A Logic based Approach for Service Discovery with Composition Support

Adina Sîrbu, Ioan Toma, and Dumitru Roman

Digital Enterprise Research Institute (DERI)
University of Innsbruck, Austria
<firstname>.<lastname>@deri.org

Abstract. Web service discovery given a user request becomes a fundamental challenge in a service-oriented world. The overall success of Service Oriented Architectures (SOA) however will very much depend on automatic and accurate solutions for the discovery problem. Furthermore such solutions need to be efficiently integrated with other service related tasks (e.g. service composition). In this paper we propose a logic based approach for service discovery with composition support. First, we provide a formal model for service discovery based on semantic description of services and then we show how such approach can be integrated with service composition. Furthermore we provide a prototype implementation that validates our theoretical solution.

1 Introduction

Service Oriented Architectures are emerging as a new computing paradigm for realizing distributed applications. They promote a service-based view on the world, where providers and clients are exposing and invoking functionalities in a standardized manner. Web services are one possible approach for implementing SOA ideas. They are based on technologies like WSDL [3], SOAP [14] and UDDI [2]. Despite their increasing acceptance in industry, Web services have some important drawbacks which stem mainly from the lack of machine understandable descriptions. More precisely, service related tasks like discovery, negotiation, adaptation, and composition cannot be performed by machines without the explicit intervention of a human programmer. Semantic Web services were proposed as a new paradigm that helps overcome current Web service technology limitations by providing semantically richer service descriptions. This enables machines to reason on these descriptions and to perform service related tasks in a more autonomous and accurate manner.

Service discovery, the task of finding relevant services given a user request, is one task where semantic based approaches can bring more automatization and accuracy. Solutions for discovery were proposed in [1], [12], [13], [9], [11]. However, most of the existing literature in this field refers to detecting matches by comparing the inputs and outputs of requested, respectively provided services. For example, the matching algorithms described in [11] and [9] depend only on the logical relation between the concepts associated with the inputs and outputs. Moreover, many of the proposed solutions are lacking a suitable integration with other service related tasks. Our solution is

focused exactly on these two aspects. We provide a formal model for service discovery based on semantic description of services and we show how such approach can be integrated with service composition. Furthermore we provide a prototype implementation that proves our ideas.

The paper is organized as follows: Section 2 provides the technical solution for our service discovery approach. It first gives some insights on the service model we are using for semantically describing Web services - the Web Service Modeling Ontology (WSMO) and its associated language - the Web service Modeling Language (WSML). The formal model for our discovery approach is described then in the rest of the section. Section 3 presents the prototype we have developed based on the technical solution provided in Section 2. The prototypical solution is described in terms of architecture and behavior. Furthermore a concrete run through a scenario is presented in order to exemplify the work of our prototype. Finally, Section 4 discusses the related work and Section 5 concludes the paper and presents our future work.

2 Discovery Approach

The conceptual model and the language we are using for semantically describing Web services is introduced in section 2.1. Based on this model, we present in section 2.2 two alternatives for Web service matchmaking, and for each of them the corresponding algorithm.

2.1 Modeling services

The discovery process in general, and service discovery in particular, depends heavily on how the entities that are to be discovered, in our case Web services, are modeled. For our logic based service discovery solution we adopt the Web Service Modeling Ontology (WSMO) [8] as conceptual model for services and its associated language Web Service Modeling Language (WSML) [4] as a language for semantically describing Web services. Some of the reasons behind this decision are: *(1)* WSMO is one of the major initiatives in Semantic Web services area. It provides a semantic-based solution for describing services which is crucial for a logic-based discovery approach such as ours, *(2)* WSMO provides a clean modeling solution for services, making a clear distinction between the user requests (*goals* in WSMO) and the services descriptions (*Web services* in WSMO), *(3)* WSML provides different semantic expressivity support for describing services. For our approach we consider one particular variant of the WSML languages family, namely WSML-Flight which offers a reasonable compromise between expressivity and decidability.

