Semantic Matchmaking of Web Services using Model Checking *

Akın Günay
Department of Computer Engineering
Boğaziçi University
Bebek, 34342, Istanbul,Turkey
akin.gunay@boun.edu.tr

Pınar Yolum
Department of Computer Engineering
Boğaziçi University
Bebek, 34342, Istanbul,Turkey
pinar.yolum@boun.edu.tr

ABSTRACT

Service matchmaking is the process of finding suitable services given by the providers for the service requests of consumers. Previous approaches to service matchmaking is mostly based on matching the input-output parameters of service requests and service provisions. However, such approaches do not capture the semantics of the services and hence cannot match requests to services effectively. This paper proposes an agent-based approach for matchmaking that is based on capturing the semantics of services and requests formally through temporal logic. Requests are represented as a set of properties and compared to the service representations using model checking, yielding results on whether a service can satisfy a request or not. By help of domain ontologies, our approach also supports flexible matching, where partially matching services are identified. We provide a general framework, where our approach can work with other existing matchmaking approaches and is integrated with current efforts such as OWL-S and SWRL.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent Systems

General Terms

Algorithms

Keywords

Web service, matchmaking, model checking

1. INTRODUCTION

Web services are software that provide a functionality and can be invoked over the Web in a machine independent manner [17]. An important challenge in the usage of Web services is discovery of appropriate Web services for different service needs. For this purpose, the current standard is the use of Universal Description Discovery and Integration (UDDI) directories, where services are described in Web Service Description Language (WSDL). UDDI is designed for human users and does not provide facilities for machine processing and automated discovery, which is crucial with the increasing number of available services.

One of the most influential approaches to this problem is input-output matching, where a service request is considered to match a Web service if the inputs and outputs are identical [11, 12, 14]. In this approach, when the input-output fields do not match identically, the semantics of the inputs and outputs can be used to compute the degree of match between concepts using subsumption relationships presented by ontologies. Although input-output matching is easy to implement, it cannot guarantee the precision of the results. That is, since it does not consider the internal process of services while performing the matchmaking operation, it is likely that different services with identical interfaces are counted as good matches although they perform completely different tasks.

In this paper we propose a novel agent-based service matchmaking approach that captures the semantics of both the services and the requests. We use Process or Protocol Meta Language (Promela) [8] to model the services and linear temporal logic (LTL) [6] formulae to model the requests of the consumers. Using Spin [8] as our model checker, we check if the requests are satisfied by the services. Our approach can find exact matches based on semantics of consumers requests and existing services. If the consumer requests cannot be satisfied with existing services, our approach computes the closest alternative service requests to the original request that can be satisfied by the available services. To do so, it uses a process ontology that models the atomic processes contained by the services as semantic concepts and defines semantic relations between these processes. Specifically, the ontology is used to generate similar requests to the original request that can be provided by the available services. This enables our approach to support partial matches. As a result, the consumers can choose among existing services consciously.

It is important that a new matchmaking approach be compatible with existing and emerging technologies. To address this, we show how Promela models can be presented in OWL-S [3]. This allows our matchmaker to work with regular OWL-S specifications. To help consumers generate

*This research has been partially supported by Boğaziçi University Research Fund under grant BAP07A102 and the Scientific and Technological Research Council of Turkey by a CAREER Award under grant 105E073. The first author is supported by a Graduate Scholarship Program from the Scientific and Technological Research Council of Turkey.

Cite as: Semantic Matchmaking of Web Services using Model Checking, Akın Günay, Pınar Yolum, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. XXX-XXX.
Copyright © 2008, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

their requests, we extended Semantic Web Rule Language (SWRL) [9] to show also LTL formulae. These extensions are part of our general matchmaking framework where our matchmaking approach can be integrated with other matchmaking approaches.

The rest of this paper is organized as follows. Section 2 explains our representation of semantic service requests with underlying logical structure and examples. Section 3 describes our representations of services and use of model checking for matching requests and services. Section 4 explains our matching approach in detail. Section 5 describes our general matchmaking framework. Section 6 explains our case study and elaborates on our evaluations. Finally, Section 7 reviews some relevant literature and provides directions for future work.

2. SEMANTIC REQUEST DESCRIPTION

In order to allow semantic matchmaking, the consumers should be able to express their request semantically, rather than only expressing input-output parameters or general keywords. To capture semantic description of requests, we use LTL. Each request is a set of LTL properties that define the functional and temporal requirements of the consumer from the service.

2.1 Linear Temporal Logic (LTL)

LTL is a temporal logic, where the future is seen as a sequence of states or simply as a path. LTL formulae are built up from a set of proposition variables, the usual logic connectives and four temporal modal operators X , F , G and U . X stands for *next*. It is a unary operator and it means that its bounded proposition must hold at the next state of the given path. F stands for *eventually*. It is another unary operator and it means that that its bounded proposition must hold eventually at some future state(s) of the given path. G stands for *globally* and it is a unary operator. It means that its bounded proposition must hold at all future states of the given path. U stands for *until* and it is the only binary operator. It means that the first proposition bounded to U must hold until the second proposition starts to hold. U also requires that the second proposition must hold in some future state.

