Abstract

The aim of service discovery is to find services that satisfy user requests in a precise and effective manner. An important aspect of service discovery is service matchmaking, which constitutes the mechanism to map appropriate services to requests. Current service matchmaking approaches mostly use the knowledge about the interface descriptions of services. However, these approaches suffer from lack of precision since they do not consider the internal processes of services.

This paper proposes a novel service matchmaking approach that uses the internal process models of services as primary source of knowledge. To reason about the internal process models and to identify matching services to requests, we use model checking as a reasoning mechanism. In order to facilitate partial matches, we use ontologies and relaxation techniques to generate alternative requests. Hence, even when a request cannot be satisfied by a service, our approach can identify which similar requests are satisfied by the service. This important information can enable better service selection for the service consumers. We also provide a guideline to illustrate how our proposed matchmaking approach can be realized using recent technologies from Web services and formal verification domains in a real world setting.

Keywords: service oriented computing, service discovery, semantic matchmaking, model checking

1. Introduction

Service-Oriented Computing (SOC) [32, 37] is a distributed computing paradigm that makes use of services as the fundamental building block of distributed systems. A service is a loosely coupled business process that provides some reusable functional capabilities over a network. In SOC paradigm, services can be developed by independent parties and large scale applications can be built by composing and orchestrating these independent services in heterogeneous distributed environments.

One of the fundamental challenges of SOC paradigm is service discovery. The aim of service discovery is to find available services that can satisfy particular needs of service consumers. Note that service consumers need not to be humans. A software can also act as a service consumer to a service provider. Thus, on one side, service consumers should be able to identify the service they are interested in as precisely as possible. On the other hand, the service providers should be able to explain what their services fulfill. After this is done, the service matchmaking comes into play. Put simply, the service matchmaking approach is the method that is used to determine which of the available services satisfy the requirements of the service consumer considering the request of the service consumer and the knowledge about the capabilities of available services.

The open nature of SOC causes two major challenges considering the service matchmaking approaches [2, 37, 39]. The first challenge arises due to development of services by independent parties. Since each developer follows their own methodology to create services, there are no common definitions on the provided capabilities of services. It is quite common to have various services that provide similar capabilities but have different definitions. A service matchmaking approach should be able to identify such situations. Second challenge arises due to loosely coupled structure of services. In SOC, service providers do not have prior knowledge about the requirements of potential service consumers. Hence, services are mostly developed in a generic manner such that they can be adopted into various situations. Therefore, in SOC, it is possible that there are no services that satisfy a consumer’s request. However, in most such cases there are still services in the environment that can at least partially satisfy the request of the service consumer. A desired service matchmaking approach should handle such situations and find services that can partially satisfy the request of the service consumers. In such a case the service matchmaking approach should also provide an ordering of services that shows the relevance between the partially matching services and the request of the service consumers as a guideline to help choose an appropriate service.

Web services (WSs) [2, 7] are the most successful realization of the SOC paradigm. The current service discovery mechanism of WSs is based on WSDL [10] and UDDI [12]. WSDL is an XML based language to describe properties of services. UDDI is a registry where service providers can advertise their services and service consumers can search for services. Although UDDI in

Email addresses: akin.gunay@boun.edu.tr (Akin Günay), pinar.yolum@boun.edu.tr (Pınar Yolum)

Preprint submitted to Elsevier May 25, 2013
combination with WSDL provides a basic mechanism for service discovery, it lacks support for automated discovery. Ideally, SOC needs an automated service discovery approach that will take in a service request and find all the services that can fully or partially satisfy the request.

There are various proposals to automate service matchmaking [1, 5, 6, 8, 9, 11, 14, 13, 24, 25, 41]. One of the most widely studied approach is based on the idea of interface matching [18, 39] similar in principle to matching of software components from the field of software engineering [43].

This approach is based on defining the requested service through its expected input-output interface and comparing this expected interface with the input-output interfaces of available services to find matching services. A major advantage of this approach is its simplicity. However, not surprisingly, this approach fails to be successful in many situations since it relies only on syntactic knowledge: Two services with identical input-output interfaces may be performing drastically different computations. Or, similarly, two services with different interfaces may be performing similar computations. Since syntactic knowledge does not capture any meaning, matching solely based on syntactic knowledge is not enough.

In order to overcome this problem, interface matching approach is further improved by the WS community from the syntactic level to semantic level by associating the elements of the interfaces with concepts from ontologies. This semantic approach improves the quality of matchmaking results since it does not depend on the syntactic keywords but semantic concepts [26, 31]. Hence, even if the input and outputs are not identical, one can reason about their semantic similarity and reach a conclusion based on that. Although introduction of semantic knowledge improves the performance of interface matchmaking, since it is not possible to determine what a service does only by considering its interface, this approach may still mistakenly match many services to a service request [15].

Consider the following example: A user is looking for a service that can be used to reserve a hotel and to pay for the reservation. However, the user wants to get a voucher that guarantees the reservation, before she makes the payment. Using interface matching, it is not possible to capture such a requirement of the user, since it is not possible to represent this requirement which involves a temporal fact using only the input-output interface of a service. The best one can say is that the service takes as input a reservation date as well as a payment and returns a reservation voucher. Note that, we cannot enforce any constraints, such as the reservation voucher should be for the requested dates or that payment be made after the reservation voucher. Hence, other services that provide such variations would be mistakenly matched for this request.

A possible improvement to overcome such a problem is the use of the information about precondition and effects. [23, 27, 40] When this is the case, the service explicitly defines what the precondition and the effect of the service is. Continuing from the above example, the service may state its precondition as there is a valid payment and valid reservation dates and the effect will be that a hotel voucher will be issued for those particular dates. Hence, precondition-effect matching goes beyond simple interface matching in at least capturing some constraints about the entry and exit points of a service. However, we still cannot express internal constraints about the service, such as the payment will be done only after the voucher is produced. Any service that produces the voucher after the payment will also satisfy our defined effect above.

To go beyond interface or precondition-effect matching, it is necessary to understand what a service consumer expects from a service as well as what a service can accomplish for a consumer. Only by understanding and using these information, we can achieve better matchmaking performance. This understanding can only be possible if we have knowledge about the business processes of services. Our use of business process is rather general, but we assume that the business process of a service defines characteristics of the process that are relevant for the business parties. A service’s process need not specify all the internal properties of the service. Hence, the service is autonomous in the sense that it can decide the relevant details for the consumers and only publish that information inside the business process. From a consumer’s point of view, it is important to know how a service uses the given input to produce the output, whether any constraints are being enforced, and so on.

In this paper, we argue that to achieve well-targeted matchmaking between service requests and service offerings, service providers need to specify their business processes and service consumers need to specify their requests with well-defined semantics. Business process of a service defines what the service does and how this functionality is achieved by the service in detail. Similarly, a service request defines the expectations of the customer in terms of process details. Since in our approach we capture the process details of services, we need a mechanism that can reason on these processes. For this purpose, we use model checking techniques in our approach. The main idea of our approach is to transform the service matchmaking into model checking in which we represent services as system models and service request as set of formal properties.

By using model checking as a reasoning mechanism, we determine whether a service can satisfy required properties of a service consumer and accordingly we find matching services to service consumer requests. We apply this idea for developing a family of methods and algorithms for matchmaking. Our first method provides only exact matching capabilities. However, in many settings, finding partially matching services is also important. For this reason, we develop two more methods. Our second method benefits from using semantic concepts. In this method, if we cannot find an exact match, we generate similar requests to the original request using an ontology and the
properties involved in the original request. After that we compute a similarity value between the original and generated requests using a semantic similarity metric to capture the degree of match. Our third method is based on the idea of relaxing service consumer requests. In this method, similar to our other method to handle partial matches, we generate similar requests, when there is no exact matching service for a request. However, compared to the previous case, this time we revise the temporal constraints of the request instead of the semantic properties.

There are two major advantages of our approach compared to other approaches such as the input-output and precondition-effect matching. The first advantage is that it captures the business model of a service and hence provides more precise details to the user about the workings of the services. The second advantage of our approach is the involvement of the temporal requirements of the users in the service request. This enables the user to phrase precise requirements about the service. Such requirements of the users are not involved with either of the other approaches. As a result, we can perform matchmaking more effectively and achieve precise results.

The rest of this paper is organized as follows. Section 2 provides the technical background that is used in the rest of the paper. Section 3 defines our model checking based matchmaking approach and our first method. Section 4 explains our approach and algorithms to handle partial matches. Section 5 shows how our approach can be applied in a real world setting. Section 6 studies our methods in a case study. Finally, Section 7 surveys the related work and discusses pros and cons of our family of algorithms for service matchmaking.

2. Technical Background

A service is an abstract concept that may correspond to any computational process. Each service takes a set of arguments as input, performs some computations according to the content of the inputs and produces a set of outputs as the result of the computational process [32, 37].

Definition 1. (Service) A service is a computational process that provides a functional capability over a network. It takes a set of inputs and produces a set of outputs.

Definition 2. (Service provider) A service provider is a software agent that provides one or more services.

Definition 3. (Service consumer) A service consumer is an agent that uses services provided by service providers in order to achieve its own goals.

A service definition defines the functional capabilities of a service and how these capabilities can be used. A service definition involves five kinds of information. These are the service input, the service output, the service preconditions, the service effects and the service process model.

