Towards automatic mediation of OWL-S process models*

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

Abstract

The framework for automatic mediation of two process models composed of semantically annotated web services is presented. Process mediation is hard because of many possible mismatches between process models. We introduce algorithms for the process models analysis to find possible mappings between provider's and requester's process models, or to identify incompatibilities that cannot be reconciled with given set of available data mediators and external services. Results of the analysis phase are used in the mediator runtime component. In particular, we show how the workflow and dataflow mismatches can be resolved.

1 Introduction

One of the main promises of web services standards is to enable and facilitate seamless interoperation of diverse applications and business processes implemented as components or services. Many existing services are programmed to exchange data according to a specified protocol. A service can be part of a process model that prescribes control and data flows and thus defines the overall functionality of a more complex application.

However, as business needs change, processes could get reconfigured or additional process components and services may need to be added. As a result of these changes, the previous components must become interoperable with the new one. The possibility of achieving interoperability of existing processes without actually modifying their implementation and interfaces is desirable and often necessary. In such cases, interoperation can be realized by applying a process mediation component which 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.

Current web services standards provide a good basis for achieving at least some level of mediation. WSDL [8] standard allows to declaratively describe operations and format of messages and data structures that are used to communicate with the web service. BPEL4WS [3] adds the possibility to combine several web services within a formally defined process model, and so to define the interaction protocol and possible control flows. However, neither of these two standards goes beyond the syntactical descriptions of web services. Newly emerging standards for semantic web services as WSDL-S [2], OWL-S [18] and WSMO [14] 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, e.g., matchmaking and composition [16].

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. When the requester is supposed to be used with some provider with a given process model, it can be programmed specifically for this provider. However, when the provider needs to be changed or a new provider is added, a new requester must be programmed. Another approach, that we are advocating in this paper, is to assume, that the requester behaves according to a specified process model that is expressed explicitly. This requester's process model can either correspond to a particular implementation of the requester or it can be a default process model that for example generalizes a business process of a generic requester solving some problem (e.g., generic flights booking client). When a new provider has to be used, we can use its process model and the reqester's pro-

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

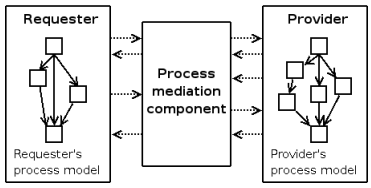

Figure 1. Mediation of process models

cess model in the process mediation component that tries to resolve all mismatches based on the analysis of both process models.

Because the problem of process mediation is complex and extensive, we do not address the data level mediation in detail (see, e.g., [15, 5, 6]). However, we define a framework, in which data mediation has its place and data mediators can be easily incorporated. We use the OWL-S ontology for semantic annotations because it provides good support for description of individual services and also explicit constructs with clear semantics for describing process models.

The rest of the paper is organized as follows. In Section 2, we define the problem and provide an example scenario. Section 3 gives an overview of our approach. In Sections 4 and 5 algorithms for finding mappings between process models are described and pruning possibilities are discussed. Section 6 summarizes related work, and Section 7 contains conclusions and future work.

2 Problem definition

Figure 1 shows our problem setting. We assume that both process models of the requester and the provider are described using OWL-S ontology[1]. The problem of creating a process mediator can be formulated in terms of finding the translations / mappings between these two process models. 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, 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 provider's services or a sequence of requester's calls is realized by one provider's call
   (b) requester's request can be realized in different ways which may or may not be equivalent (e.g., different services can be used to to satisfy requester's requirements)
   (c) reuse of information: information provided by the requester is used in different place in the provider's process model (similar to message reordering)
   (d) missing information: some information required by the provider is not provided by the requester
   (e) redundant information: information provided by one party is not needed by the other one
3. *Protocol / structural level mismatches*: control flow in the requester's process model can be realized in very different ways in the provider's model (e.g., sequence can be realized as an unordered list of steps, etc.)

We assume that both process models describe services that come from the same domain. By this we mean that inputs, outputs, preconditions and effects are defined in the same ontology and that the original and the new provider solve 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 [17] or simply be a consequence of the real situation when only applications from within the same domain need to be integrated.

In this paper, we address the data mediation (mismatches of type 1) only in a very limited way. 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 [13]. We support basic type conversions (as upcasting and down-casting) based on reasoning about types of inputs and outputs.

2.1 OWL-S

OWL-S [18] is a semantic web services description language. OWLS covers three areas: the Service Profile describes what the service does in terms of its capabilities and is used for discovery purposes; the Process Model specifies ways of how clients can interact with the service; the Grounding links the process model to the specific execution infrastructure (e.g., maps processes to WSDL operations and allows for sending messages in SOAP). The elementary unit of the Process Model is an atomic process, which represents one indivisible operation that the client can per-

[1] OWL-S process model pertains mainly to describe 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.