2.2 Example Semantic Requests

The following properties are example requirements of a consumer in selecting a service. These examples are on e-commerce domain, however the idea can be applied to any domain. In the following, in LTL formulae each letter represents a fact. For instance to state the fact that the payment is made for a good we write p, which can also be interpreted as $isPaid(payment, good)$. However, to simplify the formulae we use just letters and explain the meaning of each fact explicitly in the text.

Guarantee delivery or refund after payment: The service consumer requests a service that guarantees to deliver the ordered goods after the payment is done. In the exception case, if the payment is done but there is no delivery of the ordered item (due to the cancellation of the order by the consumer or a problem faced by the provider, which prevents the provider from delivery of the ordered goods), than the payment must be refunded by the service provider. Following LTL formula corresponds to this situation, where p, represents $isPaid(payment, good)$, d represents $isDelivered(good, customer)$ and r represents $isRefunded(p)$. Formally, $\mathsf{G}(p \rightarrow ((p\mathsf{U}d) \vee (p\mathsf{U}r)))$.

Expected final condition: This property is the expected final condition, where the world is either in a state so that the consumer made the payment and the provider delivered goods (successful transaction) or in a state, where there is no payment and delivery action performed by any of the participants (canceled transaction). In the formula p represents $isPaid(payment, good)$ and d represents $isDelivered(good, customer)$. Formally, $\mathsf{F}((p \wedge d) \vee (\neg p \wedge \neg d))$.

Delivery before payment: For some reason (i.e. the consumer is going to use the service for the first time and she does not trust the service) a consumer may request from a service that the delivery of the good is made before the payment for these goods. This fact is represented by the following LTL formula. In the formula p represents the fact $isPaid(payment, good)$ and d represents $isDelivered(good, customer)$. Formally, $\mathsf{G}(d \rightarrow (d\mathsf{U}p))$.

Secure connection while doing payment: Consumer prefers secure connection while doing payment. The following property represents the request that a secure connection must be established and the connection stay in this secure state until the payment operation completed. Otherwise the payment operation is not performed. In the formula s represents the fact $isSecure(connection)$ and p represents $isPaid(payment, good)$. Formally, $\mathsf{G}((s \rightarrow (s\mathsf{U}p)) \vee \neg p)$.

Secure connection for all the transactions: A more suspicious consumer who is concerned more about her privacy may not feel comfortable with the previous property and may require a service, where the entire connection is secure. This property is represented by the following formula where s represents the fact $isSecure(connection)$. Formally, $\mathsf{G}(s)$.

2.3 Extending SWRL for LTL

To preserve compatibility with existing work on Web services, we use OWL-S to represent service requests. However, since OWL-S does not support LTL implicitly, we need to create a new schema. OWL-S provides `Expression` class to embed expressions in XML format into OWL-S descriptions. However there is no standard schema is available to represent LTL in XML. Instead of creating a completely new XML schema to represent LTL, we choose to extend the existing SWRL, which provides appropriate facilities. SWRL is the rule language to combine Horn like rules with OWL knowledge bases. SWLR is based on the combination of OWL DL and OWL Lite sublanguages of OWL and Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language. Since SWRL supports only Horn like rules, it is not capable to represent every LTL formula. However the SWRL-FOL [15] extension of SWRL allows expression of First Order Logic (FOL), so we extend SWRL-FOL with LTL connectives U , G , F and X . The abstract syntax of our extension is as follows:

```
axiom := assertion
assertion := 'Assertion('
    [URIref] {annotation} formula {foformula}
    ')'
foformula := atom
    | 'until(' foformula foformula ')'
```

```
    | 'globally(' foformula ')'
    | 'finally(' foformula ')'
    | 'next(' foformula ')'
    | 'and(' {foformula} ')'
    | 'or(' {foformula} ')'
    | 'neg(' foformula ')'
    | 'implies(' foformula foformula ')'
    | 'equivalent(' foformula foformula ')'
    | 'forall(' variable {variable} foformula ')'
    | 'exist(' variable {variable} foformula ')'
variable := 'I-variable(' URIref description ')'
    | 'D-variable(' URIref dataRange ')'
```

The following XML file expresses the LTL formula $\mathsf{G}(p \rightarrow ((p\mathsf{U}d))$, where p stands for payment and d stands for delivery.

```
<Assertion owlx:name="SWRL-LTL Example">
 <owlx:Annotation>
  <owlx:Label>SWRL-LTL rule example</owlx:Label>
 </owlx:Annotation>
 <Globally>
  <ruleml:Var type="xsd:bool">payment</ruleml:Var>
  <ruleml:Var type="xsd:bool">delivery</ruleml:Var>
  <Implies>
   <swrlx:classAtom>
    <owlx:Class owlx:name="isPaid"/>
    <ruleml:var>payment</ruleml:var>
   </swrlx:classAtom>
   <Until>
    <swrlx:classAtom>
     <owlx:Class owlx:name="isPaid"/>
     <ruleml:var>payment</ruleml:var>
    </swrlx:classAtom>
    <swrlx:classAtom>
     <owlx:Class owlx:name="isDelivered"/>
     <ruleml:var>delivery</ruleml:var>
    </swrlx:classAtom>
   </Until>
  </Implies>
 </Globally>
</Assertion>
```