A service input defines the inputs required by the service in order to properly execute the service. A service output defines the results of the service execution. A service precondition defines logical conditions required by the service to be executed. An effect defines the resulting logical effects of the service execution. A process model defines the internal computational process of the service that describes how the service produces its output using the given input (we discuss details of process models in Section 2.2).

Definition 4. (Service definition) A service definition SD = ⟨I, O, P, E, M⟩ is a five-tuple where I is the typed input set, O is the typed output set, P is a set of preconditions represented in a formal language, E is a set of effects in a represented in a formal language and M is the process model of the service.

A service request defines the requirements of a service consumer from a service. A service request consists of a set of formal properties where each property represents a specific functional requirement of the service consumer.

Definition 5. (Service property) A service property p is a formula in a formal language that represents a functional requirement expected from a service.

A service property tells us what a service can do. Given a service property, one can check whether a given service can satisfy the service property.

Definition 6. (Service request) A service request R = {p₁, ..., pₙ} is a set of service properties.

Definition 7. (Exact Property Satisfaction) A service s satisfies a property p only if after the enactment of service s, the property p is true.

2.1. Service Matchmaking

Service matchmaking is the process of finding one or more services that satisfy a given service request. Service matchmaking can be divided into two types: exact and partial service matchmaking. In the case of exact service matchmaking, in order to be selected as a match for the request, a service must exactly satisfy all the individual properties in the service request.

Definition 8. (Exact Service Matchmaking) Given a set of services S and a service request R, exact service matchmaking is the process of finding a set of services T, such that T ⊆ S and each service s ∈ T satisfies every property p ∈ R.

In the case of partial service matchmaking, in order to be selected as a match, it is enough for a service to partially satisfy a subset of the properties in the service request. Note that an alternative definition could have required a service to satisfy all properties at least partially to be qualified as a matching service. While that definition
can also be adopted, without loss of generality we choose an even weaker definition and state that it is enough for the service to satisfy some of the properties to be qualified as a partial match.

**Definition 9. (Partial Property Satisfaction)** A service $s$ satisfies a property $p$ partially if there is another property $p'$ that can be exactly satisfied by the service $s$ and there is a semantic similarity relation between the properties $p$ and $p'$.

**Definition 10. (Partial Service Matchmaking)** Given a set of services $S$ and a service request $R$, partial service matchmaking is the process of finding a set of services $T$, such that $T \subseteq S$ and each service $s \in T$ partially satisfies at least one property $p \in R$.

### 2.2. Model Checking and Linear Temporal Logic

Our service matchmaking approach is based on model checking. Model checking is a formal method to verify that a system can satisfy an intended property, which is represented in a formal language such as linear temporal logic (LTL). In this section we present a brief overview of LTL and model checking for completeness. The reader can refer to [16, 22] for further details.

We can investigate the model checking process in three steps. The first step is modeling the system that is under consideration. A system model is a transition system that consists of a set of states, a set of propositions that are associated with each state and a set of transitions that define how the system moves from one state to another.

**Definition 11.** A transition system $T$ is a three tuple $< S, \rightarrow, L >$, where $S$ is the set of states connected by $\rightarrow$, a binary transition relation defined over $S$ and a labeling function of the form $L : S \rightarrow P(Atoms)$.

The labeling function $L(s)$ associates the particular state $s$ with the set of propositions form the power set $P(Atoms)$ that are true in this particular state.

**Definition 12.** A path $\pi$ of the transition system $T$ is an infinite sequence of states $s_1,s_2,s_3,...$, in $S$ such that, for each $i \geq 1$, $s_i \rightarrow s_{i+1}$.

The second step in the model checking process is representing the property that we intend to verify in a formal language such as LTL. LTL is a temporal logic, where the future is seen as a sequence of states or simply as a path. LTL formulae are build 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 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. Formal definition of the semantics of LTL is as follows:

**Definition 13.** Let $T = <S,\rightarrow,L>$ is a transition system and $\pi = s_1 \rightarrow ...$ be a path in $T$. Then satisfaction relation $\models$ that determines whether the path $\pi$ satisfies the LTL formula is defined as follows:

- $\pi \models \top$
- $\pi \not\models \bot$
- $\pi \models p$ iff $p \in L(s_1)$
- $\pi \models \neg \phi$ iff $\pi \not\models \phi$
- $\pi \models \phi_1 \land \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
- $\pi \models \phi_1 \lor \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$
- $\pi \models \phi_1 \rightarrow \phi_2$ iff $\pi \models \phi_2$ whenever $\pi \models \phi_1$
- $\pi \models X \phi$ iff $\pi^2 \models \phi$
- $\pi \models G \phi$ iff, for all $i \geq 1$, $\pi^i \models \phi$
- $\pi \models F \phi$ iff there is some $i \geq 1$ such that $\pi^i \models \phi$
- $\pi \models \phi U \psi$ iff there is some $i \geq 1$ such that $\pi^i \models \psi$ and for all $j = 1,...,i - 1$ we have $\pi^j \models \phi$

In the above clauses we use the $\pi^i$ notation to identify the states in the transition system. $\pi^1$ represents the first state of the transition system and $\pi^i$ represents the $i^{th}$ state.

In the last step of the model checking process, the property that we describe as an LTL formula is checked against the system model that we define as a transition system. For this purpose a model checker exhaustively checks the LTL property on all possible executions of the transition system. If for all possible executions the LTL property is satisfied by the transition system, then we conclude that the system satisfies the LTL property.

**Definition 14.** $M = (S,\rightarrow,L)$ is a model, $s \in S$ and $\phi$ is an LTL formula. We say $M,s \models \phi$ if, for every execution path $\pi$ of $M$ starting at $s$, we have $\pi \models \phi$.

### 3. Service Matchmaking as Model Checking

The main idea of our approach is to transform the service matchmaking process into a model checking process and use existing model checking techniques for service matchmaking. Our idea is based on the similarities of these two processes. The aim of service matchmaking is to determine whether a service can satisfy a set of properties given as a service request. Similarly, the aim of
model checking is to determine whether a system model can satisfy a set of given formal properties. Following this similarity, our claim is that if we represent a service as a system model and a service request as a set of formal properties, then we can use model checking in order to determine whether these properties can be satisfied by the system model, which shows that the service satisfies the service request. Accordingly, if we determine that all of the checked properties are satisfied by the system, we can conclude that the service matches the service request.

As we state in our claim, our approach requires us to represent a service as a system model and a service request as a set of formal properties. In the rest of this section, first we explain how these representations can be achieved. After that, we present our first algorithm for service matchmaking that uses these representations and model checking as a basis.

3.1. Representing Services as Transition Systems

In our approach we need to represent a service as a system model in order to be able to use model checking techniques for service matchmaking. Hence, we need a mechanism to map a service definition into a transition system. Following the Definitions 4 and 11, this process is straightforward. To represent a service as a transition system, we associate each action taken by either the service provider or the service consumer with an event and results of these actions with changes of a defined set of properties. Specifically, in our approach we represent each action taken either by the service provider or the service consumer as a single transition in the transition system. Each such transition starts from a state and reaches another state. The state from which the transition starts represents the status of the properties that are required to initiate this transition. On the other hand, the state at which the transition ends represents the state changes of the properties after performing the corresponding transition.

In Figure 1 we present an example transition system that models an e-commerce service to make a hotel reservation. Using this service, a service consumer can first make a request for a hotel in some location for some dates and accordingly can reserve a room and pay for this room. At the end of this service, the service provider delivers a voucher that confirms the reservation and payment. This transition system consists of four transitions and four propositions. In the figure, each circle represents a state with the current status of each proposition and initially the state of each proposition is set as false, which means that it does not hold. States with double circles are final states. In the figure each arc represents a transition. Note that, we do not explicitly specify the roles of agents and parameters of transitions for simplicity, since these roles and parameters are clear from the context. The first transition is the hotel-request transition. As the result of this transition the service consumer receives an offer for a hotel room and its price for the given dates. At the same time through this transition the service provider commits to reserve a room from the offered price if the service consumer confirms the room in the future. As the result of this transition the state of the proposition hotelReserved is set to true, which means that the hotel and the price are agreed by both parties. Note that this is a final state, since if the service consumer is not happy with the offered prices she can terminate the service. The second transition is the hotel-confirm transition. As the result of this transition the service consumer accepts the offered price for the offered room and confirms the reservation for the room over this price. This transition changes the state of the proposition hotelConfirmed to true in order to reflect this situation. Next transition is the make-payment transition. As the result of this transition the service consumer makes the payment for the reserved room and accordingly the state of the hotelPaid proposition changes to true. The last transition is the hotel-voucher transition. This transition generates a voucher that involves the hotel and payment information. As the result of this transition the state of the proposition hotelVouchered changes to true. This state is a final state in which all properties hold.

3.2. Representing Service Requests as Set of Properties

In order to use model checking for service matchmaking we need to represent a service request as a set of formal properties. In order to achieve this we represent each individual property in the service request as an LTL formula. LTL is a suitable formalism since it allows us to represent temporal requirements of service consumer in the service request. In the rest of this section we present several example properties from the e-commerce domain that might be requested by a service consumer.

Example 1. Generation of the voucher 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 voucher for her reservation is made before the payment for the reservation. This fact is represented by the following LTL formula. In the formula the proposition p represents the fact that the payment is made and t represents the fact that the voucher is generated. Formally, G(p → t).