In a nutshell, WSMO provides an overall framework for Semantic Web services that aims at supporting automated Web service discovery, selection, composition, mediation, execution, monitoring, etc. It follows the design principles from the Web Service Modeling Framework (WSMF) [5] and provides four top-level notions related to Semantic Web services: *(1) Ontologies* that define a common agreed upon terminology used in the description of all others WSMO elements, *(2) Goals* which are descriptions

of the objectives a client may have when consulting a service in terms of functionality, behavior and quality of service, *(3) Web services* are descriptions of services and *(4) Mediators* which address heterogeneity problems that occur between descriptions at different levels: *data*, *protocol* or *process level*.

For our discovery approach, the first three top-level WSMO elements, namely ontologies, Web services and goals are considered. Although we don't use the mediation support, our solution can be easily extended to integrate mediation aspects. In the following, all Web services, goals and ontologies are specified using WSML. Furthermore, since our approach matches goals and Web services based on the functionality requested, respectively provided, we focus on describing the functional aspects of Web services and goals. Therefore, we leave aside the description of the interfaces, which by definition provide information on how the functionality of a Web service can be achieved. In WSML, the functional aspects of a Web service or a goal are grouped under an element called *capability*. A *capability* captures in terms of *preconditions* and *assumptions*, on one hand, and *postconditions* and *effects*, on the other hand, a set of conditions that have to hold before and respectively after the execution of the service. More precisely, the pre/postconditions refer to the information space of the Web service, while the assumptions and effects refer to the state of the world.

For exemplification purposes we introduce service and goal descriptions from the real-world use cases developed in the EU project Adaptive Service Grid (ASG)[1].

In this particular use-case, we consider a domain ontology that models a telematics domain. Listing 1.1 displays a fragment of this ontology, defining the top-level concepts of person, phone number, location, and a relation that holds between an entity and its location. Furthermore, this fragment includes an axiom stating that the location of a phone is also the location of a the owner of the phone.

```
ontology _"domainOntology.wsml"
  nonFunctionalProperties
    dc#title hasValue "Telematics domain ontology"
  endNonFunctionalProperties

  concept person
    name ofType _string
    number ofType phoneNumber

  concept phoneNumber

  concept location
    hasCoordinates ofType coordinates

  relation hasLocation/2
    nonFunctionalProperties
      dc#relation hasValue hasLocationDef
    endNonFunctionalProperties

  axiom hasLocationDef
    definedBy
      ?person[number hasValue ?phoneNr] memberOf person
      and ?loc memberOf location
      and hasLocation(?phoneNr, ?loc)
          implies hasLocation(?person, ?loc).
```

Listing 1.1. Domain ontology

[1] http://asg-platform.org

Based on this ontology, a telecommunication company offers a phone location service. This service requires as input the number of a mobile phone. This can be seen as a condition over the information space before the execution of the service and therefore is modeled as a *precondition*. The service invoker receives as result the location of the mobile phone. This can be seen as a condition over the information space after the execution of the service and therefore is modeled as a *postcondition*. The complete WSML description of the service is provided in the Listing 1.2.

```
webService _"MobTelPhoneLocationService"
  nonFunctionalProperties
    dc#title hasValue "MobTel phone location service"
    dc#publisher hasValue "MobTel"
  endNonFunctionalProperties
  importsOntology _"domainOntology.wsml"
  capability phoneLocationServiceCapability
    sharedVariables {?phoneNumber}
    precondition
        definedBy
          ?phoneNumber memberOf dO#phoneNumber.
    postcondition
        definedBy
          dO#hasLocation(?phoneNumber, ?location)
          and ?location memberOf dO#location.
```

Listing 1.2. MobTel phone location service

Further on, consider the generic goal of finding the location of a person, knowing the name and the phone number of this person. A goal in WSMO is described in a similar manner to a Web service. Listing 1.3 represents the formal description of the goal template. A concrete request can then be defined at runtime, by instantiating the goal template with concrete inputs.

```
goal _"findPersonLocation.wsml"
  nonFunctionalProperties
    dc#title hasValue "Find person location goal"
  endNonFunctionalProperties
  importsOntology _"domainOntology.wsml"
  capability findPersonLocationCapability
    sharedVariables {?person}
    precondition
        definedBy
          ?person[
             dO#name hasValue ?name,
             dO#number hasValue ?phoneNr
          ] memberOf dO#person.
    postcondition
        definedBy
          dO#hasLocation(?person, ?location)
          and ?location memberOf dO#location.
```

Listing 1.3. Find person location goal

The user can specify conditions on the information space that hold before the invocation of the matching service, in this case, that the name and phone number of the person are known. These aspects are modeled as *preconditions* in the goal. In the state of the world after the execution of a suitable service, the location of the person is known. Therefore we model this as a *postcondition* of the goal.