3. SEMANTIC MATCHMAKING

Service matchmaking is the process of selecting appropriate service(s) from a set of available services for a service request. In other words, when we check whether a service matches to a request, we actually check whether some functional and temporal properties required by the consumer are satisfied by the service or not. When the matchmaking is done semantically, then the properties of the service request are related to the internal workings of the service as shown in Section 2. To compute whether such properties hold for various services, we need to represent the semantics of services and be able to check whether the semantic properties hold for the services.

3.1 Service Semantics

A service description provides information about what the service does, where the service is located and how to communicate with the service. The leading ontology for representing services is OWL-S. OWL-S provides mechanisms to annotate service information with semantic constructs. An OWL-S service definition contains a *service profile*, *service grounding* and *service model*. Service profile includes basic information such as its name, inputs, outputs, and so on. Service grounding provides information about where to find the service and the protocol to interact with it. Service model gives detailed information about the internal process of the service. That is the step by step explanation of how the given inputs are processed and outputs are produced in particular situations. In our matchmaking approach we use OWL-S to describe services. A service description in OWL-S provides all the information that we require to build a model of the service via its service model section.

3.2 Matching Request to Services

Since we express service requests semantically by LTL properties as explained in Section 2, we use model checking to find out whether a service satisfies these properties. Model checking [8] is a method to formally verify distributed systems and protocols. In model checking the aim is to verify that certain properties hold for a system. To verify a system, the model checker exhaustively checks all possible executions of a system against the specified properties.

In this study we use Spin Model Checker, since it supports verification of LTL formulae, with which our service requests are described. Spin uses Promela as the specification language, which allows dynamic creation of concurrent processes to model synchronous or asynchronous distributed systems. A Promela model consists of processes, message channels, and variables. Processes model the concurrent entities in the system. Message channels and variables can be defined locally or globally and message channels might be created as synchronous or asynchronous behavior. Promela provides case selection, repetition and unconditional jump flow control structures. Each case selection structure may include one or more possible cases, where each case has an initial statement that is called the guard statement. The case is selected only if its guard statement is true. If more than one guard statement is true, one of the corresponding cases is selected non-deterministically.

Note that, in order to use Spin, we need to translate OWL-S service descriptions to Promela. We do this translation using the rules defined by Ankolekar *et. al.* [1]. Then, using Spin, we verify the service request properties on the Promela translations of services.

4. MATCHMAKING APPROACH

As is customary, our proposed agent-based matchmaking approach works by accepting requests from consumers (either human or other agent) and returning a set of services that matches the request with additional information as the degree of match for each service. This meta-information is useful for helping the consumer choose the most appropriate service precisely from a set of services. The matchmaker agent works as follows:

When services register to the matchmaker agent, the agent translates the OWL-S service model of the service into a Promela model that can be used later for model checking. When it receives a service request from a consumer, it first extracts the required properties of the consumer, which are defined as a set of LTL formulae inside the OWL-S request description. Next, the agent checks the required properties against the Promela service models one by one by using Spin model checker. Each property in the request can be

satisfied by a service with a certain degree (as explained in Section 4.1). The degree in which a request is satisfied by a service is the average of satisfaction degrees of each individual requirement in the request. We explain the details of computing degree of match values and their combinations next.

4.1 Computing Degree of Match

In many domains, finding an exact match for a consumer's service request may be difficult if not impossible. When this is the case, it is best to provide the consumer with a set of services that at least partially satisfy the request. Obviously, it is important that the services in this set closely resemble the services the consumer is interested in. That is, the matchmaker should at least find a similar service for consumers' requests. However, when this is the case, the matchmaker should also assign a degree of similarity to its matches using a meaningful metric. To achieve this, our matchmaking approach associates a degree of match value between services and requests. This value is in the range [0, 1], where the value 1 shows that the service satisfies all the requirements of the consumer without any exception and 0 shows that none of the requirements are satisfied. The match degree is computed in two steps. The first step (property matching) computes how well a service satisfies the given request, considering the required properties individually. The second step (priority factoring) combines these satisfaction rates using the priorities of the consumer as expressed in the request.

4.2 Property Matching

Property matching is the comparison of each LTL property in the request with a service and is repeated for each service in the registry. The usual starting point is the testing of the property against a service. If the property is not satisfied by the service, the matchmaker generates *similar* alternatives that can be satisfied by the service. In order to do this, the matchmaker benefits from a process ontology.

