Web Service Composition using Answer Set Programming

Albert Rainer

ec3, Vienna, Austria,
albert.rainer@ec3.at,
http://www.ec3.at

Abstract. The description of interactions among Web Services with regard to the exchange of messages, their composition, and the sequences in which they are transmitted and received is an important problem. While some efforts tackle this problem by defining standards for static processes others aim at composing complex services on-the-fly using AI-techniques. The present paper follows this AI approach and suggests a solution that uses Answer Set Programming (ASP) as a mean to achieve the composition goal. In particular, it shows how Web service descriptions and customer requests are mapped to the input language of the ASP software DLV[1] [1]. Furthermore, it shows how composition goals and constraints can be defined to guide the composition challenge. Finally, it presents a tool that serves as a framework to build complex services and that uses DLV as one of its back-end solvers. The approach was evaluated in the first Web service composition contest and has won the first prize[2].

1 Web Service Composition

In recent research related to Web services, several initiatives are aiming to provide means to allow easy integration of heterogeneous systems by defining languages, methods, or platforms. Some of these efforts (e.g. BPEL [2], OWL-S [3]) aim towards static processes with a priori knowledge of all possible execution paths, or by applying AI methods to compose complex services on-the-fly for an individual service requester.

A Web service is a collection of protocols and standards used for exchanging data between applications. Software applications written in various programming languages and running on various platforms can use web services to exchange data over computer networks like the Internet in a manner similar to inter-process communication on a single computer.

[1] DLV is an international cooperation between the University of Calabria and the Vienna University of Technology. More information about DLV available at http://www.dbai.tuwien.ac.at/proj/dlv/

[2] The first Web Service Composition Contest took place in Hongkong in March, 2005, http://www.comp.hkbu.edu.hk/~eee05/contest/

The Web Service Description Language WSDL [4] is a standard to describe Web services. It encompasses the definition of the messages and the message exchange protocol between a requester and provider agent as well as the definitions of operations and of bindings to a network protocol. The messages themselves are described abstractly and then bound to a concrete network protocol and message format. Web service definitions can be mapped to any implementation language, platform, object model, or messaging system. As long as both, sender and receiver, agree on the service description, (i.e. the WSDL document), the implementation behind the Web service can be any kind of software that is able to deliver the described service.

Coping with Web services (or services in general) leads to the fundamental tasks of discovery, i.e., to find the appropriate services for a certain customer request, of composition, i.e., to bundle single or complex services to complex products, and of invocation and monitoring, i.e, to enact and supervise (complex) services. Additionally, with respect to composition, at least three other tasks are of interest:

- to specify goals for a composite.
- to specify constraints that have to be respected.
- to provide means that help to analyze and to compare possible solutions against each other.

This work presents an approach that uses techniques from Answer Set Programming to solve some of the tasks required to build reliable and optimal compositions. In particular, we address the problem of finding and selecting an appropriate service, and the problem of limiting solution candidates to those that solve the goals and obey the restrictions imposed by the constraints. We demonstrate our work by using the challenges as well as the data format that has been specified for the Web Service Composition Contest [5].

The following WSDL document, which is taken from the composition contest, shows the data format and operation signature used for our work. It has a single operation with request/response protocol and simple data types - just a list of string properties.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions name="interopLab" xmlns="...>
<message name="findCloseHotel_Request">
  <part name="custStreetAddress" type="xsd:string"/>
  <part name="custCityAddress" type="xsd:string"/>
   other parts... </message>
<message name="findCloseHotel_Response">
  <part name="hotelName" type="xsd:string"/>
  <part name="hotelStreetAddress" type="xsd:string"/>
   other parts... </message>
<portType name="findCloseHotel">
  <operation name="findCloseHotel">
    <input message="tns:findCloseHotel_Request"/>
    <output message="tns:findCloseHotel_Response"/>
  </operation> </portType> </definitions>
```

Since XML is verbose and the meaning of messages and operations are clear and do not change within the context of our work we use the following simplified textual notation to describe a Web service.