The discovery solution we are introducing in this paper will identify, using the background ontology, that the listed Web service represents an exact match for the goal.

More details on how services and goals descriptions are used by our solution are provided in Section 3.2.

2.2 Matching Web services and requests

We consider two alternatives for Web service matchmaking, each of them applying a different algorithm. They correspond to different phases in the Web service composition process.

The first matchmaking alternative is to locate the Web services that directly match a user request in a given state. If no Web services are discovered, the composer can construct a valid solution that fully satisfies the goal using the second alternative, which identifies all the Web services that are relevant to the request in the given state. More specifically, the service composer can construct a solution by successively discovering the executable services and virtually executing them until the state satisfies the goal. For a description of the service composer used in the context of ASG, we refer the reader to [10].

In section 2.1, we have presented our state-based approach to describing Web services and goals, which allows us to express Web services that can change the state of the world. This approach is characterized by the use of pre-state and post-state constraints for specifying the intended execution of the Web service. In WSMO, the pre-state constraints correspond to postconditions and assumptions, while the post-state constraints correspond to postconditions and effects. In this context, we have not made explicit distinction between effects and postconditions. Together, they represent the outcome of the service execution.

Both matchmaking algorithms take into account the dependence of outputs and effects of the service execution on the concrete input provided by the user. Therefore, they operate at the level of rich semantic description of services, as introduced in [6].

Matching based on capabilities The first algorithm for service matchmaking identifies the Web services whose capabilities fully match the requester goal.

Of the four possible types of match described in [6], we are taking into consideration only *exact-match* (the Web service description and the goal description coincide) and *plugin-match* (the sets of objects that the Web service claims to deliver is a superset of the set of objects that are relevant to the requestor). The other two cases (*subsumes-match* and *intersection-match*) are not considered valid matches in this context, because the services cannot fully satisfy the goal.

We consider the states of the world to be logical theories. A state of the world comprises the set of registered ontologies and, optionally, an additional set of facts. These facts can be given explicitly by means of an initial state. They can also be the outcome of previous virtual execution of services, because the execution of a service in a given state is considered to change the state of the world, resulting in an update to the logical theory.

In order to determine if the capability of a service satisfies a requester goal one must reason about the resulting updates. Reasoning about updates raises the frame problem. A solution to avoid the frame problem is offered by Transaction Logic, an

extension to First-order Logic that allows to specify the dynamics of knowledge bases in a declarative way. The theoretical approach employing Transaction Logic for Web service discovery that has been used as theoretical foundation for the implementation of this matchmaking algorithm can be found in [7].

The algorithm for service matchmaking based on capabilities implemented in our prototype is presented in Listing 1.4. The ontologies, the Web services and the goal are assumed to be loaded prior to the invocation of the matchmaking process.

```
1  algorithm Matchmaking based on capabilities
2  input: initial state I, goal G
3  output: map of <Web service S, set of <variable binding β>>
4
5   register state I
6  for each registered Web service S
7       if holds pre_S then
8            for each variable binding β
9                 if not holds (eff_S (β) and out_S(β)) then
10                     insert (eff_S(β) and out_S(β))
11                     if holds (eff_G and out_G) then
12                          add β to set of <β>
13                     endif
14                     delete (eff_S (β) and out_S (β))
15                endif
16           endfor
17           if not empty (set of <β>) then
18                add (S, set of <β>) to result map
19           endif
20      endif
21 endfor
22  unregister state I
23 return result map
```

Listing 1.4. Matchmaking algorithm based on capabilities

We consider a "stateless" functioning of the prototype, meaning that the relevant state information is given as input to each state-dependent operation. The state is loaded and respectively unloaded (lines 5, 22).