Example 2. Guarantee voucher or refund after payment: The service consumer requests a service that guarantees to generate a voucher for the reservation after the payment is done. In the exception case, if the payment is done but there is no voucher for the reservation (due to the cancellation of the reservation by the consumer or a problem faced by the provider, which prevents the provider from generating the voucher for the reservation), then the payment must be refunded by the service provider. Following LTL formula corresponds to this situation, where the proposition p represents the fact that the payment is made, t represents the fact that the voucher for the reservation is generated and r represents the fact that a refund is made for the payment. Formally, G(p → ((pU t) ∨ (pU r))).
Example 3. 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 stays in this secure state until the payment operation is completed. Otherwise the payment operation is not performed. In the formula, the proposition $s$ represents the fact that the communication is secured and $t$ represents the fact that the payment is made. Formally, $G((s \rightarrow (sUp)) \lor \neg p)$.

Example 4. 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 generates the voucher for the reservation (successful transaction) or in a state, where there is no payment and no voucher generation action is performed by any of the participants (canceled transaction). In the formula, the proposition $p$ represents the fact that the payment is made and $t$ represents the fact that the voucher is generated. Formally, $F((p \land t) \lor (\neg p \land \neg t))$.

Example 5. Secure connection for all the transactions: A more suspicious consumer who is concerned more about her privacy may not feel comfortable with a service that provides security only for payment but may require a service, where the whole connection is secured. This property is represented by the following formula where proposition $s$ represents the fact that the communication is secure. Formally, $G(s)$.

The above examples show that many real world properties of services can be formalized using LTL. Service consumers can combine any such properties to create a service request. Hence, using this formalism it is possible to represent both simple and complex service request.

### 3.3. Model Checking based Service Matchmaking Algorithm

The above representations allow us to develop an algorithm that uses model checking for service matchmaking. We present this algorithm in Algorithm 1. The algorithm takes a set of available service models represented as transition systems ($S$) and a set of LTL properties ($R$), which represent the service consumer’s request as input. It returns a set of services ($M$) that exactly matches the service consumer’s request. The outer loop of the algorithm iterates over all available services (lines 2-13). The inner loop of the algorithm checks each requested LTL property against the service in turn using model checking (lines 4-8). For efficiency if a service fails to satisfy a property in the requested set of properties, the algorithm immediately considers this service as a non-matching service and does not check the rest of the properties that are not checked for this service yet (line 7). If a service can satisfy all the properties in the request, then the algorithm selects it as a match for the request and adds it to the result set of matching services (lines 10-11). After all services are checked the algorithm returns the set of matching services.

### 4. Supporting Partial Matches

Algorithm 1 serves as our initial starting point for the family of algorithms we are developing. It captures our translation of service matchmaking into model checking well. However, it does not support partial matches. That is, according to our approach a service matches a service request only if it can exactly satisfy all the individual properties given in the service request. However, as we state in Section 1, a matchmaking approach should also support partial matches. In order to achieve this, in the rest of this section we propose two methods that builds on top of Algorithm 1. Both of our methods are based on the definitions of partial property satisfaction and partial service matchmaking that we state in Section 2.1.

The definition of partial property satisfaction (Definition 9) states that a property $p$ is partially satisfied by a service when the service can exactly satisfy another property $p'$ and there is a similarity relationship between $p$ and $p'$. The algorithm will consider the service as a match for the request. The algorithm will then add it to the result set of matching services.
Additionally, the definition of partial service matchmaking (Definition 10) states that if a service can satisfy at least one such similar property, then the service is a partial match to the request. Using these definitions, when a service cannot satisfy a requested property, both of our methods generate a set of new properties that are similar to the requested (but not satisfied) property and check whether the service can satisfy one of these generated similar properties to capture a partial match for the request. The two methods mainly differ in the way they generate similar properties.

Since the methods deal with partially matching services, we need some measurement to indicate the difference between the relevance of matching services. For this purpose in our methods, we associate a degree of match value that has a range $[0, 1]$ between each service and each requested property. If a service can satisfy a requested property as it is, then we associate 1 as the degree of match value between the service and the property, which indicates an exact match. On the other hand, when a service cannot satisfy the requested property but a similar property, first we compute a similarity value between the requested property and the similar (satisfied) property using a similarity metric. Then we assign this similarity value as the degree of match value between the property and the service. This similarity value is always smaller than 1, hence it always indicates that the partial match is less desirable than the full match. If a service cannot satisfy even a similar property to the requested property, then we assign 0 as the degree of match value between the service and the property. At the end of the matchmaking process we combine individual degree of match values between services and properties and obtain an overall degree of match value for each service and accordingly create a relevance order between services from best matching service to worst matching service using these overall degree of match values.

Although both of our methods for partial service matching are based on similar property generation, we use different techniques in each method to generate similar properties and to compute similarities between the originally requested and generated similar properties. In the first case, we start from an existing LTL formula and associate propositions of the LTL formula with concepts from ontologies. Without changing the temporal structure of the formula, we generate similar properties using only the ontology. Hence, similar concepts from the ontology are found and replaced in the property. In the second case, we again start from the requested LTL formula. However, this time we do not change the propositions but modify the temporal structure of the formula. For this purpose we define a set of relaxation relations for LTL formula based on the entailment relation of LTL operators.

Before explaining the details of our methods for partial service matching, we first provide an extended version of our model checking based matchmaking algorithm that we present in Algorithm 1. This extended algorithm that we present in Algorithm 2 is based on two generic functions $\text{genSimPropSet}()$ and $\text{compPropSim}()$. The first function defines how similar properties are generated from the originally requested property. The second function defines how the similarity is computed between a similar and originally requested property. Both of our methods use this algorithm as a base with different implementations of these two generic functions. Hence this algorithm is not restricted only to our two proposed methods and any method that appropriately implements these two functions can use this algorithm for model checking based service matchmaking.

```plaintext
Algorithm 2 Extended service matchmaking algorithm to capture partial matches

Require: ServiceSet $S$
Require: Request $R$
1: $M := \emptyset$
2: for each $s$ in $S$ do
3:   $SIM := \emptyset$
4:   for each $r$ in $R$ do
5:     if $s \models r$ then
6:       $SIM := SIM \cup \{(1.0)\}$
7:     else
8:       $simPropSet := genSimPropSet(s, r)$
9:       $highSim := 0.0$
10:      for each $simProp$ in $simPropSet$ do
11:        if $s \models simProp$ then
12:          $sim := compPropSim(r, simProp)$
13:          if $sim > highSim$ then
14:            $highSim := sim$
15:          end if
16:        end if
17:      end for
18:      $SIM := SIM \cup \{highSim\}$
19:    end if
20:  end for
21: $degreeOfMatch := f(SIM)$
22: if $degreeOfMatch > 0.0$ then
23:   $M := M \cup <s, degreeOfMatch>$
24: end if
25: end for
26: return $M$
```

As in Algorithm 1 the extended algorithm takes the set of available service models ($S$) and a set of properties ($R$) that represent the service consumers request as input and returns a set of services ($M$) that either exactly or partially satisfies the given service request. Different than Algorithm 1, the returned set of services are also associated with degree of match values that orders matching services according to their relevance to the service request.

The outer loop of the algorithm iterates over the service set (lines 2-25) and the inner loop iterates over each requested property (lines 4-20). If the property is satisfied by the service as it is (lines 5-6) then value 1 is stored as degree of match value (in $SIM$), which shows an ex-
act match. If the property is not satisfied as it is (lines 7-19) then a set of similar properties are generated by the \textit{genSimPropSet()} function considering the originally requested property. After that, each generated similar property is checked against the service model (line 11). If a similar property is satisfied by the service, its similarity value is stored as its degree of match value for the current property (lines 13-14 and 18). After all properties are checked, we need to compute an overall degree of match value for the service. This overall value can be the average of degree of match values of individual properties or it can be the maximum (or minimum) degree of match, depending on whether the intended matchmaker properties or it can be the maximum (or minimum) degree can be the average of degree of match values of individual properties are checked, we need to compute an overall degree of match value for the service. This overall value can be the average of degree of match values of individual properties or it can be the maximum (or minimum) degree of match, depending on whether the intended matchmaker will work optimistically (or pessimistically). This is deliberately left as a customizable \( f \) method that can vary between implementations (line 21). If this overall degree of match value is larger than 0 then the service is selected as a match and added to the result set with its associated degree of match value (lines 22-23).

4.1. Method-1: Replacement of Similar Propositions

In our first method to support partial service matching, we use the semantic knowledge embraced by the LTL formula, in order to generate similar properties for a requested property. Each LTL formula consists of several individual propositions and each proposition has a certain meaning. For instance, the proposition \textit{hotelPaid} in previous examples means that the payment is made by the service customer to the service provider for the hotel reservation. Accordingly, a reasonable way to generate a similar property from another property is to change some propositions in the original property with some other relevant propositions. Consider the following example to make our idea clear: Assume that there are two propositions \textit{hotelPaidCC} and \textit{hotelPaidCash}, which are specialized cases of the proposition \textit{hotelPaid}, where the payment is made by credit card and by cash, respectively. Also, assume that there is a property \( p \) that involves the proposition \textit{hotelPaidCC}. Then, to generate a similar property \( p' \) from the property \( p \) we can replace the proposition \textit{hotelPaidCC} with the relevant proposition \textit{hotelPaidCash}.

As a result we have two similar properties \( p \) and \( p' \), where the only difference is the type of the payment in low-level view. Besides, in considering a high-level examination both properties still involve a payment proposition.