Our process ontology consists of a set of atomic processes as concepts and their hierarchical relations, where each atomic process maps to a fact in a service model. For instance, the atomic process (we refer an atomic process simply as process in the rest of the paper) of making a payment for a good is mapped to the fact as $isPaid(payment, good)$ in the service model. The ontology also models the subsumption relations between the processes. For instance, the general process of making a payment is specialized as making a payment by cash, making a payment by credit card and so on. The depth of these specializations is not restricted. For instance payment by credit card can be specialized as payment by visa and payment by master card.

Algorithm 1 explains the property matching. In the algorithm first by using the model checker we check whether the property is satisfied by the service or not (line 1). If the property is satisfied by the service, the degree of match is set to 1 and no further computation is necessary (line 2). If the property is not satisfied as is, we need to check whether the service satisfies alternative properties that are similar to the the required property. Therefore, we query the process ontology to find semantic relations between the processes in the service and the required property (Algorithm 2) and use these relations to generate alternative properties that are similar to the original property (Algorithm 3) (line 4). Next, again using the model checker, we test each of these alternative properties against the service (line 6). If an alternative property is satisfied by the service, we compute the similarity of the alternative property to the original property by using a semantic similarity metric (line 8). To compute the similarity value, first we determine the semantic similarity between each process in the original property with the replaced process in the corresponding alternative property by using the process ontology. There are several semantic similarity metrics in the literature [16] and any one would suffice for this work. After all alternative properties are checked, we determine the alternative property with the maximum similarity to the original property and associate this similarity value as the degree of match of the service for property matching (lines 9, 10).

Algorithm 1 Flexible matchmaking algorithm

Require: Service *serv*
Require: Property *prop*
Require: Ontology *onto*
1: **if** $serv \models prop$ **then**
2: return 1.0
3: **else**
4: $altPropSet = \text{genAltPropSet}(onto, serv, prop)$
5: $highSim = 0.0$
6: **for** $altProp$ in $altPropSet$ **do**
7: **if** $s \models alternativeProperty$ **then**
8: $altSim = \text{compSim}(prop, altProp, onto)$
9: **if** $altSim > highSim$ **then**
10: $highSim = altSim$
11: **end if**
12: **end if**
13: **end for**
14: return $highSim$
15: **end if**

As explained above, Algorithm 1 relies on Algorithm 2 to find related processes between a service request and a service description and on Algorithm 3 to generate alternative properties that are similar to the original property. Let us visit these algorithms next.

Algorithm 2 finds semantic relations between the processes in the required property and the processes in the service. For example, if both the service request and the service description contain a process related to payment, then payment is a process that is returned by Algorithm 2. To do this, the algorithm checks each process in the required property against each process in the service for a semantic relation using the ontology. If there is a semantic relation between these two processes (line 3), then the algorithm adds the relation to a dictionary for future use (line 4). At the end of this process, for each process of the required property, the dictionary holds a set of processes, which are semantically related to the process and are contained by the current service.

Algorithm 3 generates alternative properties using the original property and the dictionary of relations created in Algorithm 2. It enumerates recursively all possible combinations of the relations in the dictionary of relations (line 4) and then creates a new alternative property for each enumerated combination by replacing the processes in the original property with the processes in the enumeration (line 6).

Let us walk through the algorithms with an example.

Algorithm 2 genAltPropSet($onto, serv, prop$)

Require: Service $serv$
Require: Property $prop$
Require: Ontology $onto$
1: **for** $propProc$ in $prop$ **do**
2: **for** $servProc$ in $serv$ **do**
3: **if** semSim($propProc, servProc$) > 0 **then**
4: $relDict[propProc] += servProc$
5: **end if**
6: **end for**
7: **end for**
8: enumAltProp($prop$, $relDict$, $altPropSet$)
9: return $altPropSet$

Algorithm 3 enumAltProp($prop$, $relDict$, $altPropSet$)

Require: Property $prop$
Require: RelationDictionary $relDict$
Require: alternativePropertySet $altPropSet$
1: **if** all $proc$ of $prop$ considered **then**
2: $altPropSet += genAlt$
3: **else**
4: **for** rel in $relDict$ **do**
5: $genalt += proc$
6: enumAltProp($prop$, $relDict$, $altPropSet$)
7: **end for**
8: **end if**

Assume that we have a service, where the first process is ordering a book (`#order`), the next process is the delivery of the book by mail (`#deliver_by_mail`) and the last process is paying for the book with cash (`#pay_by_cash`). The consumer looks for a book selling service and has the required property of delivery of the book before the payment. The consumer also wants to get the delivery by cargo (`#deliver_by_cargo`) and makes the payment with credit card (`#pay_by_creditcard`). We also assume that in our process ontology, we have relations between `#deliver_by_mail` and `#deliver_by_cargo`, and between `#pay_by_cash` and `#pay_by_creditcard` processes.