```
(findCloseHotel //operation
 (custStreetAddress custCityAddress ...) //request message
 (hotelName hotelStreetAddress ...)) //response message
```

2 Web Service Composition as Planning Problem

Dynamically composing complex products (or services) is a research topic in the AI world since a long time. So, not surprisingly, a survey of composition approaches for Web services conducted by Rao et al. [6] revealed that typically AI methods from the planning domain are applied to the problem.

In general, a planning problem can be described as a five-tuple $\langle S, S_0, G, A, \Gamma \rangle$, where S is the set of all possible states of the world, $S_0 \subset S$ denotes the initial state of the world, $G \subset S$ denotes the goal state of the world, A is the set of actions the planner can perform, and the transition relation $\Gamma \subseteq S \times A \times S$ defines the precondition and effect for the execution of each action.

In terms of Web services, S_0 and G are the initial states and the goal states that are specified in the requirements of a service request. In the notation of the composition contest the initial state corresponds to a `provided` element and the goal state corresponds to the `resultant` elements of an XML document.

```
<Challenge><CompositionRoutine name="bookTrip">
<Provided>custStreetAddress,custCityAddress,... </Provided>
<Resultant>itineraryURL,...</Resultant>
</CompositionRoutine> ...</Challenge>
```

Again, a simplified notation helps to overcome the problem of verbose XML:

```
((problem bookTrip
 (provided custStreetAddress custCityAddress ...)
 (resultant itineraryURL ...)) ...)
```

2.1 Answer Set Programming with DLV

Answer set programming (ASP) is a form of declarative programming oriented towards difficult combinatorial search problems. It has been applied, for instance, to plan generation and product configuration problems in AI and to graph-theoretic problems arising in VLSI design. The original definition of answer sets for disjunctive logic programs was given by Gelfond and Lifschitz [7][8].

Syntactically, ASP programs look like Prolog programs, but the computational mechanisms used in ASP are different: they are based on the ideas that have led to the development of fast satisfiability solvers for propositional logic. A main difference between Prolog and ASP programs is that ASP programs are strictly declarative, while Prolog programs have a procedural aspect. In Prolog,

the order of rules as well as the order of subgoals in a rule matters while in ASP this makes no difference.

DLV (Datalog with Disjunction) is a powerful deductive database system. It is based on the declarative programming language datalog, which is known for being a convenient tool for knowledge representation. With its disjunctive extensions, it is well suited for all kinds of nonmonotonic reasoning, including diagnosis and planning.

Datalog is a declarative (programming) language. This means that the programmer does not write a program that solves some problem but instead specifies what the solution should look like, and a datalog inference engine (or Deductive Database System) tries to find the way to solve the problem and the solution itself. This is done with rules and facts. Facts are the input data, and rules can be used to derive more facts, and hopefully, the solution of the given problem.

Disjunctive datalog is an extension of datalog in which the logical OR expression (the disjunction) is allowed to appear in the rules - this is not allowed in basic datalog.

A disjunctive datalog program consists of an arbitrary number of facts, rules, and constraints. A rule takes the form

$head :- body$

which can be read as "if the body is true, the head must be true also". The head can contain one or more disjunctive literals while the body can contain zero or more conjunctive literals. A fact is a special form of a rule in which the body is empty and thus is always true. The other special form of a rule, in which the head is empty, is called a constraint. The body of a constraint must not become true, because the head can never become true.

2.2 The Guess/Check/Optimize Methodology

The core language of DLV can be used to encode problems in a declarative fashion following a Guess/Check/Optimize (GCO) paradigm.

Given a set F_I of facts that specify an instance I of some problem $\mathbf{P}$, a GCO program P for $\mathbf{P}$ consists of the following three main parts:

Guessing Part. The guessing part $G \subseteq P$ of the program defines the search space, such that answer sets of $G \cup F_I$ represent "solution candidates" for I.

Checking Part. The (optional) checking part $C \subseteq P$ of the program filters the solution candidates in such a way that the answer sets of $G \cup C \cup F_I$ represent the admissible solutions for the problem instance I.

Optimization Part. The (optional) optimization part $O \subseteq P$ of the program allows to express a quantitative cost evaluation of solutions by using weak constraints [9]. It implicitly defines an objective function $f : AS(G \cup C \cup F_I) \rightarrow \Re+$ mapping the answer sets of $G \cup C \cup F_I$ to (positive) real numbers. The semantics of $G \cup C \cup F_I \cup O$ optimizes f by filtering those answer sets having the minimum value.

In general, both G and C may be arbitrary collections of rules, and it depends on the complexity of the problem at hand which kind of rules are needed to realize these parts. In the next section some examples will illustrate the GCO paradigm.

3 Solution

A Web service repository contains a set A of services. Each service maps to a WSDL operation with input and output parameters that represent preconditions and effect of this service. From the composition request two artificial services are created, named *start* (the service that provides the initial data) and *end* (the service that requires the goal data). The repository and the two additional services are depicted in Figure 1*(a)*. The repository contains the services $a0 - a9$. The arrows denote disjunctive ordering constraints, for instance, service $a5$ can be scheduled after service $a2$ OR $a3$.

In order to reduce the search space simple filtering is applied. Only those services are selected for further processing which may play a role in a solution:

Forward reduction. In the first step, breadth first forward search beginning at the *start* service is applied to reduce the set of services by such services that will in no case be invoked since their preconditions are never satisfied (cf. Figure 1*(b)*).

Backward reduction. In the second step, using the (possibly) reduced set from step one, breadth first backward search starting from the *end* service is applied in order to remove services whose effects are never used. (cf. Figure 1*(c)*).

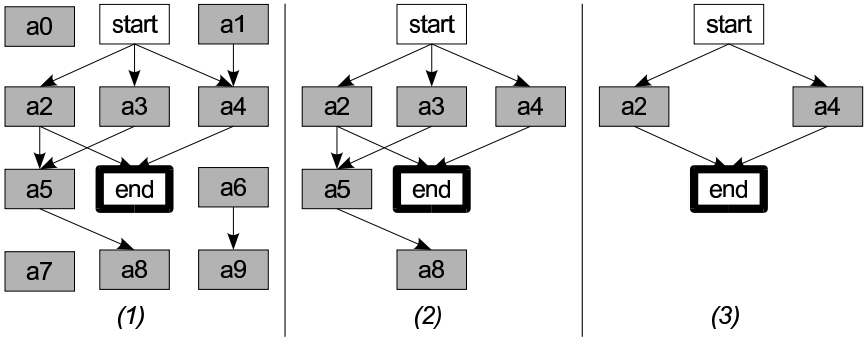

Fig. 1. Services $a0 - a9$ in repository, artificial services *start*, *end*. Edges denote disjunctive ordering constraints. Initial situation *(1)*, after forward reduction *(2)*, after backward reduction *(3)*.

3.1 Mapping Domain Model to DLV

The algorithm for mapping the services to the input of DLV: For each service a from the set of services A (the repository) select the set P of services from A that have as an effect one of the preconditions s required by a. For each of the services in P one disjunctive entry in the body of the rule will be created. In other words, each service is represented as a formula in conjunctive normal form (CNF).

$((a_0 \vee a_1 \vee .. \vee a_j)_1 \wedge (a_0 \vee a_1 \vee .. \vee a_j)_2 \wedge .. \wedge (a_0 \vee a_1 \vee .. \vee a_j)_i)$ with j being the services that are providers for the precondition i.
So for example, the requesting services in Figure 2 have the formulæ:

(1) $(a0 \wedge a0)$
(2) $((a0 \vee a1) \wedge (a2 \vee a3))$
(3) $((a0 \vee a1) \wedge (a1))$

Reduction for formulæ is as usual:
$(x \wedge x) \mapsto x$ and $(X \wedge Y) \mapsto X$ if $X \subseteq Y$.
In the example, the first formula becomes $(a0)$, the second remains unchanged, and the last is reduced to $(a1)$.

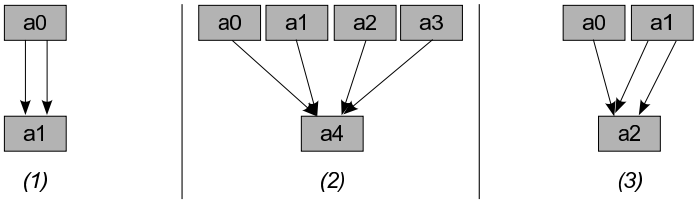

Fig. 2. Reduction of preconditions.

These formulæ are the input for the DLV solver. Each conjunction maps to one rule with the service name as body of the rule and the disjunctive part of the formula as the the head of the rule. For instance, Figure 2(2) yields two rules:

```
a0 v a1 :-a4.
a2 v a3 :-a4.
```

If the fact $a4$ is present in the database, DLV computes four models, each being a solution (a plan) for the given problem:

```
{a4, a1, a3}
{a4, a0, a3}
{a4, a1, a2}
{a4, a0, a2}
```

Constraints are means to reduce the result set either to answers that do not violate a (hard) constraint, or to limit the answer set to the cheapest answers. A constraint takes the form of a rule having no head. If the body becomes *true* the constraint is applied. This example shows the layered structure of a program that follows the GCO paradigm. The guessing part consists of disjunctive rules that guess a solution candidate, the checking part consists of integrity constraints that check the admissibility of the candidate, and the optimization part consists of weak constraints.