In order to be able to generate similar properties using this approach we have to capture semantic knowledge about the propositions involved in LTL formulae. To achieve this we create a domain ontology that defines the common concepts and their relations. For instance in the e-commerce domain for hotel reservation related services these might be \textit{hotelPrice}, \textit{hotelReserved}, \textit{hotelPaid} and \textit{hotelVouchered} such as in the example service model in Figure 1. Other than these fundamental concepts, there are more specialized types of these concepts. These specialized concepts create a hierarchical structure through parent and child relations. For instance, the fundamental concept \textit{hotelPaid} can be specialized into concepts such as \textit{hotelPaidCC} and \textit{hotelPaidCash} as we discuss. In such a case \textit{hotelPaid} has a parent relation to \textit{hotelPaidCC} and \textit{hotelPaidCash}. We can further specialize these concepts. For instance, \textit{hotelPaidCash} may have more specialized concepts such as \textit{hotelPaidCashDollar} and \textit{hotelPaidCashEuro}. Figure 2 shows a partial snapshot of such an ontology for the example that we present previously. Use of such domain ontologies is common in the service matchmaking and also in process modeling literature [29].

As we state in Section 4, it is also necessary to compute a similarity value between the originally requested and generated similar properties. The ontology of propositions provides us the suitable knowledge for this purpose. We can use this knowledge to feed any semantic similarity metric from the literature such as [28, 36, 42] to compute similarities between propositions in the original and generated LTL formulae, and by combining similarities of individual propositions (i.e. linear sum) we can obtain an overall similarity between two properties.

\begin{algorithm}
\begin{algorithmic}
\Require Service \( s \)
\Require Property \( r \)
\Require Ontology \( o \)
\Function{genSimPropSet}{\( s \), \( r \), \( o \)}
\State Dictionary \( rels \)
\For{each Concept \( c_e \) in \( r \)}
\For{each Concept \( c_s \) in \( s \)}
\If{\textit{semanticSimilarity}(\( c_e \), \( c_s \), \( o \)) > 0}
\State \( rels[c_e] + = c_s \)
\EndIf
\EndFor
\EndFor
\State \( \text{enumAltProp}(r, rels, altPropSet) \)
\State \Return \( altPropSet \)
\EndFunction
\end{algorithmic}
\caption{Method-1: Algorithm for \textit{genSimPropSet()} function using semantic similarities}
\end{algorithm}

We formalize our first method for partial service matching in Algorithm 3 and 4. These algorithms correspond to the implementation of the \textit{genSimPropSet()} function in Algorithm 2. This method is based on replacing propositions in the LTL formula that represents the request with relevant propositions in the ontology. We repeat this for each proposition in the request and for all available relevance relations and generate a new similar property for each replacement. However, replacing each proposition in the request with all possible relevant propositions in the ontology is inefficient, since the set of relevant propositions might be large. To overcome this problem, we can use the service model as a filter to limit the number of relevant propositions that we use in similar property generation as
Algorithm 4 enumAltProp() (Recursive property enumeration)

Require: Property $r$
Require: Dictionary $rels$
Require: Set $altPropSet$

1: if all Concept $c_p$ in $r$ considered then
2:  return
3: else
4:  for rel in relDict do
5:    altProp := replace($c_p$, $rels[c_p]$)
6:    altPropSet := altPropSet $\cup$ altProp
7:  enumAltProp(altProp, rels, altPropSet)
8:  end for
9: end if

follows: A similar property that we generate can be satisfied by the service under consideration only if the service model contains the replaced proposition. Hence, instead of blindly replacing a proposition in the requested property with all relevant propositions in the ontology, we only select the relevant propositions that are both part of the requested property and by the service model and use only these propositions to generate new properties.

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

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

Let us walk through our method with an example. Assume that we use the ontology that we present in Figure 2 and have a service model in which there are three propositions as follows: hotelReserved, hotelPaperVouchered and hotelPaidCash. The consumer request contains following propositions: hotelReserved, hotelEVouchered and hotelPaidCC. Note that both for the service model and request we ignore the structure components of the service model and LTL formula details of the request, since these details are irrelevant for our similar property generation method.

The matchmaking process (Algorithm 2) starts by checking if the requested property can be satisfied by the service (i.e., if there is an exact match). Since the propositions related to delivery and payment are different in the service model 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 3 first determines the relations between the propositions in the required property and the service model and creates the dictionary to hold the relations. The dictionary will contain the following:

- $property$.hotelPaidCC $\leftrightarrow$ $service$.hotelPaidCash
- $property$.hotelEVouchered $\leftrightarrow$ $service$.hotelPaperVouchered

We can see the advantage of using the service model as a filter to reduce the number of propositions to generate similar properties in this example. Normally, if we use the entire set of relations in the ontology, there are five propositions relevant to the proposition hotelPaidCC and two propositions relevant to the proposition hotelEVouchered and accordingly we generate 17 similar properties for the
requested property, which is large even for this small ontology. However, using the service model as a filter there is only one relevant property for each of the properties hotelEvouchered and hotelPaidCC. Hence, we generate only three similar properties, which is significantly small considering 17 similar properties without using the service model as a filter.

Although, using the service model as a filter reduces the number of generated similar properties significantly, this approach may still generate a large number of properties, especially if the service model is large or the property involves many propositions. In such a case, we can use a semantic similarity metric as a control mechanism to further reduce the number of generated properties. We can achieve this simply by ignoring the related propositions that have a similarity value below a certain threshold with the original proposition. The threshold value may be defined by the service matchmaker or service consumer. Hence, it provides great flexibility and control over alternative property generation. In one extreme, if the threshold is set to one, no alternative property is generated and only the original request is used for matchmaking, which corresponds to exact matching. On the other extreme, if the threshold is set to zero, then all possible alternative properties are generated.

<table>
<thead>
<tr>
<th>Req. Prop.</th>
<th>hotelPaidCC</th>
<th>hotelEVouchered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt. 1</td>
<td>hotelPaidCC</td>
<td>hotelPaperVouchered</td>
</tr>
<tr>
<td>Alt. 2</td>
<td>hotelPaidCash</td>
<td>hotelPaperVouchered</td>
</tr>
<tr>
<td>Alt. 3</td>
<td>hotelPaidCash</td>
<td>hotelPaperVouchered</td>
</tr>
</tbody>
</table>

Next, Algorithm 4 creates similar properties from the original property by enumerating all combinations in the dictionary. The similar properties created in our example is listed in Table 1. After the similar properties are generated, we return to the Algorithm 2 and it continues by checking each similar property against the service model. If a similar property is satisfied by the service model, it computes the similarity between the similar and requested property. As we state before this can be done by using one of the metrics presented in [28, 36, 42]. Also note that, this step corresponds to the compPropSim() function in Algorithm 2. As the last step, the algorithm determines the similar property with the maximum similarity value and returns this similarity value as the degree of match to the requested property for the service.

4.2. Method-2: Relaxation of Request Properties

In our second method to capture partial service matching, to generate alternative LTL formulae from the original LTL formula that represents the request of the service consumer, we use structural relations between the LTL operators. In order to achieve this, based on the entailment semantics of LTL operators, we define a set of relaxation relations. Each of these relaxation relations defines how an LTL formula in a certain form can be transformed into another LTL formula, which has relaxed temporal requirements. In our method, we do these relaxations always from an LTL formula that represents a strict property to an LTL formula that represents a more relaxed property. For instance, we transform a strict property that is required to be valid for all the time into a more relaxed property that is required to be valid only for some time. The intuition behind this approach is that when a service can satisfy a relaxed version of a more strict required property, which cannot be satisfied by the service, we can still consider this service as a partial match since it can provide a relaxed form of the required functionality. Besides these relations that follow entailment, we also define some relations based on intuition. Although such relations may not always follow entailment, semantically they are meaningful. In the rest of this section we define these relations and present examples for possible use cases. In these relations we use $\Rightarrow$ to represent the relaxation relation between the LTL formulae.

Relation 1 ($\phi \gg X\phi$). The LTL formula $\phi$ states that the proposition $\phi$ must hold in all future states starting from the current state. We relax this formula into $X\phi$, which states that the proposition $\phi$ must hold in all future states starting from the next state. In this way we slightly relax the requirements of the originally requested property through allowing $\phi$ not to hold for the initial state. In this relaxation relation the formula $\phi$ entails $X\phi$. We present this relation graphically in Figure 3.

Relation 2 ($\phi \Rightarrow FG\phi$). The LTL formula $\phi$ states that the proposition $\phi$ must hold in all future states starting from the current state. We relax this formula into $FG\phi$, which states that the proposition $\phi$ must start to hold in the current or in one of the future states and continue to hold in all remaining states. In this way we relax the original property so that instead of expecting $\phi$ to hold in all states, it is enough for $\phi$ to hold starting in some future state. In this relaxation relation the formula $\phi$ entails the formula $FG\phi$.

Relation 3 ($\phi \Rightarrow GF\phi$). The LTL formula $\phi$ states that the proposition $\phi$ must hold in all future states starting from the current state. We relax this formula into $GF\phi$, which states that the proposition $\phi$ must hold infinitely often in the future states. In this way we relax the original property by allowing $\phi$ not to hold in some of the states. In this relaxation relation the formula $\phi$ entails the formula $GF\phi$.