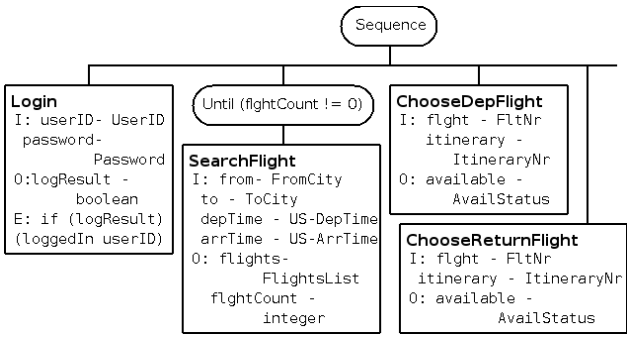

Figure 2. Requester's process model

form 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 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: sequences, 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.

2.2 Motivating example

Figure 3 depicts a fragment of the process model of a hypothetical provider from the flights booking domain. The requester's model, presented in Figure 2, 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, and 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 simple ontology showed in the Figure 4 (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.

Our 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 pro-

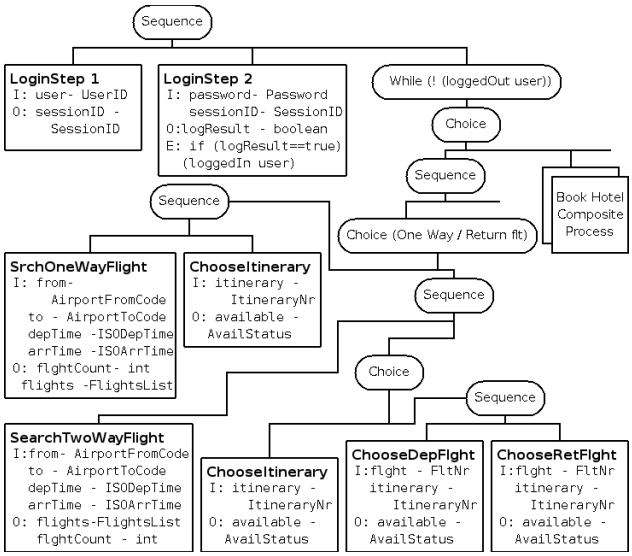

Figure 3. Provider's process model

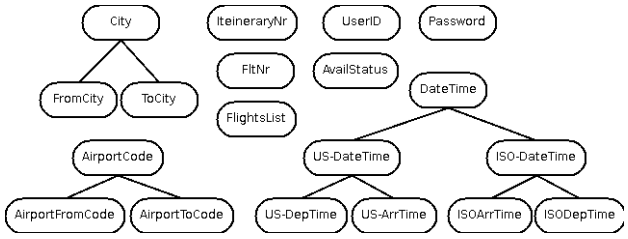

Figure 4. Simple flights domain ontology

cesses in the provider's model (mismatches 2a, 2e, see Section 2). 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 sight whether the requester can be mapped into the provider's process model (mismatch 3).

3 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[2] of the requester the provider (with some possible help of intermediate translations) 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.

Definition: *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 loops).

Since any of all possible requester's execution paths can be chosen, 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[3] 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 the runtime mediation component must decide, what actions it should perform in each given state. We use generated mappings 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.

Why mediation can fail without prior analysis: Consider the situation of the requester from Figure 2 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 [9]), these two options would appear as indistinguishable to it 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.

Mediator 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)

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. 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. (see Section 5)

4. Execution: for each requester's request until requester or provider finishes successfully or the execution fails do:

4.1 Retrieve possible actions from the mappings repository that are available in this context

4.2 Remove inconsistent actions: actions that are not consistent with actual variables bindings (e.g., preconditions fail)

4.3 If no mediation action that can be taken is available, fail

4.4 If 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[4]. If there is no such an action, we choose the one that allows to finish the most paths.

4.5 Execute selected action: depending on the type of an action either the *OWL-S Virtual Machine* [11] 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.

[2] 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.

[3] Note that the bactracking is possible because of *choise* and *any-order* control constructs that allow different execution paths to be chosen.

[4] Alternatively, if more possible actions were suggested, we could fork the execution and try to follow different possible mappings in parallel. This would be possible only for service calls that do not modify the state of the world (i.e., have no effects) and only while results returned to the requester do not differ. Currently we always choose only one action.

