Semantic Matching of Web Services Capabilities

Massimo Paolucci[1], Takahiro Kawamura[1,2],
Terry R. Payne[1], and Katia Sycara[1]

[1] Carnegie Mellon University
Pittsburgh, PA, USA
{paolucci, takahiro, terryp, katia}@cs.cmu.edu
[2] Research & Development Center, Toshiba Corp.
1, Komukai Toshiba-cho, Saiwai-ku,
Kawasaki 212-8582, Japan
takahiro@isl.rdc.toshiba.co.jp

Abstract. The Web is moving from being a collection of pages toward a collection of services that interoperate through the Internet. The first step toward this interoperation is the location of other services that can help toward the solution of a problem. In this paper we claim that location of web services should be based on the semantic match between a declarative description of the service being sought, and a description of the service being offered. Furthermore, we claim that this match is outside the representation capabilities of registries such as UDDI and languages such as WSDL.
We propose a solution based on DAML-S, a DAML-based language for service description, and we show how service capabilities are presented in the Profile section of a DAML-S description and how a semantic match between advertisements and requests is performed.

1 Introduction

Web services provide a new model of the Web in which sites exchange dynamic information on demand. This change is especially important for the e-business community, because it provides an opportunity to conduct business faster and more efficiently. Indeed, the opportunity to manage supply chains dynamically to achieve the greatest advantage on the market is expected to create great value added and increase productivity. On the other hand, automatic management of supply chain opens new challenges: first, web services should locate other services that provide a solution to their problems, second, services should interoperate to compose complex services.

In this paper we concentrate on the first problem: the location of web services on the basis of the capabilities that they provide. The solution of this problem requires a language to express the capabilities of services, and the specification of a matching algorithm between service advertisements and service requests that recognizes when a request matches an advertisement. We adopt DAML-S as service description language because it provides a semantically based view of of web services which spans from the abstract description of the capabilities of

I. Horrocks and J. Hendler (Eds.): ISWC 2002, LNCS 2342, pp. 333–347, 2002.

the service to the specification of the service interaction protocol, to the actual messages that it exchanges with other web services.

DAML-S ability to describe the semantics of web services is in stark contrast with emerging XML [14] based standards related with web services. Standards such as SOAP [15] and WSDL [3] are designed to provide descriptions of message transport mechanisms, and for describing the interface used by each service. However, neither SOAP nor WSDL are of any help for the automatic location of web services on the basis of their capabilities. Another emerging XML based standard is UDDI [13]; it provides a registry of businesses and web services. UDDI describes businesses by their physical attributes such as name, address and the services that they provide. In addition, UDDI descriptions are augmented by a set of attributes, called `TModels`, which describe additional features such as the classification of services within taxonomies such as NAICS [2]. Because UDDI does not represent service capabilities, it is of no help to search for services on the basis of what they provide.

A limitation shared by the XML based standards described above is their lack of an explicit semantics: two identical XML descriptions may mean very different things depending on the context of their use. This proves to be a major limitation for capability matching: in fact, one crucial aspect of capability matching is that it can be done only at the semantic level. This is the case because the requester does not know what services are provided at any given time, otherwise it could contact the providers directly without need to search them; furthermore, advertisers and requesters have very different perspectives and different knowledge about the same service. The major problem with capability matching is that it is unrealistic to expect advertisements and requests to be equivalent, or even that exists a service that fulfills exactly the needs of the requester. For example, a service may advertise as a financial news provider, while a requester may need a service that reports stock quotes. The task of the matching engine is to use its knowledge of the World and its semantic understanding of the advertisement and request to recognize their degree of mismatch and retrieve the advertisements of services that more closely match the request.

DAML-S supports our need for semantic representation of services through its tight connection with DAML+OIL [4]. DAML+OIL supports subsumption reasoning on taxonomies of concepts. Furthermore, DAML+OIL allows the definition of relations between concepts so that, for instance, it is possible to express statements like `X is part of Y` or more generally that a relation R exists between X and Y. The main limitation of DAML+OIL is its lack of a definition of well formed formulae and an associated theorem prover. While these limitations affect the expressivity of advertisements and requests, the language and the reasoning that it supports are rich enough to allow the description of a wide range of services and to allow matches between these descriptions.