Relation 4 ($XG\phi \gg X\phi$). The LTL formula $XG\phi$ states that the proposition $\phi$ must hold in all future states starting from the next state. We relax this formula into $X\phi$, which states that the proposition $\phi$ must hold in the next state. Through this relaxation it is enough for the system to hold $\phi$ only in the next state instead of all future states starting from the next state. In this relaxation relation the formula $XG\phi$ entails the formula $X\phi$. 

10
Relation 5 \((FG\phi \gg F\phi)\). The LTL formula \(FG\phi\) states that the proposition \(\phi\) must hold in the current or one of the future states and continue to hold until the end of the service. We relax this formula into \(F\phi\), which states that \(\phi\) must hold in the current or one of the future states but it is not necessary to continue to hold after that. In this relaxation relation the formula \(FG\phi\) entails the formula \(F\phi\).

Relation 6 \((GF\phi \gg F\phi)\). The LTL formula \(GF\phi\) states that the proposition \(\phi\) must hold infinitely often in the future states. We relax this formula into \(F\phi\), which states that it is enough for \(\phi\) to hold in the current or one of the future states. In this relaxation relation the formula \(GF\phi\) entails the formula \(F\phi\).

Relation 7 \((X\phi \gg F\phi)\). The LTL formula \(X\phi\) states that the proposition \(\phi\) must hold in the next state. We relax this formula into \(F\phi\), which states that \(\phi\) must hold in the current or one of the future states. This relation is the only one that does not follow entailment. However, through common intuition it is clear that \(F\phi\) represents a more relaxed requirement than \(X\phi\), since instead of forcing \(\phi\) to hold immediately in the next state it lets \(\phi\) to hold in any one of the future states.

We use these relations to generate alternative properties from the originally requested property as follows: When a service cannot satisfy a required property we first check which of these relations can be applied to this property. Then we apply all of the applicable relations one by one and generate a new property for each of the applicable relation. We apply the relations only on the originally requested property. That is we do not further apply relations to generated alternative properties. We formalize this process in Algorithm 5, which corresponds to the \textit{genSimPropSet()} function in Algorithm 2. This algorithm takes a request property and an ontology that represents the temporal relations that we discuss. The algorithm iterates over each LTL operator in the requested property (lines 2-8). Note that, in addition to regular LTL operators, the iteration mechanism also covers the three combined operators \(XG\), \(FG\) and \(GF\), since these combinations of the operators are involved in our relaxation relations. Then the algorithm generates a set of LTL operators using the original LTL operator in the requested property and the relaxation relations (line 3) and create a new property for each LTL operator in generated LTL operator set (lines 4-7).

Algorithm 5 Method-2: Algorithm for \textit{genSimPropSet()} function using relaxation relations

\begin{algorithm}
\begin{algorithmic}
\Require Property \(r\)
\Require Ontology \(o\)
\Function{altPropSet}{\}
\State \(\text{altPropSet} = \emptyset\)
\For {each \text{LTLOp} \(\text{op}_r\) in \(r\)}
\State \(\text{OP}_{\text{rel}} := \text{getRelatedOps}(\text{op}_r, o)\)
\For {each \text{LTLOp} \(\text{op}_{\text{rel}}\) in \(\text{OP}_{\text{rel}}\)}
\State \(\text{altProp} := \text{replace}(\text{op}_r, \text{op}_{\text{rel}})\)
\State \(\text{altPropSet} := \text{altPropSet} \cup \text{altProp}\)
\EndFor
\EndFor
\State \Return \text{altPropSet}
\EndFunction
\end{algorithmic}
\end{algorithm}

Let us give an example to clarify our algorithm. Assume that the requested property is \(GF\phi\) and the ontology involves all the seven relaxation relations that we discuss. The request property \(GF\phi\) states that \(\phi\) holds infinitely often. Table 2 presents the generated relaxed properties and the used relaxation relations for this case. Relaxed properties P-1, P-2 and P-3 are created through the relaxation relations \(G\phi \gg XG\phi\), \(G\phi \gg FG\phi\), and \(G\phi \gg GF\phi\), respectively, on the operator \(G\) of the request property. Relaxed property P-1 states that, starting from the next state \(\phi\) holds infinitely often. Relaxed property P-2 states that, starting from one of the future states \(\phi\) holds infinitely often. Relaxed property P-3 is semantically equivalent to the original formula, since there is no difference using the \(F\) operator consecutively. Hence our method simply ignores this generated property. Relaxed property P-4 is created through the relaxation relation \(GF\phi \gg F\phi\) on the combination of operators \(G\) and \(F\). This property states that, \(\phi\) holds in one of the future states.

| Table 2: Generated relaxed properties |
|--------------------------------------|-----------------|-----------------|-----------------|
| P-1 | \(GF\phi\) | \(G\phi \gg XG\phi\) | \(XGF\phi\) |
| P-2 | \(GF\phi\) | \(G\phi \gg FG\phi\) | \(FGF\phi\) |
| P-3 | \(GF\phi\) | \(G\phi \gg GF\phi\) | \(GFF\phi\) |
| P-4 | \(GF\phi\) | \(GF\phi \gg F\phi\) | \(F\phi\) |

By investigating the relaxation relations between the temporal operators, it is clear that there are some hierarchical relations between these relations. For example there is a relation between the LTL formula \(G\phi\) and \(GF\phi\) and further there is a relation between \(GF\phi\) and \(F\phi\). Us-

![Figure 3: Relaxation for Gp to GXp](image-url)
ing this hierarchical nature of the relaxation relations we create an ontology, which we use to compute semantic similarity between LTL formulæ. We present this ontology in Figure 4. As in our first method for partial service matching (Method-1), it is possible to use different approaches when computing similarities using this ontology.

![Figure 4: Relaxation relation ontology of LTL operators](image)

5. The Service Matchmaker

In this section we discuss how our service matchmaking approach can be implemented using the state of the art standards and tools from Web services and model checking domains. First, we explain how SPIN, a general purpose model checker can be integrated into our service matchmaking approach. Then we survey OWL-S, a state of the art ontology that provides facilities to describe services and service request. We also discuss SWRL a semantic web standard to represent rules and describe an extension in order to represent LTL formulæ. In the last part we describe a service matchmaker architecture that combines these standards and tools and can be used as a guideline to realize our service matchmaking approach.

5.1. SPIN Model Checker for Service Matchmaking

The main computational mechanism behind our service matchmaking approach is model checking. Hence, efficiency of our approach heavily depends on the efficiency of the underlying model checking implementation. Since model checking algorithms are complex and require careful implementation for high efficiency, it is reasonable to use an existing model checker and build the service matchmaker on top of it, instead of developing a model checker from scratch. For this purpose we choose SPIN model checker [20]. SPIN is a widely used general purpose model checker with full LTL support.

Using SPIN a verification process is carried out as follows: the first step is to create an abstract model of the system that is going to be verified. In our case this system corresponds to a service model. SPIN uses a high-level specification language called PROMELA to represent system models. A system model in PROMELA is a set of asynchronous processes. To define processes, PROMELA provides basic control structures such as conditionals, nondeterministic choices and iteration constructs. To model data flow PROMELA uses variables and message channels. Both of these structures can be defined in local or global scope considering processes. Message channels provide also support for blocking capability, which can be used for process synchronization.

The second step in the verification process is to model the property that is going to be verified on the system model. In our case such properties correspond to the individual requirements of the service consumer in a service request. In SPIN such properties are expressed as regular LTL formulæ. Providing the system model and the property, SPIN checks the system model against the property and outputs whether the property holds in the model or not.

In Figure 5.1 we show parts of the PROMELA code that represents the service model that we present in Figure 1. In the PROMELA model the lines 1-4 are the global variables that correspond to boolean properties in the service model. Each property is initiated as false to indicate that none of them holds when the service is initiated. Processes manipulate these properties in order to model the data flow. The lines 6-9 shows the atomic process hotel-request. This process has the message channel syncChan that is used to synchronize this process with other processes. By executing this atomic process the value of the variable hotelReserved set to true that indicates the service provider makes the offer to the service consumer for the requested hotel. At the end of the atomic process a done message is sent by the syncChan to inform other processes that this process is completed. Each of the lines 11, 13 and 15 corresponds to other atomic processes in the service model. The lines 17-39 represent the composite process hotel-reserv-serv. The aim of this composite process is to model the flow of execution between the atomic processes. This composite process first executes the atomic process hotel-request (line 20). Then, there is a choice situation. If the service consumer is happy with the offered hotel, she may continue and confirm the hotel (lines 22-33). On the other hand, if she is not satisfied with the hotel, she may exit from the service immediately (lines 34-35). We represent this situation as a non-deterministic choice in the PROMELA model (line 24 and line 34). If the service consumer is happy with the hotel and decides to continue, atomic processes hotel-confirm, make-payment and hotel-voucher are sequentially initiated by the hotel-reserv-serv process in lines 25, 28 and 31, respectively. In the PROMELA model we use the channel childSync for synchronization of the atomic processes by the composite process. Also note that we do not involve information such as the id of the hotel or the offered price in the PROMELA model. Such kind of information is irrelevant for the model checking process.
5.2. Representing Service Models in OWL-S

Although PROMELA language provides a suitable tool to represent process models of services, it is not very suitable to describe a service in general. A service description involves other information such as high level abstract interfaces and low level communication protocols to interact with the service. OWL-S [30] is a state of the art ontology that provides constructs to describe a service. OWL-S is divided into three components, namely Service Profile, Service Grounding and Service Model. From these three components the Service Model provides a suitable tool to define service models. It is organized as a workflow of processes. Each process is defined through its input-output interface, its pre-conditions and its effects.

