (1), 27–46 (2004)
7. Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL web ontology language reference, W3C Recommendation (February 10, 2004), http://www.w3.org/TR/owl-ref/
8. SOAP: Simple object access protocol (SOAP 1.1), http://www.w3.org/TR/SOAP
9. Wong, H.C., Sycara, K.P.: A taxonomy of middle-agents for the internet. In: ICMAS, pp. 465–466. IEEE Computer Society, Los Alamitos (2000)
10. OASIS: Universal description, discovery and integration v3.0.2 (2005), http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.htm
11. Paolucci, M., Kawmura, T., Payne, T., Sycara, K.: Semantic matching of web services capabilities. In: First Int. Semantic Web Conf. (2002)

12. Paolucci, M., Sycara, K.P., Kawamura, T.: Delivering semantic web services. In: WWW (Alternate Paper Tracks) (2003)
13. Decker, K., Sycara, K., Williamson, M.: Matchmaking and brokering. In: Proceedings of the Second International Conference on Multi-Agent Systems. The AAAI Press, Menlo Park (1996)
14. Lu, J., Mylopoulos, J.: Extensible information brokers. International Journal on Artificial Intelligence Tools 11(1), 95–115 (2002)
15. Jennings, N.R., Norman, T.J., Faratin, P., O'Brien, P., Odgers, B.: Autonomous agents for business process management. Applied Artificial Intelligence 14(2), 145–189 (2000)
16. Chen, H., Finin, T.W., Joshi, A.: Semantic web in the context broker architecture. In: Proceedings of the IEEE Conference on Pervasive Computing and Communications (PerCom), pp. 277–286. IEEE Computer Society, Los Alamitos (2004)
17. Motta, E., Domingue, J., Cabral, L., Gaspari, M.: IRS-II: A Framework and Infrastructure for Semantic Web Services. In: Fensel, D., Sycara, K.P., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 306–318. Springer, Heidelberg (2003)
18. Paolucci, M., Ankolekar, A., Srinivasan, N., Sycara, K.P.: The DAML-S virtual machine. In: Fensel, D., Sycara, K.P., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 290–305. Springer, Heidelberg (2003)
19. Ankolekar, A., Huch, F., Sycara, K.P.: Concurrent semantics for the web services specification language DAML-S. In: Arbab, F., Talcott, C.L. (eds.) COORDINATION 2002. LNCS, vol. 2315, pp. 14–21. Springer, Heidelberg (2002)
20. Paolucci, M., Srinivasan, N., Sycara, K.: Expressing wsmo mediators in owl-s. In: International Semantic Web Conference (2004)
21. Burstein, M., Bussler, C., Finin, T., Huhns, M., Paolucci, M., Sheth, A., Williams, S., Zaremba, M.: A semantic web services architecture. Internet Computing. IEEE 9(5), 72–81 (2005)
22. Spencer, B., Liu, S.: Inferring data transformation rules to integrate semantic web services. In: International Semantic Web Conference, pp. 456–470 (2004)
23. Burstein, M., McDermott, D., Smith, D.R., Westfold, S.J.: Derivation of glue code for agent interoperation. Derivation of glue code for agent interoperation V6(3), 265–286 (2003)
24. Burstein, M.H., McDermott, D.V.: Ontology translation for interoperability among semantic web services. The AI Magazine 26(1), 71–82 (2005)
25. Sycara, K.P., Klusch, M., Widoff, S., Lu, J.: Dynamic service matchmaking among agents in open information environments. SIGMOD Record 28(1), 47–53 (1999)
26. Vaculín, R., Sycara, K.: Towards automatic mediation of OWL-S process models. In: 2007 IEEE International Conference on Web Services, pp. 1032–1039. IEEE Computer Society, Los Alamitos (2007)
27. Vaculín, R., Sycara, K.: Monitoring execution of OWL-S web services. In: European Semantic Web Conference, OWL-S: Experiences and Directions Workshop, June 3-7 (2007)
28. Cimpian, E., Mocan, A.: WSMX process mediation based on choreographies. In: Business Process Management Workshops, pp. 130–143 (2005)
29. Paolucci, M., Soudry, J., Srinivasan, N., Sycara, K.: A broker for owl-s web services. In: Cavedon, M., Martin, B. (eds.) Extending Web Services Technologies: the use of Multi-Agent Approaches. Kluwer, Dordrecht (2005)
30. Wiederhold, G., Genesereth, M.R.: The conceptual basis for mediation services. IEEE Expert 12(5), 38–47 (1997)
31. Aberg, C., Lambrix, P., Takkinen, J., Shahmehri, N.: sButler: A Mediator between Organizations Workflows and the Semantic Web. In: World Wide Web Conference workshop on Web Service Semantics: Towards Dynamic Business Integration (2005)
32. Cabral, L., Domingue, J., Galizia, S., Gugliotta, A., Tanasescu, V., Pedrinaci, C., Norton, B.: IRS-III: A Broker for Semantic Web Services Based Applications (2006)

33. Domingue, J., Galizia, S., Cabral, L.: Choreography in irs-iii - coping with heterogeneous interaction patterns in web services. In: Proc. 4th Intl. Semantic Web Conference (2005)
34. Brambilla, M., Celino, I., Ceri, S., Cerizza, D., Valle, E.D., Facca, F.M.: A software engineering approach to design and development of semantic web service applications. In: Cruz, I.F., Decker, S., Allemang, D., Preist, C., Schwabe, D., Mika, P., Uschold, M., Aroyo, L. (eds.) ISWC 2006. LNCS, vol. 4273, pp. 172–186. Springer, Heidelberg (2006)