The available information sources at this point are:

- the set of ontologies referred by both goal and Web service descriptions
- the knowledge encoded in the state given as input to the matchmaking process
- the information that may be provided by the goal description itself

We select those registered Web services that are executable. In this context, a Web service $\mathcal{S}$ is considered executable if there exist input information in the available information sources such that the preconditions (what must be valid in order for the service to be executed) are fulfilled, while the effects and the postconditions (what the service guarantees after its execution) are not yet fulfilled. The assumptions describe conditions on information that is available only at run-time, and thus are not checked.

Therefore, $\mathcal{S}$ is executable if there exists at least one variable binding that satisfies the preconditions $\text{pre}_{\mathcal{S}}$, but not the effects $\text{eff}_{\mathcal{S}}$ and the postconditions $\text{out}_{\mathcal{S}}$ (lines 6-9). Checking that the effects and the postconditions of the Web service are not satisfied for the input that satisfies the preconditions is necessary due to the fact that in this context we wish to allow only a single execution of a Web service for a given input. Note however that a Web service can be executed an arbitrary number of times, with different input information.

A variable binding is a set of $< variable, value >$ pairs capturing the input information for which the service preconditions hold. More precisely, a variable binding is a complete set of bindings

$$< x_1, v_1 >, < x_2, v_2 >, ..., < x_n, v_n >$$

where $x_1, ..., x_n$ are the variables occurring in the precondition, and $v_1, ..., v_n$ is a set of constants. There can be several variable bindings for the same service, and all further tests on the service effects and postconditions will depend on the particular variable binding (line 9).

An executable service is considered a match if, for at least one of the variable bindings, the outcome of the service $\mathcal{S}$ satisfies the outcome requested in the goal $\mathcal{G}$. We perform this test by assuming the effects and the postconditions of the service for each variable binding and verifying if the effects and the postconditions of the goal hold in the resulting state (lines 10-14).

The set of matching services, and for each service all valid variable bindings, is then returned (line 23).

Matching for Web service composition The second matchmaking algorithm queries for the Web services that are relevant to composition. In this context, we consider a Web service to be relevant if it is executable in the given state.

```
1  algorithm Matchmaking on preconditions
2  input: initial state ℐ
3  output: map of <Web service 𝒮, set of <variable binding β>>
4
5   register state ℐ
6  for each registered service 𝒮
7      if holds pre_𝒮 then
8          for each variable binding β
9              if not holds ( eff_𝒮(β) and out_𝒮(β)) then
10                 add β to set of <β>
11             endif
12         endfor
13         if not empty ( set of <β>) then
14             add (𝒮, set of <β>) to result map
15         endif
16     endif
17 endfor
```

18 unregister state $\mathcal{I}$
19 **return** result map

Listing 1.5. Matchmaking algorithm for service composition

Listing 1.5 presents the algorithm. Similar to the previous algorithm, the ontologies and the services are assumed to be loaded in the reasoner prior to invocation of the matchmaking process. The state is loaded and respectively unloaded (lines 5, 18).

The available information sources for this second algorithm are:

- the set of the ontologies referred by the Web service descriptions
- the knowledge encoded in the state given as input to the matchmaking process

A Web service is considered a match in the context of this algorithm if it is executable. As already defined, a Web service $\mathcal{S}$ is executable if there exists input information such that the preconditions $\text{pre}_{\mathcal{S}}$ are fulfilled, while the effects $\text{eff}_{\mathcal{S}}$ and the postconditions $\text{out}_{\mathcal{S}}$ are not yet fulfilled (lines 6-9).

The set of executable Web services, and for each Web service all corresponding variable bindings, is then returned (line 19).

3 A Prototype System for Service Discovery

We have developed a prototype system that implements the matchmaking algorithms presented in Section 2.2. Furthermore we have tested and validated our prototype on a real-world scenario developed in the ASG project, called Attraction Booking Scenario. We now provide a high level overview of our system in terms of its architecture, components and interaction between them. The discovery process is afterwards exemplified with a run-through the previously mentioned scenario.