In the rest of the paper, we describe DAML-S profiles to some detail; we will then discuss a matching algorithm between advertisements and requests described in DAML-S that recognizes various degrees of matching. We will then

conclude by showing how DAML-S and an implemented version of the matching algorithm are used to provide capability matching to the UDDI registry.

2 DAML-S Profiles

The objective of a Service Profiles is to describe the functionalities that a Web Service wants to provide to the community. Web Services may have many functionalities, but not all of them have to be advertised. For example, a book-selling service may provide two different functionalities: the first one is to allow other services to browse its data base to find books of interest; the second one is to allow them to buy the books they found. The book-seller has the choice of advertising just the book-buying service or both the browsing functionality and the buying functionality. In the latter case the service makes public that it can provide browsing services, implicitly allowing other services to browse its data base without buying a book. In contrast, by advertising only the book-selling functionality, but not the browsing, the service hides the browsing functionality from requesters that do not intend to buy. The decision as to which functionalities to advertise determines how the service will be used: a requester that intends to browse but not to buy would select a service that advertises both buying and browsing capabilities, but not one that advertises buying only.

Figure 1 shows the upper ontology for Service Profiles, an example of Service Profile used to advertise a service is shown in figure 7. The figure is logically divided in three parts: the bottom consists of the definition of *Actor*: it records information about the provider of the service. The middle part describes the *Functional Attributes* such as Quality Rating, that is the rating assigned to the service, to Geographic Radius, that specifies whether there are geographic constraints to the service. Such constraints are used to prevent that a request for Chinese food issued in Pittsburgh is served by a restaurant in Shanghai.

The top part of the figure represents the *Functional Description* of the service [12,11]. It describes the capabilities of the service in terms of inputs, outputs, preconditions and effects. An input is what is required by a service in order to produce a desired output. For example, the inputs of a book buying service are the title and the author of the desired book. The output is a confirmation that the order has been received and successfully processed. The preconditions represent conditions in the World that should be true for the successful execution of the service. In the book buying example a precondition would be a valid credit card. The execution of the service may result in actions in the World, these conditions are described as the effects of the agent. In the buying of a book example, the credit card is charged and the book changes property.

Service Profiles describe service requests as well as services provided. A request consists of a description of an hypothetical service that performs a task needed by the requester. For instance, a requester that needs the latest quotes from the stock market may compile a profile of an hypothetical financial news service. Requests are sent to registries of web services that match them against

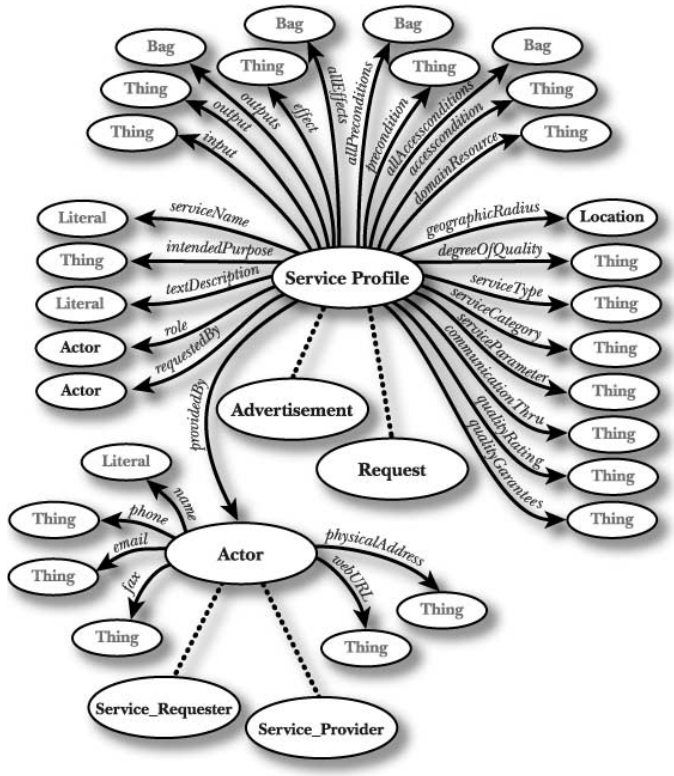

Fig. 1. Upper Ontology of Service Profiles

the profiles advertised by other services to identify which services provide the best match. An example of request is shown in figure 8.