As we explain in Section 5.1 we use SPIN model checker in our service matchmaking approach and SPIN requires service models to be represented in PROMELA. Hence in order to use OWL-S for service description there must be a mechanism to convert service models in OWL-S into PROMELA models. Ankolekar et al. [3] propose a methodology for such a translation of OWL-S service models into PROMELA models.

5.3. Extending SWRL to Represent LTL Formulae

As we discuss in the previous section OWL-S provides appropriate structures to represent the service model. On the other hand OWL-S does not provide implicit support for LTL, which we require to represent service requests. However, it provides a generic structure called Expression that allows embedding expressions in XML format into OWL-S documents. A recent standard to represent expressions is Semantic Web Rule Language (SWRL) [21]. SWRL provides a standard schema to integrate Horn like rules into OWL ontologies. However, SWRL is not capable to capture every LTL formula since it supports only Horn like rules. SWRL-FOL [33] is an extension of SWRL that allows expressions in first order logic. Hence, SWRL-FOL provides an appropriate ground to represent LTL formula through extending it for the four LTL connectives U, G, F and X. We present the abstract syntax for the extension of SWRL-FOL for LTL in Figure 5.3. The semantics of this extension maps to the semantics of LTL that we present in Section 2.2.

The example in Figure 7 expresses the LTL formula G(p → (pUd)), where p stands for hotelPaid and t stands for hotelVouched, using the extended SWRL-FOL for LTL.

5.4. The Service Matchmaker Architecture

In this section we describe an architecture that combines the discussed standards and tools to realize our service matchmaker. We present the service matchmaker architecture in Figure 8. In the architecture, the service matchmaker has two modules, namely the Service Handler Module to deal with service providers and the Request Handler Module to deal with the service consumers.
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 ')

Figure 6: Abstract syntax for SWRL-FOL LTL extension

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

Figure 7: Example of extended SWRL-FOL for LTL

The Service Handler module receives incoming service descriptions in OWL-S documents from service providers and stores these descriptions in a repository. Additionally, it sends these descriptions to the OWL-S to PROMELA Translator Module. This module translates the process model data in the OWL-S document into a PROMELA model using the approach in [3] and stores it in a separate repository. The Request Handler Module accepts service requests from service consumers as OWL-S documents. It first sends this OWL-S document to the OWL-S to LTL Translator Module. This module extracts the LTL formulae involved in the OWL-S document and feeds the Service Matchmaker Module with the extracted LTL formula. The extraction of the LTL formula from the OWL-S document is straightforward given the abstract syntax in Section 5.3. The Service Matchmaker Module executes the service matchmaking algorithms that we define in Section 3 and 4. While executing these algorithms it uses PROMELA models of existing services from the PROMELA Model Repository and the LTL formulae extracted from the service request to feed the SPIN Model Checker. After executing the service matchmaking algorithms the Service Matchmaker Module returns the results back to the Request Handler Module. According to the result, the Request Handler Module gets the OWL-S descriptions of the matching services from the Service Description Repository and sends these descriptions back to the service consumer as the result of the service matchmaking process.

We have developed an initial prototype implementation of our matchmaker architecture. Our current implementation is in Python language, which uses SPIN model checker in background for model checking. The implementation is capable of taking a set of service models in PROMELA language and a set of LTL formula and checking each formula against all service models using exact matching. For the partial matching, it uses a hard-coded ontology and hence currently only works on a predefined set of examples.

### 6. Evaluation

An experimental evaluation of our approach with comparisons to existing matchmakers is difficult, because there are no standard service sets that include service process details or even precondition-effects. As a result, in order to evaluate our approach, we first prepare a set of services and a set of matchmaking queries that correspond to user requests. We study how well our approach can find relevant services for our requests. We compare this with other approaches, mainly input-output matching and precondition-effect matching. Additionally, to validate our

[1] The code, instruction for running the current implementation of our matchmaker and examples from Section 6 are available at http://mas.cmpe.boun.edu.tr/wiki/doku.php?id=research:matchmaker
We created a test collection of 150 services from e-commerce domain. Each service specification in the collection includes the process model of the service (as explained in Section 3.1) as well as the input-output interfaces and precondition-effect information. Complexity of the service models varies from a simple look-up service to a composite selling service that includes several atomic operations such as price query, order, payment and delivery. Our service collection contains services that are similar to each other with minor modifications, so that we can include services with same fundamental functional capabilities but with varying temporal properties. This is useful for capturing situations in which many similar services exist but they all have minor but important differences from a user's point of view. We use also an ontology of propositions similar to the one that we discuss in Section 4.1 to generate services, where several services share the same structural model but some of the atomic processes are replaced with similar propositions from the ontology. For instance, there are two identical services with the exception of the payment method they use. In this way we can evaluate the performance of our partial matchmaking method that we discuss in Section 4.1.

We formulated 10 test queries with different complexities and associate each query to a set of services. Each query includes one or more individual properties such that each property represents a different requirement of the user from the service. A simple query such as "A service to query book prices." requires only one property to be represented. On the other hand, a more complex query such as "A service to query and buy a book that guarantees delivery of the book before the payment and also secure connection during the whole connection." requires four individual properties to represent such a situation.²

### 6.2. Recall-Precision Performance

In order to evaluate our approach, we adopt the experimental setup presented by Klusch et al. to our test collection [26]. This setup is based on the macro-averaging of precision and recall metrics. We compute the recall and precision metrics using the standard equations as in the Equations 1 and 2, where $TP_q$ is the number of successfully retrieved services for the query $q$, $FN_q$ is the number of services that fail to be retrieved by the matchmaker for the query $q$ and $FP_q$ is the number of falsely retrieved services for the query $q$.

$$\text{Recall}_q = \frac{TP_q}{TP_q + FN_q} \quad (1)$$

$$\text{Precision}_q = \frac{TP_q}{TP_q + FP_q} \quad (2)$$

We apply the evaluation strategy of macro-averaging of individual precision values over the set of requests $Q$ for different recall levels $\lambda$ as follows. For each query $q \in Q$, the matchmaker returns a ranking of all available services according to the computed degree of match values of services (between $[0, 1]$) for the query $q$. Then, for each recall level $\lambda_q$, we compute the precision value that at this recall level $\lambda_q$, considering the services to achieve the recall

²Our service and query collection is available at: http://mas.cmpe.boun.edu.tr/wiki/doku.php?id=research:matchmaker.
level $\lambda_l$. Finally, we average the computed precision values at each level of $\lambda_l$ over all the queries to obtain the macro averaged precision values. We compare the recall-precision (R-P) performance of our approach with input-output and precondition-effect matching. We apply our approach (MC-Match) as described in Section 3 with the partial matching method as described in Section 4.1. For input-output matching we use the method named OWLS-M0 as defined in [26] and for precondition-effect matching we adopt the approach described in [27] (PE-Match).

Figure 9 plots the R-P performance of the three matchmaking approaches that we use in our experiments. The x-axis shows the fixed recall levels ($\lambda_l$) and the y-axis shows the corresponding precision values obtained for each recall level $\lambda_l$. Our proposed approach MC-Match performs better than the other approaches OWLS-M0 and PE-Match in terms of both recall and precision\(^3\). On the average, MC-Match can keep the 0.8 precision value until recall level increases to 0.4. On the other hand, its worst precision value is 0.63 when it retrieves all matching services. The average precision value of the OWLS-M0 is around 0.6 for all recall levels, with a slight decrease for each level. PE-Match shows a similar behavior. The major reason behind this better R-P performance of our approach depends on the better representation of temporal properties of user requests in our test collection. We discuss several such properties in detail in Section 6.3. All approaches lose their precision when recall is increased. This is an expected result as observed before by Klusch et. al [26]. This is due to the increasing number of falsely retrieved services, while trying to improve precision. Overall our precision-recall ratio is promising. Note that compared to some previous results [26], on our test case none of the approaches demonstrate a low R-P performance, even for high recall levels. We predict that this is due to the nature of our service set, in that the services are from one domain and more similar to each other compared to the large and diverse set of services in previous studies.

6.3. Case Study

In this section, we demonstrate the capabilities of our approach by explaining how some properties are handled in detail using an example service. For this purpose, we use the hotel reservation service in Figure 10. This service is a more detailed version of the hotel reservation service that we present in Figure 1. In Figure 10, the labels associated with arcs are the transitions. Transitions whose labels start with a "?” represent the inputs to the service and the transition whose labels start with a "!” represent the outputs of the service. The transitions without a "?” or "!” are internal processes of the system, which do not involve any interaction with external parties. The bold typed labels associated with states are the propositions that start to hold in that state. For simplicity, we do not explicitly present the truth value of all propositions in each state. Instead we only show a proposition only if its truth value changes. We assume that initial value of all propositions are set to false.