3.1 System overview

The high level architecture of our prototype system is provided in Figure 1. It consists of a set of loosely-coupled components which includes: a *System Interface*, a *Semantic Matchmaker*, a *Reasoner* and a *Repository*.

The system itself acts as a component having a defined *System Interface*. This interface offers a programmatic access to the system. Agents that act on behalf of service providers or service requestors can invoke functionalities exposed through this interface. The interface includes methods for managing semantic descriptions (e.g. register, unregister ontologies, services, goals), methods for querying the reasoner and methods for matching goals against registered services.

The *Semantic Matchmaker* is one of the core components of the Discovery System. It implements the matchmaking algorithms described in Section 2.2. It uses the reasoner to determine if the requested capability specified in a goal matches the capabilities of registered services.

The *Reasoner* provides querying and inference support required by the Semantic Matchmaker component. More precisely it supports a set of reasoning tasks like query

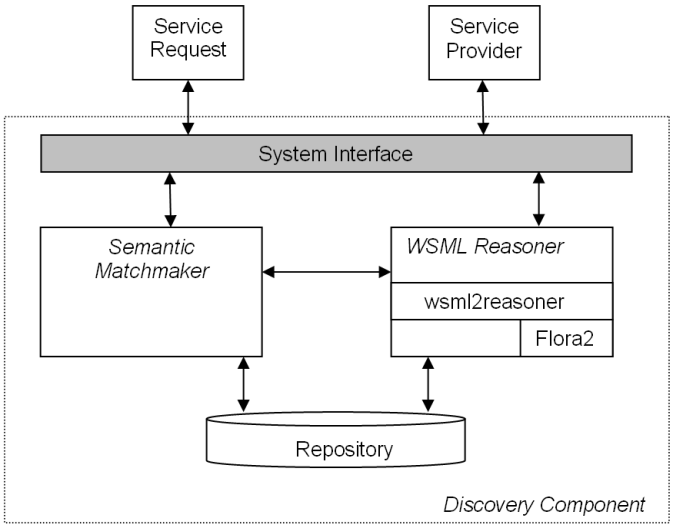

Fig. 1. Discovery system architecture.

answering with ontologies. As a backbone reasoner we have used the $\mathcal{F}lora-2$ system[2], integrated into the overall discovery system by using a generic framework called *wsml2reasoner*[3]. The framework allows easy integration of different reasoning engines for WSML language.

The *Repository* stores semantic descriptions like ontologies, goals and Web services. It provides methods to register and unregister the semantic descriptions mentioned before. Additionally, sets of facts that represent the states of the world at certain points in time can be registered or unregistered.

3.2 Application to use case scenario

The use case presented in the following paragraphs is a simplified fragment from the *Attraction Booking* scenario developed in ASG[4]. In this scenario taken from the telematics domain, a customer uses a mobile device, such as a handheld, to retrieve information on the attractions located in the nearby surroundings. Depending on the information received, the customer can additionally request for attraction details (e.g. the starting time of the event), for the description of a route leading to the attraction or, if the attraction is bookable, for a reservation to the event.

The domain ontology used in this scenario defines concepts, relations and instances associated to attractions (e.g. *attraction*, *attractionBag*, *attractionCategory*), locations (e.g. *city*, *street*, *coordinates*), mobile devices (e.g. *phoneNumber*). Listing 1.6 is an

[2] http://flora.sourceforge.net
[3] http://dev1.deri.at/wsml2reasoner/
[4] https://asg-platform.org/

additional fragment of the ontology introduced in Listing 1.1, that refers to attractions. Besides concepts and instances associated to attractions and the search for attractions, we introduce an axiom which specifies that all events are bookable.

```
concept attraction
  name ofType _string
  description ofType _string
  bookingPossible ofType _boolean
  priceRangeA ofType priceRange
  categories ofType (1 *) attractionCategory
  locationA ofType location

concept event subConceptOf attraction

concept attractionBag
  nonFunctionalProperties
    dc#description hasValue "a list of attractions"
  endNonFunctionalProperties
  members ofType (1 *) attraction

concept attractionQuery
  keyword ofType _string
  numberOfResults ofType _integer
  attractionCategories ofType (1 *) attractionCategory

concept attractionCategory

instance categoryCinema memberOf attractionCategory
instance categoryMusic memberOf attractionCategory
instance categoryEatAndDrink memberOf attractionCategory

axiom allEventsAreBookableDef
  definedBy
    ?attraction memberOf Event implies
      ?attraction[bookingPossible hasValue _true].
```

Listing 1.6. Fragment from the domain ontology

Further on, we introduce two Web services from the Attraction Booking service space, that provide information about attractions.