Algorithm 1 starts by checking if the property can be satisfied by the service (i.e., if there is an exact match). Since the delivery and payment processes are different in the service and in the requested property, this check fails. Therefore, we need to check whether the service can partially satisfy the property or not. To do this, Algorithm 2 determines the relations between the processes in the required property and the service and creates the dictionary to hold the relations. The dictionary will contain the following:

- `property.#pay_by_creditcard` ↔ `service.#pay_by_cash`
- `property.#deliver_by_cargo` ↔ `service.#deliver_by_mail`

Next, Algorithm 3 creates alternative properties from the original property by enumerating all combinations in the dictionary. The alternative properties created in our example is listed in Table 1. After the alternative properties are generated, Algorithm 1 continues by checking each alternative property against the service. If an alternative property is satisfied by the service, it computes the similarity between the alternative and original property based on the average similarity of the individual processes in the original and alternative properties. For instance, if we consider the alternative property 3 in Table 1, the processes `#pay_by_creditcard` and `#deliver_by_cargo` are replaced with `#pay_by_cash` and `#deliver_by_mail` respectively. If we assume that the similarity between `#pay_by_creditcard` and `#pay_by_cash` is 0.8 and between `#deliver_by_cargo` and `#deliver_by_mail` is 0.6, than the overall similarity of the alternative property 3 to the original property is the average of these two values: 0.7. As the last step, the algorithm determines the alternative property with the maximum similarity value and returns this similarity value as the degree of match to the original property for property matching.

Table 1: Alternative Properties

Req. Prop.	`#pay_by_creditcard`	`#deliver_by_cargo`
Alt. 1	`#pay_by_creditcard`	`#deliver_by_mail`
Alt. 2	`#pay_by_cash`	`#deliver_by_cargo`
Alt. 3	`#pay_by_cash`	`#deliver_by_mail`

4.3 Priority Factoring

After the degree of match values are computed in property matching, we combine these values according to property priorities defined by the consumer and reach a final degree of match value at the request level.

Our approach allows definition of two degrees of priority for each property. The first degree of properties are primary properties, which must be satisfied by the service. That is, the degree of match computed for this property in the property level must be greater than 0. Otherwise the matchmaker does not match the service to the request. For instance a consumer may want to guarantee that a secure connection is established before a payment transaction and that this connection stays secure until the transaction is completed. In such a case this property must be defined as a primary property and only the services that satisfy this property are returned by the matchmaker.

Properties in the second degree are called the secondary properties. These properties are recommended by the consumer, however the matchmaker may still match a request to a service which does not satisfy these recommended secondary properties. An example for such a secondary property might be related to the order of the processes. For instance, in a service to buy books, the consumer may prefer to get the books before she makes a payment. However the consumer may still accept services where payment is done before delivery if there is no better alternative and since the service satisfies the primary properties such as buying a book.

For services that satisfy all the primary properties, we compute the degree of match value in the request level as the linear sum of the individual degree of match values computed in the property level. For instance if the request has one primary property with degree of match value 0.8 for a corresponding service and one secondary property with degree of match value 0.6 for the same corresponding service than the overall degree of match value of the service for the property is 0.7. This scheme does not consider any importance between the primary and secondary properties, except that the primary properties must be satisfied by the service but the secondary properties not. If we want to emphasize primary properties further we might weight property

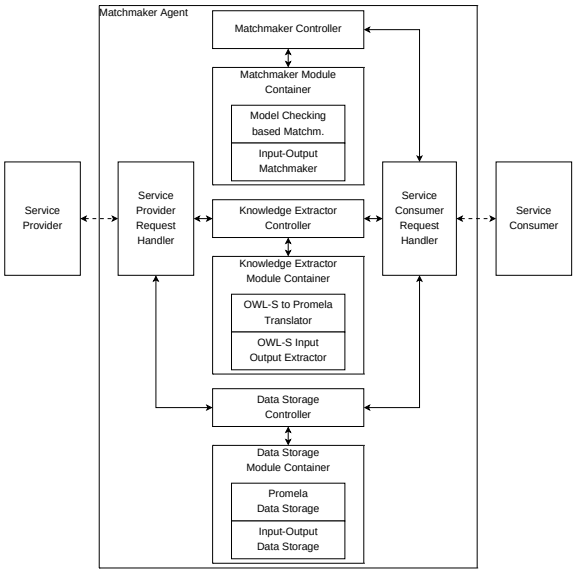

Figure 1: Modular matchmaking framework.

priorities or we might weight each property individually to emphasize importance of some properties for the consumer. Using any of these schemes as the result of matchmaking process we will obtain a set of matching services where each service is associated with a degree of match.

5. THE MATCHMAKING FRAMEWORK

In our study we developed a service matchmaking framework (Figure 1), which provides an abstract architectural structure to realize our matchmaking approach. The advantage of this framework is its modular structure, which allows us to use multiple matchmaking approaches in conjunction.

The framework consists of three main types of entities. These are *service providers*, *service consumers* and the *matchmaking agent*. Service providers are the entities that are capable of providing one or more services to the service consumers. They register their services to the matchmaker agent for advertisement. Service consumers are entities that want to use one or more services provided by the service providers. When they need services they make a request from the matchmaker agent to find matching services for their request. In the framework there is no restriction for the implementation of service providers and service consumers, except that they have capabilities to communicate with the matchmaker agent in a common protocol.