We provide a detailed description of the process executed by the service. The service is initiated by the service consumer by making a request to book a hotel room (hotel-request). If the request cannot be satisfied, then the service informs the service consumer about the situation and exits (not-available). On the other hand, if the hotel is available, then the service makes an offer to the service consumer (hotel-offer). As the result of this transition, the value of the proposition hotelReserved is set to true. At this state, if the service consumer is not happy with the hotel offer, she may reject the offer and exit the service (reject-offer). On the other hand, if the service consumer is happy with the offer, she confirms the offer (hotel-confirm). Accordingly, the value of the proposition hotelConfirmed is set to true. The service continues by creating the hotel voucher (hotel-voucher) and sets the value of the proposition hotelVouched to true. After that the service enables secure connection (enable-secure-connection), which sets the value of the proposition secureConnection to true. Than the service waits for the payment of the service consumer. When the consumer pays (payment) the value of the proposition hotelPaid is set to true. Than the service disables the secure connection (disable-secure-connection) and the value of the proposition secureConnection is set to false. At that state the service consumer may exit from the service (exit). On the other hand, the service consumer may cancel her voucher (hotel-cancel). In this case the service makes a refund, which changes the state of the proposition refunded to true and state of propositions hotelReserved, hotelConfirmed, hotelPaid and hotelVouched to false. Then, the service exits.

The first type of property that we discuss enforces a requirement about the temporal ordering of two or more transitions. As an example, we consider G(payment -> hotelVouched). This property states that in any state of the service, if the customer makes a payment, then she should also have a hotel voucher. Since we only consider sequential executions, this is equivalent to saying that hotel voucher must be available before payment. In our approach, we use model checking on the service specification to confirm that indeed the hotel reservation service satisfies this property, since the service creates the voucher first in state five and accepts the payment later in state seven.

On the other hand such a case cannot be captured by input-output or precondition-effect matching since those approaches only consider the entry and exit points of the service but not the temporal knowledge while matchmaking. In input-output matching, a sample input can be payment and reservation dates and the output can be hotel voucher. However, many services that process hotel reservations will have the same input-output and hence

\footnote{For more recent results of different input-output matchmakers see S3 contest: http://www-ags.dfki.uni-sb.de/~klusch/s3/index.html}
Figure 9: Recall-precision performance of MC-Match, OWLS-M0 and PE-Match service matchmaking approaches.

Figure 10: The transition system of the extended hotel reservation service
Let us present the following example to validate our claims. Assume that there is another hotel reservation service, which is identical to the one in Figure 10 except for the order of voucher preparation and payment. Figure 11 shows this different order of transitions. It is obvious that when the above property is checked against this service, our approach will capture the difference and state that the property cannot be satisfied with this service. On the other hand, from precondition-effect matching point of view these two services are identical and both are claimed as a match to the service consumer’s request, although this is not true.

The second property we discuss is \( G(payment \rightarrow ((payment \lor hotelVouched) \lor (payment \lor refunded))) \), which states that in all states, the service must guarantee that the service consumer receives either a voucher or a refund in any remaining state after she makes a payment. The hotel reservation service satisfies this property as follows. The value of the proposition \( hotelPaid \) is set to \( \text{true} \) in state seven. Since the value of the proposition \( hotelVouched \) is already set to \( \text{true} \) in state five, the requested property is satisfied by the service for this state. Furthermore, the only possible transition to set the value of the proposition \( hotelPaid \) to \( \text{false} \) is the refund transition. However, if this transition is executed, the value of the proposition \( \text{refund} \) is set to \( \text{true} \), hence the service still satisfies the requested property. Using model checking techniques, our approach can capture satisfaction of this property by the hotel reservation service.

On the other hand, like in the previous case, input-output matching and precondition-effect matching do not provide precise results. In the input-output matching, we can define the inputs the same way we did for the previous case and output as either a refund or a hotel voucher. Our service will satisfy this request as well as other services that produce any kind of refund at any time or hotel voucher of any sort. Clearly, not all of these services will respect our expected requirement.

If we try to capture the above property using preconditions and effects, we face two difficulties: (1) it is not possible to represent a temporal requirement using only service inputs and outputs and (2) the property states a requirement for all states of the service, not just the final effect. If we were to express this property in terms of an effect, one way would be to state that the expected effect is that both the payment and the hotel reservation be made or the payment and refund be made. When deriving this effect, we are ignoring both the temporal requirements and the fact that all states of the service should satisfy this requirement. As a result, this request again will incorrectly be matched to several services that do not respect the temporal relation.

Let us demonstrate the situation with another example. Assume that there is another hotel reservation service, which is identical to the one in Figure 10 except the proposition \( hotelVouched \) is set to \( \text{false} \) in state nine immediately after the transition \( hotel-cancel \) instead of state ten. Figure 12 shows this difference. In this case, the service is in an inconsistent state in state nine, since there is a payment but no hotel voucher. If some exception occurs (e.g., service gets offline), it is not clear how the service should recover or what guarantees can be given to the service consumer. Clearly, in this state our desired property does not hold. However, if in the end, somehow the service recovers by giving a refund, our simplified effect will hold and hence the service will be considered a match for this request. However, our approach will be aware of
this subtle difference and even if one state does not satisfy the requested condition, it will signal a mismatch.

The third property we discuss is $G((secureConnection \rightarrow \neg payment) \vee \neg payment)$, which is required by the service consumer to guarantee that if there is ever a payment transition, then the service provides a secure connection during this transition. Our approach captures that this property is satisfied by the hotel reservation service, since the service makes the connection secure by the transition $enable-secure-connection$, which makes the proposition $secureConnection$ true, in state six and the only payment transition is done after this state.

Like in the two previous cases, this property contains an interesting structure that is not easy to represent as precondition and effect: The effect is based on a condition. In this case, the secure connection is only necessary if there is a payment. Using precondition-effect matching it might be possible to capture that the service provides some secure connection, if a related input or output exist in the service definition, however it is not possible to understand, when the secure connection is realized by the service.

Let us present two example services, which have minor differences than the hotel reservation service, to demonstrate how our approach successfully captures different cases. The first service provides a secure connection for the whole execution of the service. As shown in Figure 13, this service executes the transition $enable-secure-connection$ immediately when the service is initiated and keeps it in the secure state until the end of execution. This service satisfies the requested property. Our approach successfully identifies this service as a match. The second service provides a secure connection, but only if there is a refund transition as shown in Figure 13. Since during the execution of the payment transition the value of the proposition $secureConnection$ is false, our approach successfully captures that the service does not satisfy the required property and does not match the service to the request. If we consider how precondition-effect matching would treat these two services, we will see that if the effect is specified as $secureConnection$, then both services will match the request, which is clearly not the expected result.

In all above cases, we see that when the service consumer expresses her request in terms of the process captured by the service, denoting the request to a precondition-effect pair does not capture the request precisely. Either the temporal ordering of transitions is lost, or the conditions that need to hold in all states of the service are represented as a requirement to hold only at the end, and so on. Hence, in all three cases, we see that when the request is expressed as an effect, the request will match to more services than are expected. This clearly decreases the precision of the precondition-effect matching compared to our proposed approach.

On the other hand, if a request is specified as an expected effect, we can always specify this using a temporal logic formula. Basically, specifying the expected condition with the $F$ operator will mean that eventually that condition holds, which is identical in meaning to the effect of the service. Hence, our proposed approach will match a given request in precondition-effect matching to only those services that are also matched with precondition-effect matching.

The last property we discuss is $G(secureConnection)$, which is required by the service consumer to guarantee that the service provides secure connection during the whole execution of the service. Our hotel reservation service does not satisfy this property and our approach can capture this, since in some states the value of $secureConnection$ is false. Now assume that the service provides secure connection for the whole execution of the service. That is starting from the first state the value of $secureConnection$ is true in all states in the service model. Our approach can capture that in this case the requested property is satisfied by the service. Now, consider another service that provides two alternative executions. In the first alternative execution the service provides a secure connection for the whole connection. For the second alternative execution, the service provides identical functionality except a secure connection. The decision of which alternative to execute depends on the choice of the user. Intuitively, a user can engage this service, choose the alternative that applies secure connection throughout, and be satisfied. On the other hand, our approach would yield that the request cannot be satisfied by the service. This is due to the semantics of the operator $G$ and $LTL$, which implicitly quantifies universally over all possible executions. According to that the proposition associated with the operator $G$ must hold in all states of all possible executions, which is not true in our case. Hence, using our approach we cannot capture that
this service is a match for the requested property. This problem arises, since LTL does not have an operator to represent existence of paths, as some other logics such as computation tree logic (CTL) has.

7. Discussion

In this paper we present a novel service matchmaking approach that is based on model checking. In our approach we represent services as system models and service requests as a set of LTL formula, in which each formula corresponds to a specific functional property required by the service consumer. Considering these two representations we use model checking techniques as a ground for our service matchmaking approach. Our service matchmaking approach is capable of handling both exact and partial matching of services for service requests. In order to capture partially matching services we propose two methods, which are based on user request restructuring. Our first method uses ontologies to capture semantic similarities between concepts that are involved in service models and service requests and accordingly restructures service requests. Our second method uses entailment and subsumption relations between LTL operators through a hierarchical structure and uses these relations for service request restructuring. Our primary contributions are as follows:

- We propose a novel model checking-based approach for service matchmaking.
- We describe how services can be modeled as system models and service requests as a set of LTL formula.
- We develop a service matchmaking algorithm that supports partial matches through service request restructuring. The algorithm is generic enough such that different restructuring methods can easily be integrated into the algorithm.
- We propose two restructuring methods for service requests. The first method is based on semantic similarity of service request concepts. The second method is based on the entailment relations of LTL operators.
- We provide a guideline to realize our approach using state of the art technologies in the Web services and formal verification domains.