```
a4. %fact
% disjunctive rules -> guess
```

```
a0 v a1 :-a4.
a2 v a3 :-a4.
% hard constraint -> check
:-a2,a0.
% weak constraints -> optimize
:~a0.[1:1] :~a1.[1.3:1] :~a2.[1:1] :~a3.[1:1]
```

This example shows weak constraints denoting the costs to invoke a service. Service $a1$ has cost 1.3 while all other services have cost 1 associated. The second part of the cost array is used for the level the cost belongs to. The last line is a hard constraint that does not allow answers that have service $a0$ as well as service $a2$ in the result set. DLV comes up with a single answer in the result:

```
Best model: {a4, a0, a3}
Cost ([Weight:Level]): <[2:1]>
```

3.2 Plan construction for DLV input

The input for DLV is constructed by transforming the CNF representation for each service to rule(s). The literals in the head are the links from the providing service to the requesting service. Theses links express the ordering constraints between services. The body for each rule is the requesting service. The (artificial) *end* service denoting the goal is added as initial fact. This forms a graph with nodes $n(service)$ representing services and links $l(from - service, to - service)$ representing causal relationships. Searching is done backwards beginning at the goal, i.e. the *end* service.

Ordering constraints: Services can have cyclic dependencies between preconditions and effects, i.e. one service, e.g. service $a1$ may have as a predecessor service $a0$ which in turn has $a1$ as its predecessor. To avoid such answers (which are valid models but not valid plans) a constraint is applied that computes the path between nodes and becomes true if a node has been already visited.
A sample program:
Input: a problem and services $s_0 - s_6$ with preconditions and effects.

```
((problem p1(provided s)(resultant u v w)))
((s0(s)(a))
(s1(s)(b))
(s2(s)(c))
(s3(a)(u b))
(s4(b)(a c v))
(s5(c)(b w)))
```

Transformation of this problem for DLV yields the following GCO program:

```
n(end).
l(s5,end)  :- n(end).
```

```
l(s4,end)  :- n(end).
l(s3,end)  :- n(end).
l(s0,s3)  v  l(s4,s3)  :- n(s3).
l(s5,s4)  v  l(s1,s4)  v  l(s3,s4)  :- n(s4).
l(s2,s5)  v  l(s4,s5)  :- n(s5).
n(X):-l(X,_). %add node
r(X,Y) :-l(X,Y). %compute path
r(X,Z) :-r(X,Y),l(Y,Z).
:-r(X,Y),X==Y. %check admissibility
:~ n(s2).[1:1] :~ n(s0).[1:1] :~ n(s5).[1:1]
:~ n(s4).[1:1] :~ n(s1).[1:1] :~ n(s3).[1:1]
```

Description of the program:
The first part is the representation of services, with one fact $n(end)$, i.e., the goal.
The second part `n(X):-l(X,_).` adds a node for a selected link to the database, `X` is a variable and `_` is an anonymous variable.
The third part computes the path for any two nodes.
The fourth part is a hard constraint. This constraint prohibits answers that contain a path from one node to itself from being included in the result set.
The final part are the cost constraints, each service costs 1 unit.

The output for the domain and problem above is:

```
{l(s5,end), l(s4,end), l(s3,end), n(s5), n(end), n(s4),
n(s3), l(s0,s3), l(s3,s4), l(s4,s5), n(s0)}
Cost ([Weight:Level]): <[4:1]>
... other answers (the problem has 8 answers,
3 are optimal in terms of costs)
```

From the eight solutions generated for the problem above two plans are shown in the following table: the cheapest answer (left) is constructed from four services, ordered sequentially and concurrently (requires 3 time steps). The most expensive answer (right) with eight services ordered in three parallel sequences (requires two time steps).