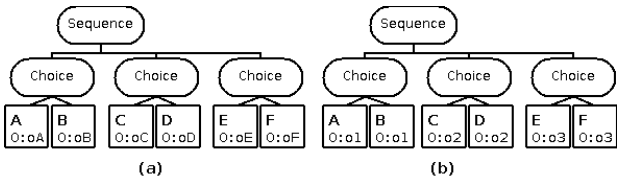

Figure 5. Analysis of choices

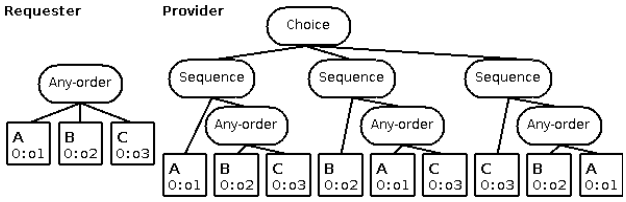

Figure 6. Analysis of any-order

4.6 Update state of the mediator: *mappings repository* and variables values and valid expressions are updated.

4 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. We show, however, that in some situations we do not need to explore every possible path.

Figure 5a depicts a sequence of three choice constructs. For simplicity we only show names of services and their output types. Consider for example that we want to figure out what actions can be performed to reconcile the step E. There are four different combinations of available variables depending on the path that leads to step E: (oA, oC), (oA, oD), (oB, oC), (oB, oD). This means that in this particular case all possible paths leading to E must be explored, because for every path different variables are available which can allow different mediation actions to be taken. On the other hand, consider the same situation in case of Figure 5b. No matter which path was taken, only one combination of available variables before executing step E is possible, namely (o1, o2). This simple situation can be generalized as follows:

1. Each service call (atomic process) must be included in at least one explored path.
2. In case of branching which is followed by other steps two situations are possible:
   (a) The state after finishing the branching does not depend on which branch was chosen, i.e., available variables and valid effects are always the same (as in case 5b) In this case we can choose any of the paths going through the branching with the same effect.
   (b) The state is different for different branches, therefore all possibilities need to be explored.

This observation allows us to reduce the amount of explored paths significantly in some cases. In the example 5a eight paths must be explored while in case 5b only two. Still, in the worst case, the number of explored paths can be exponentially high ($2^{n/2}$ paths for n steps) if we consider branching only.

Repeat-while can be treated similarly as branching — either the body of while is skipped, or it is executed. More then one execution of the *while loop* has no additional effect, since we perform only static evaluation. For *repeat-until* construct, we know that the body is always executed at least once and for the same reason as with *while*, we consider only situation when it is executed exactly once.

Any-order presents the main source of possible branching since its steps can be executed in any order. Thus one *any-order* construct with n steps can generate $n!$ possible orderings. However, if the process model is well formed, we can assume that all steps forming *any-order* construct are mutually independent (since they can be executed in any order they should be). This means that (1) no matter what ordering is chosen, the outcome after finishing *any-order* will be always the same, and (2) the available reconciliation actions for each step of the *any-order*, should depend only on steps performed prior to entering the *any-order* construct. Thus, in an ideal case we only need to choose one random ordering of steps. The situation can be complicated by the way in which the *any-order* is mapped into the provider's process model. The general idea is following:

1. We find all mappings for the path with one random ordering of steps of *any-order*
2. If one of the mappings maps steps of any-order to "equivalent" *any-order* construct in the provider's process model, we are done, because by this mapping all orderings can be covered.
3. If there is no such a mapping (i.e., no equivalent *any-order* can be found), other options must be considered. Specifically, the *any-order* can be represented as the partial or complete enumeration of all orderings in the provider's process model.

Figure 6 illustrates one of the situations for which the any-order can be successfully mapped into the provider's process model in which it is represented by the partial enumeration. In this case we need to explore three different paths,

starting with step A, or B or C instead of picking just one random ordering. In the worst case, in which the *any-order* is represented as an enumeration of all possible orderings in the provider's process model, we can end up with exploring $n!$ paths. However, in this situation the number of paths is limited by the number of steps in the provider's process model, which is linear. The algorithm for deciding which paths need to be explored is a recursive procedure which step-wisely explores partial enumerations of the *any-order*. It is bit tricky because different ways of how *any-order* can be represented in the provider's process model must be analyzed. However, it is more a matter of careful analysis than of an interesting new idea.

Split and *split-join* can be treated as *any-order*, because their semantics says that all their steps must be executed and the ordering is not specified.