3 Matching Engine

We envision a Web wide infrastructure for web services supported by a set of registries that function as directories. These registries record advertisements of services that come on line, and support search of services that provide a set of requested functionalities. In this section we describe an algorithm for matching service advertisements and service requests.

An advertisement matches a request, when the advertisement, describes a service that is *sufficiently similar* to the service requested. Of course, the problem of this definition is to specify what "sufficiently similar" means. In its strongest interpretation, an advertisement and a request are "sufficiently similar" when they describe exactly the same service. This definition is too restrictive, because advertisers and requesters have no prior agreement on how a service is represented; furthermore, they have very different objectives. A restrictive criteria on

matching is therefore bound to fail to recognize similarities between advertisements and requests.

To accommodate a softer definition of "sufficiently similar" we need to allow matching engines to perform *flexible* matches, i.e. matches that recognize the degree of similarity between advertisements and requests. Service requesters should also be allowed to decide the degree of flexibility that they grant to the system. If they concede little flexibility, they reduce the likelihood of finding services that match their requirements, i.e. they minimize the false positives, while increasing the false negatives. On the other hand, by increasing the flexibility of match, they achieve the opposite effect: they reduce the false negatives at the expense of an increase of false positives.

An additional problem related with performing flexible matches is that the Matching Engine is open to exploitation from advertisements and requests that are too generic in the attempt to maximize the likelihood of matching. For instance, a service may advertise itself as a provider of everything, rather than to be honest and precise with what it does. Similarly, a the requester may ask for any service, rather than specifying exactly what it expects. The matching engine can reduce the efficacy of these exploitations by ranking advertisements on the basis of the degree of match with the request.

In a nutshell, we expect the matching engine to satisfy the following desiderata:

- The matching engine should support flexible semantic matching between advertisements and requests on the basis of the ontologies available to the services and the matching engine.
- Despite the flexibility of match, the matching engine should minimize false positives and false negatives. Furthermore, the requesting service should have some control on the amount of matching flexibility it allows to the system.
- The matching engine should encourage advertisers and requesters to be honest with their descriptions at the cost of paying the price of either not be matched, or being matched inappropriately.
- The matching process should be efficient: it should not burden the requester with excessive delays that would prevent its effectiveness..

The algorithm we propose strives to satisfy all four desiderata. Semantic matching is based on DAML ontologies: advertisements and requests refer to DAML concepts and the associated semantic. By using DAML, the matching process can perform inferences on the subsumption hierarchy leading to the recognition of semantic matches despite their syntactic differences and difference in modeling abstractions between advertisements and requests.

The use of DAML also supports accuracy: no matching is recognized when the relation between the advertisement and the request does not derive from the DAML ontologies used by the registry. Furthermore, the semantic of DAML-S descriptions allows us to define a ranking function which distinguishes multiple degrees of matching.

Finally, the matching process is necessarily a complex mechanism that may lead to costly computations. In order to increase efficiency, the algorithm de-

scribed here adopts a set of strategies that rapidly prune advertisements that are guaranteed not to match the request, thus improving the efficiency of the overall matching engine while maintaining its precision.

3.1 Matching Algorithm

The main rational behind our algorithm is that an advertisement matches a request when the service provided by the advertiser can be of some use for the requester. Specifically, an advertisement matches a request when all the outputs of the request are matched by the outputs of the advertisement, and all the inputs of the advertisement are matched by the inputs of the request. This criteria guarantees that the matched service satisfies the needs of the requester, and that the requester provides to the matched service all the inputs that it needs to operate correctly.

In this section we discuss the matching algorithm in some detail. We will first present the main loop in which a request is matched against all the advertisements recorded by the registry; then we will discuss the rules for matching each advertisement with the request; we will then show how the degree of match is computed and how the results of the match are sorted. We will conclude the section with an example and a discussion of how the matching algorithm proposed satisfies the desiderata listed above.