```
(1)  s1          (1)  s2
(2)  s4          (1)  s0
(3)  s5          (1)  s1
(3)  s3          (2)  s5
                 (2)  s4
                 (2)  s3
```

3.3 Problem relaxation

Composition as described above results in totally ordered plans, i.e., every step is ordered with respect to every other step, and partially ordered plans, i.e., steps can be unordered with respect to each other.

Search space can become very large when many alternative services are available. So, for instance, when two services, `(s1(a)(b c))` and `(s2(b)(a d))`, and data `a b` are available and `c d` is a (sub)goal, then the answer will contain three compositions having the same costs:
`s1 -> s2`, `s2 -> s1` and `s1 && s2` with `->` and `&&` denoting sequence and concurrency, respectively.

The problem can be relaxed when search is done beginning at the start node, again using the GCO paradigm. The idea is that in the "guessing" part a service whose preconditions are satisfied is either selected (service is true) or not selected (service is false). When it is selected, the effects of the service are added to the facts which in turn may guess other services. The constraint is that an answer must contain the *end* node, i.e., the goal. Optimization is as usual with soft constraints.

```
%data available
a. b.
% Guess - either service is true or false
n(s2) v ~n(s2) :-b.
n(s1) v ~n(s1) :-a.
n(goal):-c,d.
% Check constraint - prunes illegal branches
:-not n(end).
% rules for adding effects of services
a :- n(s2). d :- n(s2).
b :- n(s1). c :- n(s1).
% Soft constraints, i.e. costs
:~n(s2).[1:1] :~n(s1).[1:1]
```

Running this program with DLV results in the single solution:
`{a, b, n(s2), n(s1), d, c, n(end)}`.
This solution has no ordering information, it contains just the selected services, the services that are not selected (none for this example) and the data elements added by selected services.

To gain an ordered solution we combine the two approaches: In a first step, guess and check is employed to find the cheapest solution candidates quickly. In the second step, starting backwards with the goal yields ordered plans from the unordered solution candidates as described above. Considering only services that are members of a solution found in the first step may reduce the search space extremely, but this may depend on the problem instance.

3.4 Web service Composition with MOVE

We have implemented the composition framework within the MOVE[3] environment using Java. The framework consists of a number of components with interfaces to discover Web services at different sources, to compose complex services

[3] MOVE is an acronym for Management and Optimization of Virtual Enterprises.

by employing different solvers, and to transform inputs and results to different data formats. Figure 3 illustrates the global architecture of the framework:

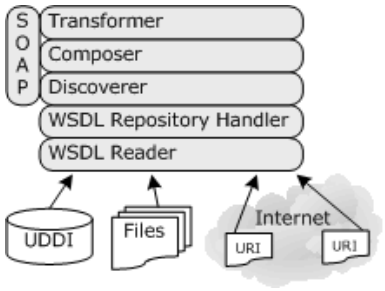

Fig. 3. Components

- The *WSDL reader* takes a WSDL document and converts it into a Java data structure. WSDL documents may stem from various sources: files in the local system, resources referenced by URIs, or resources located in a repository such as UDDI.
- The *WSDL repository handler* offers access to a set of WSDL data using Iterator and Caching.
- The *Discoverer* provides searching functionality to find services that match certain requirements.
- The *Composer* is responsible for building complex services. Apart from applying the DLV solver other planning and optimization software may be used.
- The *Transformer* component helps to convert input and output data to different data formats. Translating service composites to an executable scripting language such as XPDL [10] or creating DLV input files from WSDL documents are two examples where transformation is used.
- Finally, *SOAP* [11] is used to deploy and access the Web service composition framework itself as a Web service.

4 Summary and Outlook

This paper presents an approach to construct complex business processes on-the-fly using AI-techniques. It shows how service descriptions can be expressed in a rule based language that allows to search a repository efficiently and to build solutions that solve a goal with respect to soft and hard constraints.

In related work, typically methods from the planning domain are applied for this kind of problem. For instance, mapping WSDL documents to PDDL [12] which can subsequently be processed by a planner is described by Peer [13]. This is very similar to our approach. But a number of researchers see plain WSDL as

too limited to express preconditions and effects of services and they propose a richer syntax and semantic for service descriptions. As an example, composition of Web services annotated with semantic information expressed in OWL-S is addressed by Sirin and Parsia [14]. They show how a reasoner for description logics (DL) in combination with a planner can be applied to combine services. An important finding of their work is that using DL may cause performance problems.

