Service Discovery with SWSL-Rules

Michael Kifer
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
kifer@cs.stonybrook.edu

Position paper for the W3C Workshop on Frameworks for Semantics in Web Services, June 2005

Abstract

This position paper presents a use case for service discovery using SWSL-Rules, a logic language for specifying semantic Web services. SWSL-Rules plays the role of both a specification language for services and user goals, and of an implementation language for the mediators and the discovery engine itself. The example illustrates the use of a number of advanced features of the language such as frame-based syntax, reification, Lloyd-Topor extensions, and logical theory updates based on Transaction Logic.

1 Introduction

This example illustrates a use of SWSL-Rules for Web service discovery. SWSL-Rules plays the role of both a specification language for services and user goals, and of an implementation language for the mediators and the discovery engine itself. The particular features of the language that this example relies on include frame-based representation, reification, and nonmonotonic Lloyd-Topor extensions. In addition, logical updates of Transaction logic (*http://flora.sourceforge.net/aboutTR.php*) are used in the discovery engine.

To make the example manageable, services are described only by their names and conditional effects. To discover a service, users must represent their goals in one of the two simple ontologies as described below. These goals are described in terms of *user requests*, which are descriptions of what the user wants to be true in the after-state of the service (i.e., the state that would result after the execution of the service).

User goals and service descriptions may be expressed in different ontologies and so *mediators* are needed to translate between those ontologies. In this example, we assume that each service advertises the mediators that can be used to talk to the service though the attribute `mediators`.

A fundamental belief underlying this framework is that most of the users of a Web service discovery infrastructure lack any training in logic and, therefore, the discovery goals that they can pose should be simple and expressed in a very high-level language whose statements will be generated by a GUI. Service providers, on the other hand, are businesses and they will usually be able to hire a limited number of knowledge engineers to help create semantic descriptions of their services. Alternatively, they may be able to buy software that would enable creation of such descriptions. However, we do not believe that there will be sufficient number of engineers (or that the software will be sophisticated enough) who will be sufficiently skilled in logic. Therefore, semantic descriptions of Web services are expected to be relatively simple. In our view, the bulk of intelligence will be concentrated in the mediators, which will be produces by a relatively small number of firms who will be able to tap into the small market of highly skilled knowledge engineers. The use case presented below illustrates this philosophy: very simple user goals, relatively simple service descriptions, and much more subtle logic is reserved for mediators.

2 Ontologies

Geographical ontology. To begin, we assume the following simple geographic taxonomy, which is shared by all users and services. It defines several regions and subregions, such as `America`, `USA`, `Europe`, `Tyrol`. Each region is viewed as a class of cities. For instance, `Innsbruck` is a city in `Tyrol` and `StonyBrook` is a town in the New York State (`NYState`). In the following, the symbol ":" is used to denote class membership; "::" denotes the subclass relationship.

```
USA::America.                    Innsbruck:Tyrol.
Germany::Europe.                 Lienz:Tyrol.
Austria::Europe.                 Vienna:Austria.
France::Europe.                  Bonn:Germany.
Tyrol::Austria.                  Frankfurt:Germany.
NewYorkState::USA.               Paris:France.
StonyBrook:NewYorkState.         Nancy:France.
NewYork:NewYorkState.
// regions are defined recursively
Europe:Region.                   ?Reg:Region :- ?Reg1:Region and ?Reg::?Reg1.
America:Region.                  ?Loc:Location :- ?Reg:Region and ?Loc:?Reg.
```

To make it easier to specify what is a region and what is not, we use a rule (the penultimate statement above) to say that a subclasses of a region are also regions and, therefore, such subclasses do not need to be explicitly declared as regions. The last rule simply says that any object that is a member of a geographical region is a location.

Goal ontology. Services write their descriptions to conform to specific ontologies. Likewise, clients describe their goals in terms of goal ontologies. Here we will not describe these ontologies, but rather the forms of the inputs and outputs that the services expect and produce and the structure of the user goals. Furthermore, since users and service designers

are unlikely to be skilled knowledge engineers, we assume that the inputs, the outputs, and the goals are fairly simple and that most of the intelligence lies in the mediators.