CinemaxXAttractionInformationService - modeled after CinemaxX.de, this service retrieves a set of cinema events using as search criteria the location and an attraction query. The service requires that the cinema category is explicitly specified in the attraction query.

```
webService _"CinemaxXAttractionInfoService.wsml"
  nfp
    dc#title hasValue "CinemaxX Attraction Information Service"
    dc#publisher hasValue "CinemaxX.de"
  endnfp
  importsOntology _"domainOntology.wsml"
  capability CinemaxXAttractionInfoCapability
    precondition
      definedBy
        ?location memberOf dO#location
        and ?query[dO#attractionCategories hasValue dO#cinema] memberOf dO#attractionQuery.
    postcondition
      definedBy
        ?bagOfEvents[dO#members hasValue ?event] memberOf dO#attractionBag
        and ?event memberOf dO#event.
```

Listing 1.7. CinemaxX Attraction Information Service

StarbucksAttractionInfoService - similar to the CinemaxX Web service, this service retrieves a set of bookable attractions if a location and an attraction query are given. The service requires that the attraction category list present in the query to contain the ”eat-and-drink” category.

```
webService _"StarbucksAttractionInfoService.wsml"
  nfp
    dc#title hasValue "Starbucks Attraction Information Service"
    dc#publisher hasValue "Starbucks.com"
  endnfp
  importsOntology _"domainOntology.wsml"
  capability StarbucksAttractionInfoCapability
    precondition
      definedBy
        ?location memberOf dO#location
        and ?query[dO#attractionCategories hasValue dO#eatAndDrink] memberOf
              dO#attractionQuery.
    postcondition
      definedBy
        ?attrBag[dO#members hasValue ?attraction] memberOf dO#attractionBag
        and ?attraction[dO#bookingPossible hasValue _true] memberOf dO#attraction.
```

Listing 1.8. Starbucks Attraction Information Service

Matching based on capabilities Consider a generic goal of finding attractions that can be booked, located in the nearby surroundings of the user. The user request we wish to model is equivalent to the following natural language specification: ”Given a query that specifies the categories of attractions, the problem is solved when the list of bookable attractions is known.” The formal specification of the request is given in Listing 1.9.

```
goal _"findBookableAttractionsGoal.wsml"
  nfp
    dc#title hasValue "Find Bookable Attractions Goal"
  endnfp
  importsOntology _"domainOntology.wsml"
  capability findBookableAttractionsCapability
    precondition
      definedBy
        ?query[dO#attractionCategories hasValue ?category] memberOf dO#attractionQuery.
    postcondition
      definedBy
        ?bookableAttrBag[dO#members hasValue ?bookableAttr] memberOf dO#attractionBag
        and ?bookableAttr[dO#bookingPossible hasValue _true] memberOf dO#attraction.
```

Listing 1.9. Find bookable attractions goal

The user input is captured in the initial state of the problem (Listing 1.10), which defines a person, a location and an attraction query specifying a list of categories.

```
ontology _"initialState.wsml"
  importsOntology _"domainOntology.wsml"

  instance me memberOf dO#person
  instance myLocation memberOf dO#location
  relationInstance dO#hasLocation(me, myLocation)

  instance myQuery memberOf dO#attractionQuery
    dO#attractionCategories hasValue {dO#cinema, dO#music}
```

Listing 1.10. Initial state

The discovery process starts by checking if the goal holds in the initial state. Since the goal postcondition is not satisfied for the initial state, the next phase is service matchmaking based on capabilities, according to the algorithm presented in 2.2.