In our contribution to a Web service composition contest [15] we have shown the usefulness of our approach. Our tool has performed best in finding solutions for a given set of problems. The tool is also used to analyze the randomly generated data for the next Composition Contest[4]. Finding all possible solutions helps to control the complexity of the challenges. Additionally, having all solutions computed ensures that solution claims from competitors can be verified.

We see as a strength of our approach that it provides means to gain all solutions for a given problem despite the cost of computation time and space. We imagine that at runtime a dedicated planning software employing fast heuristic algorithms is used to rapidly find a solution for a customer request. But there must be a possibility to analyze and verify results delivered by the planner.

The current solution is performing very well in this rather simple domain. We see a number of possible extensions for our framework to cope with "real world" requirements. First of all, the message format is very simple and may be enriched with complex data types or ontological information, for instance, OWL[16] expressions. We think that a logic reasoner like DLV fits very well the needs imposed by such a description logics. Secondly, services in a repository may have a control flow already defined and the challenge is to generate solutions that combine different processes. Again, we think that such requirements can be solved efficiently within our framework, but this needs more clarification. And thirdly, using a simple cost function such as plainly adding service costs may be not suitable to distinguish solution candidates and may be replaced by a complex utility function that represents better the requirements of a single service request.

Acknowledgements

This work is supported by MOVE, a research unit of ec3, the *Electronic Commerce Competence Center* in Vienna, Austria.
The author would like to thank Andreas Wombacher for providing services and problem instances as input data. This helped to verify the correctness of the results of the composition tool.
Special thanks go to Jürgen Dorn for his valuable comments and to Peter Hrastnik for implementing the WSDL reader, the repository handler, and the SOAP component of the tool.

[4] The second contest will take place in Beijing in October, 2005, http://www.comp.hkbu.edu.hk/~ctr/wschallenge/

References

1. Leone, N., Pfeifer, G., Faber, W., Calimeri, F., Dell'Armi, T., Eiter, T., Gottlob, G., Ianni, G., Ielpa, G., Koch, C., Perri, S., Polleres, A.: The dlv system. In: JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence, London, UK, Springer-Verlag (2002) 537–540
2. Andrews, T., et al.: Business Process Execution Language for Web Services. 2nd public draft release, Version 1.1. (2003)
3. Martin, D., et al.: OWL-S: Semantic Markup for Web Services. (2004) http://www.daml.org/services/owl-s/1.1.
4. Booth, D., Liu, C.K.: Web Services Description Language (WSDL) Version 2.0. (2005) http://www.w3.org/2002/ws/desc/.
5. Blake, M.B., Tsui, K.C., Wombacher, A.: The eee-05 challenge: A new web service discovery and composition competition. In: EEE '05: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05), Washington, DC, USA, IEEE Computer Society (2005) 780–781
6. Rao, J., Su, X.: A survey of automated web service composition methods. In: SWSWPC. (2004) 43–54
7. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R.A., Bowen, K., eds.: Proceedings of the Fifth International Conference on Logic Programming, Cambridge, Massachusetts, The MIT Press (1988) 1070–1080
8. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–386
9. Buccafurri, F., Leone, N., Rullo, P.: Enhancing disjunctive datalog by constraints. Knowledge and Data Engineering **12** (2000) 845–860
10. WfMC: XML Process Definition Language (XPDL) 1.09. (2005) http://www.wfmc.org.
11. W3C: Simple Object Access Protocol (SOAP) 1.2. (2003) http://www.w3c.org/TR/2003.
12. McDermott, D.: Pddl — the planning domain definition language (1998)
13. Peer, J.: A pddl based tool for automatic web service composition. In Ohlbach, H.J., Schaffert, S., eds.: PPSWR. Volume 3208 of Lecture Notes in Computer Science., Springer (2004) 149–163
14. Sirin, E., Parsia, B.: Planning for semantic web services. In: Semantic Web Services Workshop at 3rd International Semantic Web Conference (ISWC2004). (2004)
15. Dorn, J., Hrastnik, P., Rainer, A.: Web service discovery and composition with move. In: EEE '05: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05), Washington, DC, USA, IEEE Computer Society (2005) 791–792
16. Bechhofer, S., et al.: OWL Web Ontology Language Reference. (2004) http://www.w3.org/TR/owl-ref/.