The main control loop of the matching algorithm is shown in figure 2. Requests are matched against all the advertisements stored by the registry. Whenever a match between the request and any of the advertisements is found, it is recorded and scored to find the matches with the highest degree.

```
match(request) {
   recordMatch= empty list
   forall adv in advertisements do {
      if match(request, adv) then
          recordMatch.append(request, adv) }
   return sort(recordMatch);}
```

Fig. 2. Main control loop

A match between an advertisement and a request consists of the match of all the outputs of the request against the outputs of the advertisement; and all the inputs of the advertisement against the inputs of the request. The algorithm for output matching is described in detail in figure 3: a match is recognized if and only if for each output of the request, there is a matching output in the advertisement. The degree of success depends on the degree of match detected. If one of the request's output is not matched by any of the advertisement's output the match fails. The matching between inputs is computed following the same algorithm, but with the order of the request and the advertisement reversed:

whereas the request's outputs are matched against the advertisement's outputs, the advertisement's inputs are matched against the request's inputs.

```
outputMatch(outputsRequest, outputsAdvertisement) {
   globalDegreeMatch= Exact
   forall outR in outputsRequest do {
      find outA in outputsAdvertisement such that
         degreeMatch= maxDegreeMatch(outR,outA)
         if (degreeMatch=fail) return fail
         if (degreeMatch<globalDegreeMatch)
            globalDegreeMatch= degreeMatch
   return sort(recordMatch);}
```

Fig. 3. Algorithm for output matching

The degree of match between two outputs or two inputs depends by the relation between the concepts associated with those inputs and outputs. For instance, consider how a request whose output is specified as `vehicle` matches the advertisement of a car selling service whose outputs are `car` and `price`. Given the ontology fragment shown in figure 5, the matching engine would match `vehicle` with `car` instead of matching it with `price`, because `car` is subsumed by `vehicle`, while no subsumption relation is found between `vehicle` and `price`.

```
degreeOfMatch(outR,outA):
   if outA=outR then return exact
   if outR subclassOf outA then return exact
   if outA subsumes outR then return plugIn
   if outR subsumes outA then return subsumes
   otherwise fail
```

Fig. 4. Rules for the degree of match assignment

The degree of match is determined by the minimal distance[1] between concepts in the taxonomy tree. We differentiate between four degrees of matching according to the rule displayed in figure 4, where `outR` corresponds to one output of the request and `outA` corresponds to one output of the advertisement[2]. The rational for the degree assignment is described below.

exact If `outR=outA` then `outR` and `outA` are equivalent, which we label as `exact`.

The second clause is a bit more complicated; if `outR subclassOf outA` then

[1] DAML+OIL supports multiple inheritance, therefore there may be more than one path between two nodes. We optimistically always select the shortest.

[2] The degree of match of inputs is assigned in the same way, but the arguments reversed: degreeOfMatch(inA,inR)

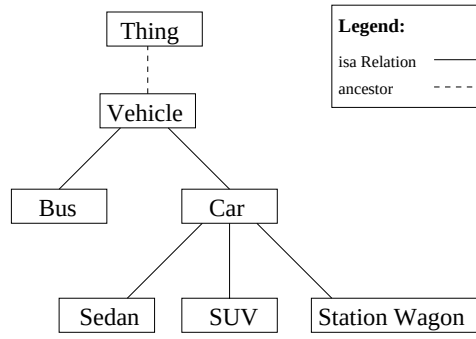

Fig. 5. A fragment of the Vehicle Ontology

the result is still `exact` under the assumption that by advertising `outA` the provider commits to provide outputs consistent with every immediate subtype of `outA`. This is like to say that, given the ontology fragment in figure 5, the provider, by advertising `car`, commits to provide `sedan`, `station wagon` and `SUV`. If instead it provides only `station wagon`, than a better strategy would be to restrict its advertisement to the latter.

plug in If `outA subsumes outR`[3] than `outA` is a set that includes `outR`, or, in other words, `outA` could *be plugged* in place of `outR` [16]. For example, the a service that provides (any type of ...) `vehicles` could be of use for another service that expects `station wagons`. This rule acknowledges that there is a weaker relation between `outR` and `outA` in this case, than in the exact case above: we can expect that a service that advertises an output of vehicle provides some type of cars, but we cannot expect that it provides every type of SUV.

subsumes If `outR subsumes outA`, then the provider does not completely fulfill the request. The requester may use the provider to achieve its goals, but it likely needs to modify its plan or perform other requests to complete its task.

fail Failure occurs when no subsumption relation between advertisement and request is identified.