The algorithm analyzes every registered Web service. The preconditions in the CinemaxX Web service are fulfilled, and using the background ontology we determine that the outcome advertised in the Web service satisfies the outcome requested in the goal. The CinemaxX attraction information service is thus considered a match. On the other hand, even though it advertises only attractions that can be booked (and thus meets the goal postcondition), the Starbucks Web service is not a valid match, because its preconditions are not satisfied.

The algorithm returns the identifier of the matching Web service, together with the corresponding variable binding.

```
Service:
        CinemaxXAttractionInfoService.wsml
Service Variables Binding:
        location = initialState#myLocation
        query = initialState#myQuery
```

Listing 1.11. Find bookable attractions matching service

Matching for Web service composition Further on, we present a run-through that uses for matchmaking the algorithm defined in 2.2.

In order to simulate the Web service composition, we add to the service repository the phone location service introduced in section 2.1.

We consider the same generic goal of finding attractions that can be booked. For this second example, the initial state specifies the user, the phone number and the attraction query. However, in this initial state, no information related to the location of the user is known.

```
ontology _"altInitialState.wsml"
  importsOntology _"domainOntology.wsml"

  instance myNumber memberOf dO#phoneNumber
  instance me memberOf dO#person
  dO#number hasValue myNumber

  instance myQuery memberOf dO#attractionQuery
    dO#attractionCategories hasValue {dO#cinema, dO#music}
```

Listing 1.12. Alternative initial state

The simulation of the service composition process consists of one or more iterations through a series of steps. The steps are executed in the following order:

1. check if the goal holds in the current state. If true exit, else go to 2;
2. query for executable services. If no service is discovered exit, else go to 3;
3. virtually execute one of the discovered services.

First iteration: Testing whether the goal is reached in the initial state returns false. We proceed to the next step, finding executable services. The result contains only the phone location service, as it is the only Web service whose preconditions are satisfied (Listing 1.13).

In case more executable Web services are found, the composition planner can employ a complex approach for selecting the best matching Web service, while also taking into consideration non-functional properties like optimization criteria (e.g. price or speed) and static restrictions (e.g. only services from provider X).

```
Service:
   MobTelPhoneLocationService.wsml
Service Variables Binding:
   phoneNumber = altInitialState#myNumber
```

Listing 1.13. Executable service in the initial state

We start constructing the first alternative with the virtual execution of the phone location service. By assuming the outcome of this service, a dummy instance of type location is created and related to the phone number. Listing 1.14 gives the equivalent WSML description of the inserted facts.

```
domainOntology#location1 memberOf domainOntology#location.
domainOntology#hasLocation(altInitialState#myNumber, domainOntology#location1).
```

Listing 1.14. Virtual execution of the phone location service

Second iteration: The test whether we have reached the goal returns false.

The result of querying for the executable services in the current virtual state is the CinemaxX Web service, as it is the only service whose preconditions are satisfied.

```
Service:
        CinemaxXAttractionInfoService.wsml
Service Variables Binding:
        location = domainOntology#location1
        query = initialState#myQuery
```

Listing 1.15. Executable service in the second state

The virtual execution of the CinemaxX service adds a new dummy instance of the attraction bag concept, containing one dummy instance of the event concept. Listing 1.16 displays the WSML description of the added facts.

```
domainOntology#event1 memberOf domainOntology#event.
domainOntology#attractionBag1[
  domainOntology#members hasValue domainOntology#event1
    ] memberOf domainOntology#attractionBag.
```

Listing 1.16. Virtual execution of the CinemaxX attraction information service

Third iteration: Testing whether the goal was reached yields true.

```
Goal Variables Binding:
        bookableAttrBag = domainOntology#attractionBag1
        bookableAttr = domainOntology#event1
```

Listing 1.17. Goal variables binding

The output of the presented run-through is a possible service execution plan that can be constructed by a service composition planner. In this execution plan Mobtel phone location service and CinemaxX attraction information service are composed in order to achieve the user goal. Alternative service execution plans can be achieved in case more executable services are discovered at each step.

4 Related Work

The automatic discovery of services is nowadays a very popular research topic. Many solutions have been proposed ranging from pure syntactic to highly logic based approaches. However many of them lack a clear discovery model and a formal specification of the discovery process. Furthermore many of them cannot be easily integrated with solutions for other service related tasks.