In the framework, the matchmaker agent acts as a middle agent [4] to match service requests of the service consumers to the service advertisements of the service providers. The most important aspect of the matchmaker agent is its modular structure, which allows the use of multiple matchmaking approaches in conjunction. It is possible to specify which matchmaking approach(s) are going to be used by the matchmaker separately for each service request. This allows the consumers to choose between performance and quality of the matchmaking operation according to their needs.

The matchmaker agent is composed of four types of entities. These are *task modules*, *module containers*, *module controllers*, and *request handlers*. Task modules or simply modules are the primitive entities that are responsible for performing specific data processing tasks.

There are three types of modules: *knowledge extractor modules*, *data storage modules* and *matchmaker modules*.

Knowledge extractor modules: These are responsible for extracting the information required by matchmaking approaches from the service and request descriptions. For instance, in our approach we need Promela models of services to use Spin model checker for matchmaking. However, service providers register their services through OWL-S, which does not include the Promela model of the service. Therefore, we have a knowledge extractor module in the framework to generate the Promela model of the service by using the information extracted from the OWL-S service model. If a service provider decides to describe its services in another language, we can simply add a new module to do the same extraction task for the new language, which makes the matchmaker agent capable of working with the new language without any modification to the rest of the system.

Data storage modules: These are responsible from the storage of the information that is extracted by the knowledge extractor modules from service descriptions. Each matchmaking algorithm may require different information to work. For instance, our approach require the model of the service in Promela and other approach may require the input-output interface of the service. By using multiple data storage modules we separate information required for each matchmaking approach, which allows flexible addition or removal of new matchmaking approaches to the matchmaker agent.

Matchmaker modules: These are the implementations of the actual matchmaking algorithms.

Module containers and controllers: Containers provide an abstraction layer to collect similar modules together (knowledge extractor, data storage and matchmaker) and controllers provide an interface to communicate with the modules of the corresponding container. Only controllers know that which modules are contained by the corresponding container, therefore the controller is responsible to find the appropriate module for a specific task.

Request handlers: These are responsible to handle requests from the service providers and service requesters as well as manage the data flow between the modules through corresponding controllers. There are two different handlers. *Service consumer request handler* is responsible for handling service matchmaking requests from service consumers. It collects information from knowledge extractor and data storage controllers and feed the matchmaker controller with this information and sends the matchmaking results back to the service consumer. *Service provider request handler* is responsible for handling service registry requests from service providers. It requests all the required service information for an incoming registry request from the knowledge extractor controller and forwards this information to the data storage controller to finalize the registration.

6. CASE STUDY

We evaluate our approach in a case study. In this case study there is a consumer, who is looking for a service to buy books. Her primary property states that she requires a service where the connection is secured by https protocol when she makes the payment by her credit card and otherwise she will not do any payment. We represent this

property with the LTL formula $\mathsf{G}((s_r \rightarrow (s_r \mathsf{U} p_r)) \vee \neg p_r)$, where s_r means `secure_https_connection` and p_r means `pay_by_creditcard`. Her secondary property states that she prefers to receive the book by cargo before she makes any payment by her credit card. We represent this property with the LTL formula $\mathsf{G}(d_r \rightarrow (d_r \mathsf{U} p_r))$, where d_r means `delivery_by_cargo` and p_r means `pay_by_creditcard`.

There are 16 registered book selling services available in the matchmaker agent. These services differ in how well they satisfy the request. Broadly, there are five categories of services. The first category contains services that exactly match the request. There is one such service. In the second category, there are services that satisfy only one property exactly while they satisfy the other property only partially or not satisfy at all. There are six of these services. In the third category, there are services that satisfy both types of properties partially. There are four of these services. In the fourth category, we have services that partially satisfy only one of the properties. There are four of these services. Finally, there is one service in category five that does not satisfy any property, neither fully nor partially.

Table 2 summarizes the results of our approach for this matchmaking request. The table lists all the services with their related properties to the request. If a property is exactly satisfied, we write `satisfied` in the corresponding field and if the property is not satisfied even when we use semantics, we write `not satisfied`. Recall that if the service does not satisfy the original property, our approach generates alternative properties and finds out which of these are satisfied by the service. Further, it computes a matching degree among the possible alternative and picks the one with the highest degree. In Table 2, when a property is not satisfied by a service, we write the closest alternative property that is satisfied by the service. Facts s_a, p_a and d_a are the processes provided by the services that are semantically related in our ontology to the processes s_r, p_r and d_r in the request, respectively.

Let us examine the results. Table 2 shows how our approach successfully generates alternative properties from the originally requested property and identify partially matching services. Without generating alternative properties, our approach can only match four services (13, 14, 15 and 16), which satisfy the primary property exactly. By generating the alternative properties we identify eight more services (04, 05, 06, 07, 08, 09, 11 and 12) that partially satisfy consumer requirements and might be useful for the consumer.