Degrees of match are organized along a discrete scale in which exact matches are of course preferable to any another; plugIn matches are the next best level because the output returned can probably be used instead of what the requester expects. Subsumes is the third best level since the requirements of the requester are only partially satisfied: the advertised service can provide only some specific cases of what the requester desires. Fail is the lower level and it represents an unacceptable result.

The last piece of the algorithm to discuss is the scoring system used to sort the resulting matches. The rules used to sort are shown in figure 6. The

[3] subclassOf in DAML also defines a subsumption relation, therefore the exact match defined above is also based on the subsumption relation. The rules for *plug in* matching apply when the concepts are not the same and no subclassOf relation holds.

rationale behind them is that the requester expects first and foremost that the provider achieves the output requested at the highest degree. This is reflected in our rules by establishing that the main sorting criteria is to select the match with the highest score in the outputs. Input matching is used only as secondary score to break ties between equally scoring outputs: the requester may solve any mismatch between the information that it has available and the expectations of the provider with additional problem solving or by querying the registry to find additional providers.

```
sortRule(match1,match2) {
   if match1.output > match2.output then match1 > match2
   if match1.output = match2.output
      & match1.input > match2.input then match1 > match2
   if match1.output = match2.output
      & match1.input = match2.input then match1 = match2
```

Fig. 6. Rules for the degree of match assignment

3.2 An Example: Looking for Cars

In this section we show a simple example of how a request for service is matched with service advertisements. The service advertised is a car selling service which given a price reports which car can be bought for that price. A strip down version of the advertisement for the service is shown in figure 7: it shows that the inputs expected by the service are restricted to instances of the concept *Price* as defined in the *Concepts* ontology, while the outputs the service generates are instance of the concept *Car* as defined in the ontology *Vehicle* shown in figure 5.

A request for service is expressed in the same format of the advertisement; a possible request is expressed in figure 8. The request shows that the service sought sells sedans, specifically, it should accepts as inputs to instances of *Price* and it generates as outputs instances of *Sedan*.

The match between the advertisement and the request requires the matching between their inputs and outputs restrictions respectively. For ease of example, both inputs are restricted to the same concept, therefore they match exactly. The algorithm for output matching is shown in figure 3 and 4; it recognizes that *Car* and *Sedan* are an exact match because Car is a superclass of Sedan in the Vehicle ontology displayed in figure 5. As a result the advertisement and the request match exactly because of the exact match of both their inputs and outputs. As a consequence, the Car service advertised is reported to the requester.

The example shows a case of an advertisement and request that look superficially different but match exactly nevertheless using ontological information. More relaxed matches would result if the advertising service produces more general outputs, such as *Vehicle* instead of *Car*. The latter case would result in a

```
<profile:Profile rdf:ID="CarSellingService">
  <profile:serviceName>CarSellingService</profile:serviceName>
  <profile:providedBy> ... </profile:providedBy>
  <input>
    <profile:ParameterDescription rdf:ID="Price_Input">
      <profile:parameterName>Price</profile:parameterName>
      <profile:restrictedTo rdf:resource="Concets.daml#Price"\>
    </profile:ParameterDescription>
  </input>
  <output>
    <profile:ParameterDescription rdf:ID="Car_Output">
      <profile:parameterName>Car</profile:parameterName>
      <profile:restrictedTo rdf:resource="Vehicle.daml#Car"\>
    </profile:ParameterDescription>
  </output>
</profile:Profile>
```

Fig. 7. Advertisement of a car selling service

```
<profile:Profile rdf:ID="RequestSedanSellingService">
  <input>
    <profile:ParameterDescription rdf:ID="Price_Input">
      <profile:parameterName>Price</profile:parameterName>
      <profile:restrictedTo rdf:resource="Concets.daml#Price"/>
    </profile:ParameterDescription>
  </input>
  <output>
    <profile:ParameterDescription rdf:ID="Sedan_Output">
      <profile:parameterName>Sedan</profile:parameterName>
      <profile:restrictedTo rdf:resource="file:data/Vehcle.daml#Sedan"/>
    </profile:ParameterDescription>
  </output>
```

Fig. 8. Advertisement of car selling service

lower degree of matching: *plugIn* instead of *exact* because the output of the advertisement subsumes the output of the request. A failure would instead result if the outputs of the advertisement are instances of *Bus* because no subsumption relation is recognized between the outputs of the advertisement and the outputs of the request.