5 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 simulaition backtracks to the last branching (e.g., *choice* or *any-order*). The mapping is constructed during the simulation and is represented as a sequence of actions that the mediation component should execute during the runtime mediation (see Fig. 7 for an example of a mapping). To make our pseudocode simpler we assume that each time some action is executed during the simulation, it is also added to the 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*

Since services are not executed during the simulation, we do not use values of inputs and outputs, but we only reason with variables names and their types. For example, the atom *(Available userId UserID)* is saying that a variable *userId* of type *UserID* is available and can be used as an input of some task. Preconditions and effects cannot be fully evaluated because of missing variables values. However, if the effect or precondition does not depend on variables values, it can be partially evaluated, as, e.g., in case of the effect *(loggedIn userId)* of the *LoginStep2* atomic process from the motivating example.

First, we provide the pseudocode for *executeNextRequesterCalls* method which is responsible for simulation of one or more requester's calls:

Method executeNextRequesterCalls
if $requesterStepsSequence == \emptyset$ **then**
  exit reconciliation procedure // *all steps reconciled sucessfully*
end if
repeat
  $rqstStep$ = deleteFirst($requesterStepsSequence$)
  $inputs$ = inputs of $rqstStep$; $outputs$ = outputs of $rqstStep$;
  $effects$ = effects of $rqstStep$
  for each $input$ **in** $inputs$ add(*(Available inputName inputType)*)
  for each $ouptut$ **in** $outputs$ add(*(RequesterGoal (Available outputName ouptutType))*)
  for each $effect$ **in** $effects$ add(*(RequesterGoal effect)*)
until ($outputs \neq \emptyset$) or ($requesterSteps == \emptyset$)

This method adds each input of the requester's call to the simulator's state and for each output and effect creates an appropriate goal. If the requester does not expect any output to be returned, subsequent step can be executed immediately.

Reconciliation method for atomic process P
if all inputs of P are available and precondions satisfied **then**
  simulate P (add outputs and effects of P to the simulator's state)
else
  $Goals$ = transform missing inputs & preconditions to goals
  // *e.g., missing toCode => (Goal (Available toCode AirportToCode))*
  if solve-goals($Goals$, true) == false **then**
    fail // *no way how to get inputs and preconditions*
  else
    simulate P (add outputs and effects of P to the simulator's state)
  end if
end if
if all required outputs are available and effects true **then**
  send outputs to requester
  executeNextRequesterCalls()
else
  $Goals$ = transform missing outputs & effects to goals
  //*(RequesterGoal goal) => (Goal goal)*
  if solve-goals($Goals$, false) == false **then**
    continue //*unsatisfied requester's goals can be resolved by subsequent provider's atomic processes*
  else
    executeNextRequesterCalls()
  end if
end if

This method adds all outputs and effects of the atomic process to the simulator's state if the inputs of the process are available and its preconditions are satisfied, and returns the outputs to the requester if they are available. If something is missing, appropriate goals are created and the reconciliation method *solve-goals* is called. The *solve-goals* method implements a backward chaining algorithm, which tries to supply missing variables and satisfy false expressions by means of applying translators and external services. If new goals are created by some of translators or external services, *solve-goals* is called recursively.

Method solve-goals($Goals$**,** $solveAll$**)** *//when solveAll is true, all goals must be resolved to succeed*
$returnStatus$ = true
for all goals g in $Goals$ **do**
  if solve-goal(g) == false **then**
    if $solveAll$ == true **then**
      return false
    else
      $returnStatus$ = false
    end if
  end if
end for
return $returnStatus$

Method solve-goal(g**),** g **of the form (Goal atom)**
for all methods m that can satisfy g **do**
  call method m *//methods m represent data mediators*
  if m fails **then**
    continue with next method
  end if
  NG = new goals added by executing m
  if $NG \neq \emptyset$ **and** solve-goals(NG, true) == true **then**
    return true
  end if
end for
return false

Data mediators can be seen as atomic processes: for given inputs, the outputs and effects are returned if preconditions are true. Therefore, we represent a data mediator similarly to atomic processes of the provider. However, since we are using data mediators in the backward chaining algorithm, if some of mediator's inputs are missing or preconditions are not satisfied, they are translated into new goals that need to be resolved. After the data mediator is called, its outputs and effects are added to the simulator's state. The other difference is that data mediators can be executed at any time when they can help to fulfill some goal, while provider's atomic processes are restricted by its process model.