Our choice of LTL is motivated by the fact that LTL is powerful enough to capture many temporal relations that are expected to be seen in a process model. Further, tools such as PROMELA are geared towards LTL. By using LTL, we can show why and how it is necessary to capture the process models of services, rather than only their syntactic or semantic inputs and outputs. However, if the process model is more intricate, it might be necessary to have more advanced temporal relations. If that is the case, other temporal logics, such as CTL or CTL* may be preferred.

Our proposed approach can be used as a matchmaking approach as discussed in detail. As an alternative, it can be used in cooperation with other approaches that are based on interface matching. That is, one can apply interface matching to find a subset of the possible services that can satisfy the input-output requirements of the user. But, as we have demonstrated, not all such services would satisfy the process related requirements of the user. To even further filter the services found with interface matching, we can apply our proposed approach that will check if the service satisfies various temporal requirements. We expect that this two step procedure would improve the efficiency of matchmaking. Since interface matching is computationally less expensive than model checking, it would be an efficient way to prune the service space with that approach first. We would then need to only apply model checking on a selected list of services.

Service matchmaking attracts attention from the service-oriented computing community in the recent years. Most of the studies on service matchmaking concentrate on interface matching. The idea of interface matching is inherited from the approaches on software component matching [43]. In interface matching approaches, services and service requests are represented as a set of inputs and outputs and it is assumed that a service matches to a request
if both the service and the service request share a common interface. LARKS [39, 40] is one of the first proposed systems for service matchmaking that use interface matching. LARKS is an agent capability description language in which services provided by agents are described through detailed service profiles. A service profile involves input and output parameters of the service and constraints on these parameters. In LARKS it is also possible to associate these parameters with semantic concepts from ontologies, which are defined in the concept language ITL. The matchmaking mechanism of LARKS use the input output parameters and the constraints on them for interface matching. Partial matches are supported through plug-in matching in which a service plugs in to a service request when the service expects more general inputs than specified in the service request and produces more general outputs than specified in the service request. Such situations are captured through checking the semantic subsumption relations of concepts that are associated with input output parameters in ontologies. Additionally LARKS also provide other syntactic mechanisms for service matchmaking such as term frequency-inverse document frequency on the service profile to find services that are similar to service request. Paolucci et al. propose a general service matchmaking algorithm that is similar to the interface matching mechanism of LARKS [31]. In this algorithm, addition to the exact and plug-in matches there is also a subsumption match in which a service request subsumes a service when both the input and output parameters in the service request semantically subsume the input and output parameters of the service. Li and Horrocks [27] formalize a framework based on the description logic considering these degree of matches and adding a new degree of match as intersection match. Gao et al. [18] analyze the exact and plug-in matches in a theoretical framework using abstract state machines and show isomorphism and simulation correspond to these matches, respectively. In OWL-MX [26] Klusch et al. propose to combine semantic subsumption based methods with well known syntactic similarity metrics from information retrieval research in order to achieve better service matchmaking for services described in OWL-S language. In addition to improving the matchmaking performance of interface matching, using similarity metrics they are able to associate quantitative results between matching services and requests. Hence, they are able to provide to the service consumer more fine-grained results. In [23] Kaufe and Klusch apply a similar approach to Web Service Modeling Ontology, which is a different formalism for modeling Web services. Contrary to our approach, these approaches are mainly based on interface matching. In real life, our approach can be combined with an interface matching approach to create a hybrid matchmaker. This would improve our results even further.

Benatallah et al. [6] propose a request rewriting based approach for service matchmaking. In this approach, input and output concepts are replaced by their corresponding descriptions of an ontology. By using these descriptions, they define the concept of profile cover that represents whether a service can cover a service request. Then using profile cover concept they define the problem of best profile covering. They show this problem can be reduced to the problem of finding minimal traversals in hyper-graphs and accordingly develop an algorithm for service matchmaking using this reduction. Colucci et al. [11] propose a description logic based approach in which different than subsumption they use two non-standard inference mechanism called concept contraction and concept abduction. Through these mechanism they try to create negotiation spaces and rank partial matches, which allows further bargaining of matchmaking results. Brogi et al. [8] propose a composition-oriented service discovery approach. In that approach when the requested service interface cannot be satisfied by one service, they try to find a composition of a set of services that can provide the demanded interface. Keller et al. [24] propose a two phase approach for service discovery. In the first phase called service discovery a set of services selected comparing interfaces in an abstract level. Hence these selected services do not accurately match to the service request. In the second phase called contraction they examine the services selected in the first phase in a more concrete level and determine services that actually match to the service request.

Although interface matching based service matchmaking approaches are quite popular they have the following two major weaknesses: First weakness of the interface matching based service matchmaking approaches is the precision of results. Some interface matching based service matchmaking approaches suffer from low precision, since they do not consider the internal processes of services while performing the matchmaking operation. As a result, different services with identical interfaces are counted as good matches although they may perform completely different tasks. Accordingly, the number of false positives increases and the precision of service matchmaking decreases. Our approach overcomes this issue by using the process models of services as primary source of knowledge. Since these models precisely define what a service does, the use of them in service matchmaking increases the precision of the process. Second weakness of the interface matching based approaches is the granularity of matchmaking results. Generally, the results of the interface matching based service matchmaking is coarse-grained. That is, the matching services are associated only with some general qualitative degree of match values such as exact, plug-in, and so on and it is not possible to further discriminate between services that have the same match degree. This level of granularity is unacceptable, especially when the number of matching services is large. A better matching approach should provide more precise and quantitative values about degree of match between services and service requests and should be able to rank the services based on this quantitative values. Accordingly, our approach can provide quantitative results using semantic similarity metrics in the case of partial matches.
Other than the interface matching based service matchmaking methods there is a small number of work that uses process models as a basis for service matchmaking. Klein and Bernstein [25] propose an indexing mechanism to create a hierarchy of process models, where models are represented using a work-flow language. The process models are defined in an ontology, which is used by both service providers and service consumer. For this purpose the authors rely on the MIT Process Handbook [29]. They also develop a query system that works on the hierarchy of process models for service retrieval purposes. Similar to our approach, this approach also uses the process models of services instead of the input-output interfaces. However, the indexing and matchmaking mechanisms are totally different than our model checking based method. Wombacher et al. [41] propose a matchmaking approach, which uses a finite state machine (FSM) to model a service and for matchmaking they mainly use disjunction, conjunction and intersection relations between these FSM models. In this approach, they convert the FSM model of a services into a set of conjunctive FSM models. That is, instead of using a single FSM model for a service they use several FSM models, where each model represents a conjunctive branch in the original FSM model. The aim of this approach is to distinguish the implicit disjunction and conjunction relations in the original FSM model and guarantee that all the requested conjunctive branches are satisfied by services. Similar to this approach, we use transition systems to model services. However, our matchmaking method is different than their method that uses conjunctive and disjunctive relations. Additionally, they do not provide any partial matching mechanisms. In our previous work [19], we proposed another service matchmaking approach that uses internal process models of services as primary source of information. In that approach we also used FSM models to represent both services and requests. For matchmaking we computed a similarity between the FSM models of services and requests based on metrics that we developed by combining structural and semantic similarity metrics. In [5] Bansal and Vidal use a tree structure to represent a DAML-S service model and apply an algorithm that starts from the root node and recursively examines all nodes in order to determine that the service matches to a request. However, in this approach a service request is still a set of inputs and outputs. Hence it does not capture the semantic of the request as we achieve using LTL formulae.

In [4, 35] semantic service discovery is used in pervasive-computing and Blue-tooth environments. These papers mainly discuss the architecture and protocols for service discovery in peer-to-peer environments. However, partial matching and ranking issues are also discussed and some logical formulation is provided, but no formal framework is defined for these issues.

To the best of our knowledge, although model checking methods applied to verification of services [3, 17, 38], they are not used in the matchmaking of services. Hence our model checking based method is unique for service matchmaking.

An interesting work that uses process models and model checking in the context of service composition is the paper of Pistore and Traverso [34]. In order to achieve automated composition of Web services, they apply a model checking based approach using a set of service process models in combination with a set of choreographic assumptions expressed in temporal logic. The main advantage of their approach is the consideration of assumptions while composing services, which are crucial since when they are violated the composition does not make sense. Our work differs from theirs in the sense that they use process models and model checking for composition, while we use them for service matchmaking. Also, they do not provide techniques to relax service requests as we have done here.

In our service matchmaking approach we do not explicitly consider pre-conditions, which is discussed in some related work [23, 39]. Ankolekar et al. [3] also discuss that there is no way to translate pre-conditions defined in OWL-S into PROMELA models. However, it is obvious that pre-conditions may provide useful knowledge for service matchmaking in certain situations. Hence, we plan to investigate the use of pre-conditions into our matchmaking approach in our future work. We keep the similarity value computation of the relaxation relation based method rather abstract. In our future work, we plan to conduct an experimental study on this issue in order to investigate the influence of possible similarity metrics. Hybrid service matchmaking approaches is a recent trend emphasized in WS community [23, 26, 40]. Our service matchmaking approach can be combined with other matchmaking approaches such as interface matchmaking. We leave this also as a future work.

Acknowledgments
This research has been partially supported by by Turkish State Planning Organization (DPT) under grant DPT 2007K120610, Boğaziçi University Research Fund under grant BAP09A106P and the Scientific and Technological Research Council of Turkey (TÜBİTAK) by a CAREER Award under grant 105E073. This paper significantly extends a paper that appeared in the proceedings of the seventh International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008). The first author has been partially supported by TÜBİTAK 2211 National PhD Scholarship Program.