3.3 Satisfaction of Desiderata

The matching algorithm supports a flexible semantic match between advertisements and requests. The only thing that matters during matching is whether the matching engine can draw an inference between inputs and outputs of the advertisements and requests on the basis of the ontologies available to the registry. Furthermore, the result of the match is not a hard true or false, but it depends on the degree of similarity between the concepts in the match.

Despite this flexibility, the matching engine still rejects advertisements that do not match the requests, and accepts, but with a low score, matches that may be unsatisfactory for the requester. The requester can specify which types of match it wishes the matching engine to perform by constraining the minimal acceptable degree of match. Also, the amount of search required may be constrained by forcing the matching engine to restrict the search within a close subset of concepts in the ontology. The last desiderata: that the matching process be efficient is currently under testing.

4 Application to UDDI

Universal Description Discovery and Integration (hereafter UDDI)[13] is an industrial initiative whose goal is to create an Internet wide registry of web services. UDDI allows businesses to register their contact points, and the web services that they provide. UDDI supports the registration of attributes of services via a construct called *TModel*. A TModel is a form of meta data that provides a reference system for information about services. For instance services can specify that they are based on the WSDL specification by referring to a publicly known WSDL TModel. In general TModels have two functions: the first is to tag the type of service advertised and whether some specific conventions on the use of the UDDI registry have been applied. The second is to provide abstract keys to be associated with a service specific value. For example, a service may specify its category using the North American Industry Classification System (hereafter NAICS) [2] published by the US Census.

UDDI provides poor search facilities: it allows only a keyword based search of businesses, services and TModels on the bases of their names. In addition services can be searched by their type specification through TModels. For instance, it is possible to search for all the services that adhere to the WSDL representation or that have a some value associated with a TModel. Since search in UDDI is restricted to keyword matching, no form of inference or flexible match between keywords can be performed.

We implemented a matching engine that can be used to augment UDDI registries[4] with an additional semantic layer that performs a capability based matching. The matching engine that we implemented is based on the algorithm described above and it takes advantage of DAML ontologies published on the web. The result of this work is that services that advertise using DAML-S are also advertised with the UDDI registry, and therefore they can be found and retrieved by using UDDI keyword search. In addition, they can also be found through our capability matching engine.

The architecture of the combined DAML-S/UDDI Matchmaker is described in figure 9. The Matchmaker receives messages from outside through the *Communication Module*; upon recognizing that a message is an advertisement, the Communication Module sends it to the *DAML-S/UDDI Translator* that constructs a UDDI service description using information about the service provider,

[4] We are currently using the IBM test site.