We assume that there is one ontology for goals and two for services. Consequently, there are two mediators: one translating between the goal ontology and the first service ontology, and the other between the goal ontology and the second service ontology. The goal ontology looks as follows:

```
Goal[requestId *=> Request,
     request   *=> TravelSearchQuery,
     result    *=> Service ].
```

In SWSL-Rules, the symbol `*=>` is used to specify the type of an inheritable attribute. For instance, `requestId *=> Request` means that the value of the attribute `requestId` on any member of the class `Goal` must be an object that belongs to class `Request`.

The classes `Request` and `Service` will be specified explicitly by placing specific object Ids in them. The class `TravelSearchQuery` consists of the following search queries:

```
searchTrip(?F,?To):TravelSearchQuery :- ?F:(Region or Location) and ?To:(Region or Location).
searchCitipass(?Loc):TravelSearchQuery :- ?Loc:(Region or Location).
```

The meaning of the query `search(X,Y)` depends on whether the parameters are regions or just locations. For location-parameters, the query is assumed to fetch the services that serve those locations. For region-parameters, the query is assumed to find services that service *every* location in the region that is known to the knowledge base. For instance, searchTrip(Paris, `Germany)` is a request for travel services that can sell a ticket from Paris to *any* city in Germany. Similarly, `searchCitipass('New York')` is interpreted as a search for travel services that can sell city passes for New York and the request `searchCitipass(USA)` is looking for services that can sell city passes for every location in USA. The `result` attribute is provided by the ontology as a place where the discovery mechanism is supposed to put the results.

Domain-specific service ontologies. A service ontology is intended to represent the inputs and outputs of the service as well as the effects of the service. Since the inputs are not generally provided in the user goal (since the user is not expected to know anything about such inputs), the job of translating goal queries into the inputs to the services lies with the mediator. Service ontology #1 is defined as follows:

```
// Service input
search(?requestId,?fromLocation,?toLocation):ProcessInput :-
                ?requestId:Request and ?fromLocation:Location and ?toLocation:Location.
search(?requestId,?city):ProcessInput  :- ?requestId:Request and ?city:Location.
// Service output
ItineraryInfo::ServiceOutput.
PassInfo::ServiceOutput.
ItineraryInfo[from*=>Location, to*=>Location].
PassInfo[city*=>Location].
itinerary(?reqNumber):ItineraryInfo :- ?reqNumber:Request.
pass(?reqNumber):PassInfo :- ?reqNumber:Request.
```

Note that services expect locations as part of their input and they know nothing about regions. In contrast, as we have seen, user goals can have region-wide requests. It is one of the responsibilities of the mediators to bridge this mismatch.

Service ontology #2 is similar to ontology #1 except that it understands only requests for citipasses and the formats for the input and the output are slightly different.

```
// Service input
discover(?requestId,?city):ProcessInput  :- ?requestId:Request and ?city:Location.
// Service output
ServiceOutput[location*=>Location].
?reqNumber:ServiceOutput :- ?reqNumber:Request.
```

Shared core ontology for services. In addition, we need a core ontology that is shared by everyone in order to provide a common ground for the service infrastructure. In this example, the core ontology is represented by a single class `Service`, which is declared as follows:

```
prefix xsd = "http://www.w3.org/2001/XMLSchema".
Service[ name                *=> xsd#string,
         effect(ProcessInput) *=> Formula,
         mediators            *=> Mediator ].
```

Note that the definition of the class `Service` belongs to the core ontology and therefore it is shared by everybody. The method `effect` represents the conditional effect of the service. It takes an input to the service as a parameter and returns a set of rules that specify the effects of the service for that input. `Formula` is a predefined class. The attribute `mediators` indicates the mediators that the service advertises for anywho would want to talk to that service.

Note that the class `ProcessInput` belongs to the core ontology, but it's extension is defined the the domain-specific ontologies.