Technical notes Let us mention some issues that we had to address during the simulation. Processes can have multiple conditional results. Because we cannot fully evaluate expressions, we treat all conditional results as possible deferring the precise evaluation to the mediator runtime, which is able to chose only the correct effect. Because of the same reason we treat *if-then-else* as *split* (i.e. both branches are possible) without evaluating the *if* condition. *Repeat-while* and *repeat-until* also do not evaluate terminating conditions and are treated as potentially infinite loops. To avoid infinite looping during the simulation, we introduced a hard limit on number of iterations. If this limit is reached, the simulation branch fails, simulation backtracks and different branches are tested. Data bindings in general are quite useful, because they specify the sources of input variables and thus eliminate the need to find a suitable source during the simulation. Currently, the support for bindings is limited only to *valueSource* binding type. IS-A relationships between types of inputs and outputs are evaluated by using the reasoner. This allows us to identify necessary data castings.

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 flghtCount
mediator-prepare-to-send flights
mediator-prepare-to-send flghtCount
mediator-send

Figure 7. Example solution for a requester's path

Example Figure 7 shows part of one mapping generated for a requester's execution path. 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 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 explicit down-casting, user will be asked, if it is allowed or not. In this case the all castings are fine. (see [15] for details on analyzing casting operations for ontology classes).

6 Related work

[19] provides a conceptual underpinning for automatic mediation. The work closest to ours is [9]. Mediation between two WSMO based process is performed strictly during the runtime. Besides structural transformations (e.g., change of message order) also data-mediators can be plugged into the mediation process. [1] 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 requeter's process model is not taken into

considerations. OWL-S broker [12] 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. [7] and [10] 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). [4] applies 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 [15], 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 [5] while [6] argues for published ontology mapping to facilitate automatic translations.

7 Conclusions and further work

In this paper we described an automatic mediation of two OWL-S process models by using provided data mediators in the form of built-in converters and external services used to provide missing information and translations. 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. Although we used OWL-S for describing process models, our approach can be used also with different process specification languages (e.g., BPEL4WS in combination with WSDL-S). We are aware of some most obvious problems that we want to address in future work. To allow real-life mediation, better support for data mediation must be provided. 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.

References

[1] C. Aberg, P. Lambrix, J. Takkinen, and N. Shahmehri. sButler: A Mediator between Organizations Workflows and the Semantic Web. *World Wide Web Conference workshop on Web Service Semantics: Towards Dynamic Business Integration*, 2005.

[2] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma. Web service semantics - wsdl-s, 2005. http://www.w3.org/Submission/WSDL-S/.

[3] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. Business Process Execution Language for Web Services, Version 1.1. *Specification, BEA Systems, IBM Corp., Microsoft Corp., SAP AG, Siebel Systems*, 2003.

[4] M. Brambilla, I. Celino, S. Ceri, D. Cerizza, E. D. Valle, and F. M. Facca. A software engineering approach to design and development of semantic web service applications. In I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 2006.

[5] M. Burstein, D. McDermott, D. R. Smith, and S. J. Westfold. Derivation of glue code for agent interoperation. *Autonomous Agents and Multi-Agent Systems*, V6(3):265–286, May 2003.

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

[7] L. Cabral, J. Domingue, S. Galizia, A. Gugliotta, V. Tanasescu, C. Pedrinaci, and B. Norton. *IRS-III: A Broker for Semantic Web Services Based Applications*. 2006.

[8] E. Christensen, F. Curbera, and G. M. S. Weerawarana. Web services description language, 2001.

[9] E. Cimpian and A. Mocan. Wsmx process mediation based on choreographies. In C. Bussler and A. Haller, editors, *Business Process Management Workshops*, volume 3812, pages 130–143, 2005.

[10] J. Domingue, S. Galizia, and L. Cabral. Choreography in irs-iii - coping with heterogeneous interaction patterns in web services. In *Proc. 4th Intl. Semantic Web Conference.*, 2005.

[11] M. Paolucci, A. Ankolekar, N. Srinivasan, and K. P. Sycara. The DAML-S virtual machine. In D. Fensel, K. P. Sycara, and J. Mylopoulos, editors, *International Semantic Web Conference*, volume 2870 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2003.

[12] M. Paolucci, J. Soudry, N. Srinivasan, and K. Sycara. A broker for owl-s web services. In *Cavedon, Maamar, Martin, Benatallah, (eds) Extending Web Services Technologies: the use of Multi-Agent Approaches*. Kluwer, 2005.

[13] M. Paolucci, N. Srinivasan, and K. Sycara. Expressing wsmo mediators in owl-s. In *International Semantic Web Conference*, 2004.

[14] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77 – 106, 2005.

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

[16] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan. Automated discovery, interaction and composition of semantic web services. *Journal of Web Semantics*, 1 (1):27–46, 2004.

[17] K. P. Sycara, M. Klusch, S. Widoff, and J. Lu. Dynamic service matchmaking among agents in open information environments. *SIGMOD Record*, 28(1):47–53, 1999.

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

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