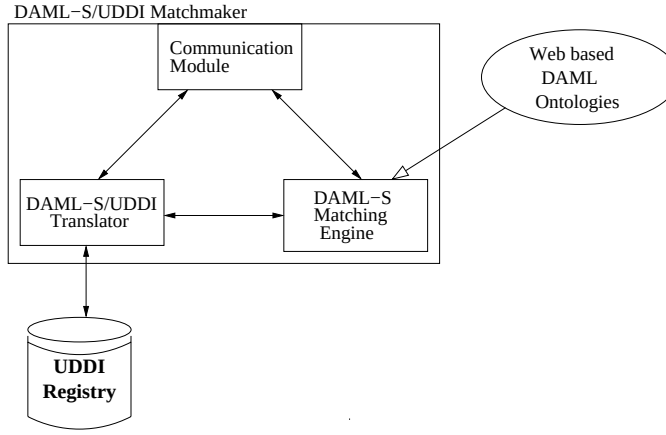

Fig. 9. The architecture of the DAML-S/UDDI Matchmaker

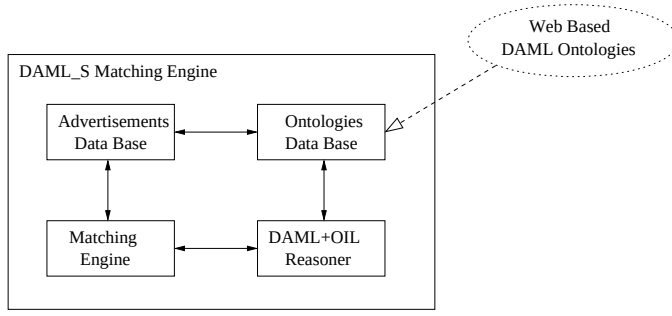

Fig. 10. The architecture of the DAML-S Matching Engine

and the service name. The result of the registration with UDDI is a reference ID of the service. This ID combined with the capability description of the advertisement are sent to the DAML-S Matching Engine that stores the advertisement for capability matching. Requests follow the opposite direction: the Communicator Module sends them to the DAML-S Matchmaker that performs the capability matching. The result of the matching is the advertisement of the providers selected and a reference to the UDDI service record. The combination of UDDI records and advertisements is then send to the requester.

The actual DAML-S based matching engine architecture is displayed in figure 10. Upon receiving a request, the *Matching Engine* component selects the advertisements from the *AdvertisementDB* that are relevant for the current request. Then it uses the *DAML+OIL Reasoner* to compute the level of match. In turn the DAML+OIL Reasoner uses the *OntologyDB* to as data to use to compute the matching process. The AdvertisementDB also takes advantage of the *OntologiesDB* to index advertisements for fast retrieval at matching time.

This system show the limits of UDDI and the value added by DAML-S and its support for functional descriptions and matching upon functional descriptions of services. In its current form UDDI does not provide any support for finding services on the basis of what tasks they perform. It is impossible ask UDDI for a "car selling service" because UDDI because such a request cannot even be expressed. By adding an additional layer for service capability matching and by using DAML-S as service capability language we allow services to select each other on the bases of what they do and ultimately to interoperate and solve problems autonomously minimizing human intervention.

5 Discussion

DAML-S and its Service Profile take up the challenge of representing the functionalities of web services. This paper contributes to this challenge by describing a matching engine that allows matching of advertisements and requests on the bases of the capabilities that they describe. This is a major improvement on current technology that allows only location of services based on keyword matching. Indeed we show how the matching engine can be used to improve the functionalities of existing web service repositories such as UDDI.

The Service Profile is an evolution of the work on representation of agents in open Multi-Agents Systems (hereafter MAS) and specifically of LARKS [12]. DAML-S as well as LARKS represents services on the bases of their inputs and outputs. The major difference between DAML-S and LARKS is that DAML-S relies on DAML and its ontologies, while LARKS allowed for their incremental creation by associating needed concepts directly with the advertisements and requests. The two systems rely on similar matching algorithms. LARKS identifies a set of filters that progressively restrict the number of advertisements that are candidates for a match. The filtering mechanism allows services to strike the most advantageous trade off between the precision of matching and the time required for a match: the higher the precision, the longer the time the matchmaker needs before delivering an answer. The matching engine described in this paper is based on the more restrictive of the LARKS filters that performs logic and ontological inferences between advertisements and requests. While, the filters adopted by LARKS cannot be efficiently ported into DAML-S, we suggest similar filters that achieve the same results.

The Multi-Agent community has addressed the problem of capability based matching in an open MAS suggesting a number of solutions. The OAA [6] represents agents by their "solvables": a representation of the queries the agent replies to. The problem with OAA solvables is that any agent should know at request time what solvables the provider replies to, but the solvables are not known until the provider is selected. Ultimately this impasse can be solved only by abstracting from the solvables to the information that is exchanged. Infosleuth [9] associates an ontological concept with each type of services that agents perform, then at matching time, it selects only those services that perform the desired function. In practice InfoSleuth uses an extensive representation of func-

tionalities (one concept for each possible type of services), while DAML-S use an intensive representation in which services are implicitly defined by the transformation that they produce. More recently DReggie [8] defined an ontology based on DAML+OIL to describe mobile devices and then use a matching engine to locate devices on the bases of their features. Unfortunately, publicly available descriptions of the system are still sketchy.