We will investigate S12 in detail to show the advantage of identifying these partially matching alternative services. S12 satisfies the secondary property of the request exactly, however it does not satisfy the primary property and therefore it is not identified as a matching service at the beginning. To check whether this service partially satisfies the request, we generate three alternative properties from the original primary property using the semantic relations between s_r and s_a and between p_r and p_a. The three generated properties are; (1) $\mathsf{G}((s_r \rightarrow (s_r \mathsf{U} p_a)) \vee \neg p_a)$ in which we use payment process p_a instead of p_r, (2) $\mathsf{G}((s_a \rightarrow (s_a \mathsf{U} p_r)) \vee \neg p_r)$ in which we use security process s_a instead of s_r and (3) $\mathsf{G}((s_a \rightarrow (s_a \mathsf{U} p_a)) \vee \neg p_a)$ where we use s_a and p_a instead of s_r and p_r, respectively. Then, we use the model checker to find which of these properties are satisfied by the service and we determine that S12 satisfies only the first alternative property, where p_r process is replaced by p_a. This alternative property shows us S12 can satisfy the request by replacing the payment process. Assume that at some point the only available service in the environment is S12 and the rest of the 15 services are not available for some reason. In this situation, rather than returning no result in reply to the request, showing the partial matching service S12 to the consumer by indicating the difference in the payment process is more informative for the service consumer and allows she to make a more precise decision.

Table 2: List of the 16 potential services. s_r: secure https connection, p_r: payment by credit card, d_r: delivery by cargo. s_a, p_a and d_a are semantically related processes.

S	Primary Property	Secondary Property
S01	not satisfied	not satisfied
S02	not satisfied	$\mathsf{G}(d_a \rightarrow (d_a \mathsf{U} p_a))$
S03	not satisfied	$\mathsf{G}(d_r \rightarrow (d_r \mathsf{U} p_a))$
S04	$\mathsf{G}((s_a \rightarrow (s_a \mathsf{U} p_a)) \vee \neg p_a)$	not satisfied
S05	$\mathsf{G}((s_r \rightarrow (s_r \mathsf{U} p_a)) \vee \neg p_a)$	not satisfied
S06	$\mathsf{G}((s_a \rightarrow (s_a \mathsf{U} p_a)) \vee \neg p_a)$	$\mathsf{G}(d_a \rightarrow (d_a \mathsf{U} p_a))$
S07	$\mathsf{G}((s_a \rightarrow (s_a \mathsf{U} p_a)) \vee \neg p_a)$	$\mathsf{G}(d_a \rightarrow (d_a \mathsf{U} p_r))$
S08	$\mathsf{G}((s_r \rightarrow (s_r \mathsf{U} p_a)) \vee \neg p_a)$	$\mathsf{G}(d_a \rightarrow (d_a \mathsf{U} p_a))$
S09	$\mathsf{G}((s_a \rightarrow (s_a \mathsf{U} p_r)) \vee \neg p_r)$	$\mathsf{G}(d_a \rightarrow (d_a \mathsf{U} p_r))$
S10	not satisfied	satisfied
S11	$\mathsf{G}((s_a \rightarrow (s_a \mathsf{U} p_a)) \vee \neg p_a)$	satisfied
S12	$\mathsf{G}((s_r \rightarrow (s_r \mathsf{U} p_a)) \vee \neg p_a)$	satisfied
S13	satisfied	not satisfied
S14	satisfied	$\mathsf{G}(d_a \rightarrow (d_a \mathsf{U} p_a))$
S15	satisfied	$\mathsf{G}(d_a \rightarrow (d_a \mathsf{U} p_r))$
S16	satisfied	satisfied

As we explained, finding partially matching alternative services is useful. However, these alternative services must be scored according to their relevance to the original request, to inform the consumer about the possible differences. For this purpose, our approach assigns degree of match values to services as explained in Section 4. To evaluate this approach we conduct the following experiment, where we compute the degree of match values of the services in Table 2. For simplicity, in our experiment we take the similarity value between all semantically related processes as 0.5. This means, similarity between process s_r and s_a, p_r and p_a, and d_r and d_a are all equal to 0.5. According to this setting, the similarities between the original property requested by the consumer and the alternative properties satisfied by the services are computed by taking the average of the similarities between the individual processes in the properties. For instance, in the original primary property, there are s_r and p_r processes. In the alternative primary property of S12 s_r process is identical with the required property but p_a is used instead of p_r. The similarity of the properties will be equal to 0.75, which is the average of 1.0 from s_r and 0.5 from p_a.

Considering this trivial weighting schema, our our matchmaking approach assigns a matching degree to each service. Important findings among these matches are the following: S16 is assigned a matching degree of 1.0, i.e., it is successfully identified as an exact match by the matchmaker agent. Additionally, the matchmaker agent successfully finds the two good alternative services S15 and S12, which are equally assigned a matching degree of 0.875. Both of these services fall into the category, where one requirement is fully satisfied but the second requirement is only partially satisfied.