3 Examples of Concrete Services

We now present instances of concrete services. In these instances, the symbol `->` represents the values of attributes (as opposed to the types of the attributes, which are represented using the symbol `*=>`). For instance, the statement `serv1:Service[ name -> "Schwartz Travel, Inc.", ...]` below says that `serv1` is an object of class `Service`, that `name` is an attribute of `serv1`, and that its value is `"Schwartz Travel, Inc."`.

```
// This service uses ontology #1, and mediator med1 bridges it to the goal ontology
serv1:Service[
    name -> "Schwartz Travel, Inc.",
     // Input must be a request for a ticket from somewhere in Germany to somewhere
     // in Austria  OR  a request for a city pass for a city in Tyrol
     // The output object is either an itinerary object with Id
     // itinerary(requestId) or a citipass object with Id pass(requestId).
    effect(?Input) -> $ (itinerary(?Req)[from->?From,to->?To] :-
                           Input = search(?Req, ?From:Germany, ?To:Austria)) and
                           (pass(?Req)[city->?City] :- ?Input=search(?Req,?City:Tyrol)) ,
    mediators -> med1 ].

serv2:Service[ // Another ontology #1 service
    name -> "Mueller Travel, Inc.",
    effect(?Input)-> $ itinerary(?Req)[from->?From, to->?To] :-
                              ?Input = search(?Req,?From:(France or Germany),?To:Austria) ,
    mediators -> med1 ].

serv3:Service[ // An ontology #2 service
    name -> "France Citeseeing, Inc.",
    effect(?Input)-> $ ?Req[location->?City] :- ?Input=discover(?Req,?City:France) ,
    mediators -> med2 ].

serv4:Service[ // Another ontology #2 service
    name -> "Province Travel",
    effect(?Input)->
          $?Req[location->?City] :- ?Input = discover(?Req,?City:France) and ?City != Paris,
    mediators -> med2 ].
```

4 User Goals

Next we show examples of user goals. Note that the value of the attribute `result` is initially the empty set. When the goal is posed to the discovery engine, this value will be changed to contain the result of the discovery.

```
goal1:Goal[ requestId -> _#:Request,
            request   -> searchTrip(Bonn,Innsbruck),
            result    -> {} ].
goal2:Goal[ // search for services that serve all cities in France and Austria
   requestId -> _#:Request,
   request   -> searchTrip(France,Austria),
   result    -> {} ].
goal3:Goal[ requestId -> _#:Request,
            request   -> searchCitipass(Innsbruck),
            result    -> {} ].
goal4:Goal[ // services that can sell citipasses for every city in France
   requestId -> _#:Request,
   request   -> searchCitipass(France),
   result    -> {} ].
```

5 Mediators

Each of the two mediators, `med1` and `med2`, consists of several main clauses. The first clause in each mediator takes a user goal and translates it into input that is appropriate for the corresponding domain-specific service ontology.

The remaining clauses define the mediator's method `getResult`. This method is invoked in the after-state of the service execution. It takes as parameters the user goal and the service (in whose after-state the method is invoked). Depending on the form of the goal's request, `getResult` poses a query that is appropriate for that request and the service ontology of the service. For instance, if the request is `searchCitipass(?City:Location)`, i.e., finding services that can sell citipasses for a specific location, then the query appropriate for services that use ontology #1 is `pass(?)`[`city->?City`] and the query for ontology #2 is `?`[`location->?City`]. Finally, if the query yields results, the mediator constructs output that can be used to return results to the user and that is compliant with our goal ontology.

Each form of the input has two cases: one where the parameters are locations (e.g., `searchCitipass(?C:Location)`) and the other where they are regions (e.g., `searchCitipass(?C:Region)`). Therefore, for each form of the input our mediators have two clauses. Since ontology #2 understands only one input, `med2` uses only two clauses to define `getResult`. The mediator for ontology #1, `med1`, needs four clauses to cover both forms of the input.