Approaches like [9], [11], although logic-based, are missing a clear discovery model. These approaches are also too general and is not clear how they can support other service related tasks like service composition.

Same holds for other approaches (e.g. [13], [1])that were mainly provided to work in distributed environments like P2P. Besides the lack of clear discovery model and support for other service related tasks, many of these approaches lack as well a formal, concise algorithms for service discovery.

5 Conclusions and Future Work

In this paper we presented a logic based approach for service discovery that can be easily used by service composition modules. Two kinds of algorithms for service discovery were presented, one based on capability matching, the other supporting service composition. Furthermore we have implemented a proof of concept prototype that validates our solution on real use-case scenarios. As future work we plan to compare our solution and implementation against other service discovery solutions. Also, we plan to refine our solution to include the possibility of ranking the matching Web services, the problem of ranking being one of the main challenges in Web service discovery. Performance and scalability tests are also left as future work.

6 Acknowledgements

The work is funded by the European Commission under the projects ASG, DIP, enIRaF, InfraWebs, Knowledge Web, Musing, Salero, SEKT, Seemp, SemanticGOV, Super, SWING and TripCom; by Science Foundation Ireland under the DERI-Lion Grant No.SFI/02/CE1/I13 ; by the FIT-IT (Forschung, Innovation, Technologie - Informationstechnologie) under the projects Grisino, RW^2, SemNetMan, SeNSE and TSC.

References

1. Rama Akkiraju, Richard Goodwin, Prashant Doshi, and Sascha Roeder. A method for semantically enhancing the service discovery capabilities of UDDI. In Subbarao Kambhampati and Craig A. Knoblock, editors, *Proceedings of the IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03)*, pages 87–92, 2003.
2. T. Bellwood, L. Clment, D. Ehnebuske, A. Hately, Maryann Hondo, Y.L. Husband, K. Januszewski, S. Lee, B. McKee, J. Munter, and C. von Riegen. Uddi version 3.0. http://uddi.org/pubs/uddi-v3.00-published-20020719.htm, July 2002.

3. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl, March 2001.
4. Jos de Bruijn, Holger Lausen, Reto Krummenacher, Axel Polleres, Livia Predoiu, Michael Kifer, and Dieter Fensel. The Web Service Modeling Language WSML. Technical report, WSML, 2005. WSML Final Draft D16.1v0.21. http://www.wsmo.org/TR/d16/d16.1/v0.21/.
5. Dieter Fensel and Christoph Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.
6. Uwe Keller, Ruben Lara, Axel Polleres, Ioan Toma, Michael Kiffer, and Dieter Fensel. WSMO discovery. Working Draft D5.1v0.1, WSMO, 2004. Available from `http://www.wsmo.org/TR/d5/d5.1/v0.1/`.
7. Michael Kifer, Rubén Lara, Axel Polleres, Chang Zhao, Uwe Keller, Holger Lausen, and Dieter Fensel. A logical framework for web service discovery. In *ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, volume 119, Hiroshima, Japan, 2004. CEUR Workshop Proceedings.
8. H. Lausen, A. Polleres, and D. Roman (eds.). Web Service Modeling Ontology (WSMO). W3C Member Submission 3 June 2005, 2005. online: http://www.w3.org/Submission/WSMO/.
9. Lei Li and Ian Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, Budapest, Hungary, May 2003.
10. Harald Meyer and Mathias Weske. Automated service composition using heuristic search. In *Proceedings of the Fourth International Conference on Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, Vienna, Austria, 2006.
11. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In I. Horrocks and J. Handler, editors, *1st Int. Semantic Web Conference (ISWC)*, pages 333–347. Springer Verlag, 2002.
12. K. Sycara, S. Widoff, M. Klusch, and J. Lu. Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. *Autonomous Agents and Multi-Agent Systems*, pages 173–203, 2002.
13. K. Verma, K. Sivashanmugam, A. Sheth, and A. Patil. Meteor-s wsdi: A scalable p2p infrastructure of registries for semantic publication and discovery of web services. *Journal of Information Technology and Management*, 2004.
14. W3C. SOAP Version 1.2 Part 0: Primer, June 2003.