Software Reuse Systems also need to index software components appropriately for efficient and precise retrieval. Still, work on software reuse differs sharply from our attempt to represent and match web services principally because software reuse requires programmers, rather than automatic services, to construct a request for a software component to search; furthermore, our aim with DAML-S as a whole, is automatic interaction between services, while work in software reuse requires programmers to program the interaction between different software components. Because of this difference, techniques like the faceted classification [10] are of no use to help automatic queries since they represent features of the providers rather than the goals it achieves. Techniques such as analogical software reuse [7] share a representation of components that is based on goals achieved by the software, roles, conditions. To this extent their approach is similar to ours, but it requires a complex compilation of a case to match against. Zaremsky and Wing [16] describe a specification language and matching mechanism for software components that bear many similarities with the Matching Algorithm described here. As in our work they allow for multiple degrees of matching. We depart from their work because we match on the semantics associated with inputs and outputs, while they consider only type information. Of all the reuse models UPML [5] shares the greater similarities with our representation by representing inputs, outputs, preconditions and effects of tasks. Nonetheless, UPML still requires programmers in the loop.

Despite superficial similarities with Case Based Reasoning Systems (CBR), and specifically CBR supported planning [1], the work described here is very different. The goal of Case Base Reasoning Systems is to retrieve a previously learned case and to adapt it to the problem solving case that they are facing. To this extent they have a fix retrieval function while here is flexible retrieval mechanism is used. Furthermore, when a profile is retrieved by the repository it is not applied as a case, rather the requesting service and the provider interact following a script described by the DAML-S Process model.

The result of the research effort shows that web services can indeed find each other automatically and interoperate autonomously without the need of hardcoded interactions. Our matching algorithm provides a way for automatic dynamic discovery, selection and interoperation of web services, which is a crucial feature in the web of the future in which services dynamically reconfigure their supply chain to better match changes in the market.

References

1. Jim Blythe and Manuela Veloso. Analogical replay for efficient conditional planning. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 668–673. AAAI Press / MIT Press, 1997.
2. US Census Bureau. North american industry classification system (naics). http://www.census.gov/epcd/www/naics.html, 1997.
3. Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315, 2001.
4. DAML Joint Committee. Daml+oil (march 2001) language. http://www.daml.org/2001/03/daml+oil-index.html, 2001.
5. Dieter Fensel, V. Richard Benjamins, Enrico Motta, and Bob J. Wielinga. UPML: A framework for knowledge system reuse. In *IJCAI*, pages 16–23, 1999.
6. David Martin, Adam Cheyer, and Douglas Moran. The Open Agent Architecture: A Framework for Building Distributed Software Systems. *Applied Artificial Intelligence*, 13(1-2):92–128, 1999.
7. P. Massonet and A. van Lamsweerde. Analogical reuse of requirements frameworks. In *Proc. of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97)*, pages 26–39, 1997.
8. Yun Peng and Nenad Ivezic. Semantic resolution inf multi-agent systems. In *Proc.of Goddard/JPl Workshop On Radical Agent Concepts*, 2002.
9. Brad Perry, Malcolm Taylor, and Amy Unruh. Information aggregation and agent interaction patterns in infosleuth. In *cia99*. ACM Press, 1999.
10. Ruben Prieto-Diaz. Implementing Faceted Classification for Software Reuse. *Communications of ACM*, 134:88–97, 1991.
11. Katia Sycara and Mattheus Klusch. Brokering and matchmaking for coordination of agent societies: A survey. In Omicini et al, editor, *Coordination of Internet Agents*. Springer, 2001.
12. Katia Sycara, Mattheus Klusch, Seth Widoff, and Janguo Lu. Dynamic service matchmaking among agents in open information environments. *ACM SIGMOD Record (Special Issue on Semantic Interoperability in Global Information Systems)*, 28(1):47–53, 1999.
13. UDDI. The UDDI Technical White Paper. http://www.uddi.org/, 2000.
14. W3C. Extensible markup language (xml) 1.0 (second edition). http://www.w3.org/TR/2000/REC-xml-20001006, 2000.
15. W3C. Soap version 1.2, w3c working draft 17 december 2001. http://www.w3.org/TR/2001/WD-soap12-part0-20011217/, 2001.
16. Amy Moormann Zaremski and Jeannette M. Wing. Specification matching software components. *ACM Transactions on Software Engineering and Methodology*, 1997.