The degree of match values associated with the partially matching services show that the matchmaker favors the services that are more similar to the original request, which use fewer number of alternative processes. The matchmaker agent also assigns a matching degree of 0 to services S01, S02, S03 and S10, since neither of them satisfy the primary property or any alternative similar property.

7. DISCUSSION

We propose a novel agent-based service matchmaking approach that captures the semantics of both the services and the requests. We use Promela and LTL formulae to model service descriptions and consumer requests respectively and we use Spin as model checker. Our approach can find exact matches based on semantics of consumer requests and existing services. If the consumer requests cannot be satisfied with existing services, our approach computes the closest service requests that can be satisfied by the available services using a process ontology, which models the atomic processes contained by the services as semantic concepts and defines semantic relations between these processes. This enables our approach to support also partial matches and associate match degrees to these partially matching services. We also propose a general matchmaking framework for our approach where it can be used in conjunction with other approaches.

In the literature there are several approaches based on input-output matching [12, 14]. Most recently Klusch *et al.* [11] extend the input-output matching approach by combining syntactic matching techniques from information retrieval with semantics in order to improve matchmaking performance. Although this approach provides a mechanism to rank services according to their relevance to the request it still suffers from low precision. Additionally as stated in [5] syntax based information retrieval techniques are not efficient for Web service matching due to insufficient amount of textual data in Web service descriptions. There are other approaches that try to capture the semantics of the service for better matchmaking. Klein and Bernstein [10] propose an indexing mechanism to create a hierarchical ontology of process models and develop a query language to perform matching on the created ontology. Wombacher *et al.* [18] propose a process based matchmaking approach, where process are modeled as finite state machines and matchmaking is done through operations such as disjunction, conjunction and intersection. However this approach based on syntactic information and does not consider any semantic knowledge. Additionally it does not support partial matches. Brogi and Corfini [2] present a matchmaking approach similar to the one we present in this paper. Instead of LTL and model checking they use Petri Nets to model services and apply Petri Net control flow verification for matchmaking. However, different than ours, the main aim in this approach is to find possible compositions of available services to satisfy service requests. Model checking and temporal logic is used for the composition of Web services [7, 13, 17]. However, to the best of our knowledge, our approach is the first use of these methods for matchmaking of web services.

8. REFERENCES

[1] A. Ankolekar, M. Paolucci, and K. Sycara. Spinning the OWL-S Process Model: Towards the verification of the OWL-S Process Models. In *Proc. of ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, 2004.

[2] A. Brogi and S. Corfini. Behaviour-aware discovery of Web service compositions. *International Journal of Web Services Research*, 4(3):1–25, 2007.

[3] David Martin et. al. OWL-S: Semantic Markup for Web Services, 2002.

[4] K. Decker, K. Sycara, and M. Williamson. Middle-agents for the Internet. In *Proc. of Int. Joint Conf. on Artificial Intelligence*, pages 578–583, 1997.

[5] X. Dong, A. Y. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Simlarity Search for Web Services. In *Proc. of 13th Int. Conf. on Very Large Data Bases*, pages 372–383, 2004.

[6] E. A. Emerson. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, chapter Temporal and Modal Logic, pages 997–1072. Elsevier and MIT Press, 1990.

[7] X. Fu, T. Bultan, and J. Su. Formal verification of e-services and workflows. In *Int. Workshop on Web Services, E-Business, and the Semantic Web*, 2002.

[8] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[9] I. Horrocks et. al. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, 2004.

[10] M. Klein and A. Bernstein. Toward High-Precision Service Retrieval. *IEEE Internet Computing*, 8(1):30–36, 2004.

[11] M. Klusch, B. Fries, and K. Sycara. Automated Semantic Web Service Discovery With OWLS-MX. In *Proceedings of 5th Internatioanl Joint Conference on Autonomous Agents and Multiagent Systems*, pages 915–922, 2006.

[12] L. Li and I. Horrocks. A Software Framework for Matchmaking Based on Semantic Web Technology. In *Proc. of 12th Int. Conf. on World Wide Web*, pages 331–339, 2003.

[13] S. Narayanan and S. A. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. of 11th Int. Conf. on World Wide Web*, pages 77–88, 2002.

[14] M. Paolucci, T. Kawamura, T. R. Payne, and K. P. Sycara. Semantic Matching of Web Services Capabilities. In *Proc. of 1st Int. Semantic Web Conference*, pages 333–347, 2002.

[15] P. Patel-Schneider. Semantic Web Rule Language First-Order Logic (SWRL FOL), 2005.

[16] P. Resnik. Using Information Content to Evaluate Semantic Similarity in a Taxonomy. In *Proc. of 14th Int. Joint Conf. on Artificial Intelligence*, volume 1, pages 448–453, 1995.

[17] M. P. Singh. Distributed Enactment of Multiagent Workflows: Temporal Logic for Web Service Composition. In *Proc. of the 2nd Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 907–914, 2003.

[18] A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for Business Processes Based on Conjunctive Finite State Automata. *International Journal of Business Process Integration and Management*, 1(1):3–11, 2005.