Finally, we remark that the clauses that deal with region-based requests have to construct more sophisticated queries to be asked in the after-state of the services. In our example, we use Lloyd-Topor extensions (which allow the `forall` quantifier in the rule body) to simplify such queries.

```
med1:Mediator.  // mediator for ontology #1
med1[constructInput(?Goal)->?Input] :-
        ?Goal[requestId->?ReqId, request->?Query] and
        (?Query = searchTrip(?From,?To) ==> ?Input = search(?ReqId,?From1,?To1))
        and (?Query = searchCitipass(?City) ==> ?Input = search(?ReqId,?City1)).

med1[getResult(?Goal,?Serv) -> $?Goal[result->?Serv]] :-
        ?Goal[request->searchCitipass(?City:Location)] and pass(?)[city->?City].
med1[getResult(?Goal,?Serv) -> $?Goal[result->?Serv]] :-
        ?Goal[request->searchCitipass(?Region:Region)] and
        forall ?City (?City:?Region ==> pass(?)[city->?City]). // Lloyd-Topor

med1[getResult(?Goal,?Serv) -> $?Goal[result->?Serv]] :-
        ?Goal[request->searchTrip(?From:Location,?To:Location)] and
        itinerary(?)[from->?From, to->?To].
```

```
med1[getResult(?Goal,?Serv) -> and ?Result = $?Goal[result->?Serv]] :-
        ?Goal[request->searchTrip(?From:Region,?To:Region)] and
        forall ?From,?To (?City1:?FromReg and ?City2:?ToReg    // Lloyd-Topor
                          ==> itinerary(?)[from->?City1, to->?City2]).

med2:Mediator.  // mediator for ontology #2
med2[constructInput(?Goal)->?Input] :-
        ?Goal[requestId->?ReqId, request->?Query] and
        ?Query = searchCitipass(?City) ==> ?Input = discover(?ReqId,?City1).

med2[getResult(?Goal,?Serv) -> $?Goal[result->?Serv]] :-
        ?Goal[request->searchCitipass(?City:Location)] and ?[location->?City].
med2[getResult(?Goal,?Serv) -> $?Goal[result->?Serv]] :-
        ?Goal[request->searchCitipass(?Region:Region)] and
        forall ?City (?City:?Region ==> ?[location->?City]).
```

6 The Discovery Engine

The final piece of the puzzle is the actual engine that performs service discovery. It relies on the features, borrowed from Transaction Logic (*http://flora.sourceforge.net/aboutTR.php*) — a possible extension of SWSL-Rules. These features include modifications to the current state of the knowledge base and *hypothetical execution* of such modifications.

```
findService(?Goal) :-
        ?Serv[mediators -> ?Mediator, effect(?Input) -> ?Effects] and
        ?Mediator[constructInput(?Goal) -> ?Input] and
        hypothetically(insert{?Effects} and ?Mediator[getResult(?Goal,?Serv) -> ?Result])
        and insert?Result.
```

The `findService` transaction performs the following tasks:

1. For each service instance, *s*, it finds the mediator that the service advertises. The mediator is then used to construct an input to that service. In turn, the input is used to compute the effects that the service guarantees to be true in the after-state of the execution.
2. The discovery engine then hypothetically inserts the effects into the knowledge base and, using the mediator, checks whether the user goal is true in the after-state of the service *s*. The result of this hypothetical execution is returned and saved in the corresponding goal object.
3. The hypothetical execution succeeds iff service *s* matches the goal. After the execution, the state of the knowledge base is what it was before the execution of the service, but the variable `?Result` is now bound to a formula of the form `goal[result->s]`, which represents one result of the discovery. This is then inserted into the knowledge base. In this way, the set of answers to the goal is built as the value of the `result` attribute of the goal object.

For instance, if the query `?- findService(goal1)` is executed then the following will become true: `goal1[result -> serv1,serv2]`. Similarly, executing `?- findService(goal2)` yields `goal2`[`result ->` `serv2`]. Executing the query `?- findService(goal3)` yields `goal3`[`result ->` `serv1`].

More interesting is `goal4`, which requests citipasses for an entire region (France). Given the information available in our knowledge base, only `serv3` should match. Note that `serv4` does not match because it does not serve Paris, while the goal specifies only those services that can sell citipasses for *every* location in France.

7 Conclusion

We presented a non-trivial use case that illustrates Web service discovery using semantic descriptions specified in SWSL-Rules. The example illustrates a number of important features that, we believe, must be part of any language for semantic Web services. These features include frame-based representation, reification, non-monotonic reasoning (as exemplified by non-monotonic Lloyd-Topor extensions), and Transaction Logic based updates. We have also illustrated how mediators can be specified and used in our language.