

# Coupled Signature and Specification Matching for Automatic Service Binding<sup>\*</sup>

Michael Klein<sup>1</sup> and Birgitta König-Ries<sup>2</sup>

<sup>1</sup> Institute for Program Structures and Data Organization, Universität Karlsruhe,  
D-76128 Karlsruhe, Germany [kleinm@ipd.uni-karlsruhe.de](mailto:kleinm@ipd.uni-karlsruhe.de)

<sup>2</sup> Institut für Informatik, Technische Universität München, D-85748 Garching,  
Germany [koenigri@in.tum.de](mailto:koenigri@in.tum.de)

**Abstract.** Matching of semantic service descriptions is the key to automatic service discovery and binding. Existing approaches split the match-making process in two steps: signature and specification matching. However, this leads to the problem that offers are not found although they are functionally suitable if their signature is not fitting the requested one. Therefore, in this paper, we propose a matching algorithm that does not use a separated and explicit signature matching step, but derives the necessary messages from the comparison of pre- and postconditions. As a result, the algorithm not only finds all functionally suitable services even if their signatures do not match, but also is able to derive the messages needed for an automatic invocation.

**Keywords:** Automatic Service Discovery/Invocation, Matching Semantic Service Descriptions, OWL-S

## 1 Introduction

An important vision of service oriented computing is to enable semantic, dynamic service binding, i.e. it should become possible to automatically choose *and* invoke service providers at runtime.

To achieve this, appropriate means to describe and match services are needed. The techniques developed by both the web services and the semantic web communities offer a suitable basis, but fall short of realizing the vision.

Maybe the major drawback of existing approaches to service description and matching is that they focus on the message flow of services. Effects and preconditions of services are either regarded separately from the message flow or, more frequently, not regarded at all. The matching process is thus performed in two steps:

- *Signature matching.* Checks whether the exchanged messages of the requested and offered service fit together. In OWL-S, e.g., this is done by comparing the **input** and **output** properties. Typically, it is analyzed whether the client provides at least the inputs needed by the service and whether vice versa the service produces at least all the outputs needed by the client.

---

<sup>\*</sup> This work is partially funded by the Deutsche Forschungsgemeinschaft (DFG) within SPP 1140.

- *Specification matching.* Checks whether the offered service provides the requested functionality. In OWL-S, e.g., this is done by comparing the state transition given in the precondition and effect properties.

An offer fits a request if their descriptions match in both steps. If this is the case, the service can directly be invoked as the message flows are essentially the same.

The big problem of this approach with respect to dynamic service binding is that the assumption that equality of message flow equals equality of functionality is wrong. Services with identical messages flows may offer completely different functionality; services with identical functionality may use different message flows to achieve this functionality. Thus, if matching is based on the comparison of message flows, on the one hand, services offering a functionality different from the one intended by the requestor may be invoked, on the other hand, appropriate services are overlooked, if their message flow does not match the request. The latter case is very common:

- The request demands for outputs that are not specified in the offer description because they are constant. For example, the requestor searches for a printing service that informs him about the location of his printout after service execution. For the offerer, however, the location is not an output of the service description as his service always prints on the printer in Room 335.
- The offerer demands for inputs that are not specified in the request as the requestor did not know that this value was necessary or omitted this input because it was constant. For example, it could be possible that the requestor did not know that the offered printing service needed the location of the printout or he wanted the printout always to be in Room 350, so he did not specify it as a input in his request.
- The offer demands for inputs that are specified as outputs in the request. For example, the offerer could request for the location of the printout, while the requestor wants this value as output.

In these cases, the signature matching fails and prevents the requestor from using a functionally suitable service.

In this paper, we present a novel matching approach that does not rely on an explicit signature matching step, but derives the necessary messages from the comparison of pre- and postcondition. The algorithm operates on state based service descriptions, which have been presented in [1]. Here precondition and effect are described by states with integrated variables which represent the message flow. With this approach we are able to find service providers that offer the desired functionality, even if their signature differs from the one specified in the request. At the same time, we are able to determine precisely which messages need to be sent for a successful service invocation. With this knowledge, it becomes possible to automatically generate these messages from the request and thus to fully automate service invocation.

The paper is structured as follows: In Section 2, we inspect existing approaches on matching semantic service descriptions. After that, in Section 3, we revisit our state oriented service description by showing which additional description elements are needed and by giving an example for a request and an offer description. Section 4 introduces our novel matching algorithm that operates on these descriptions and couples signature and specification matching. Finally, a conclusion is given in Section 5.

## 2 Related Work

As addressed in the introduction, existing approaches compare services mainly by explicitly matching their messages descriptions.

A prominent representative is the *Semantic Matchmaker* of the Software Agent Group at CMU developed by PAOLUCCI ET AL. [2]. The algorithm operates on OWL-S descriptions and tests if the request can provide all required inputs and if the offer's output satisfies the requestor's demands. As an exact match of the types is very strict, the matching degree **exact** is weakened: An output  $r$  of a request is also matching 'exactly' to an offer's output  $o$  if  $r$  is a direct subclass of  $o$ . When comparing inputs, the condition is inverted:  $r$  has to be a direct superclass of  $o$ . The reasoning behind this is that a service will only use the superclass to describe its functionality if the output offers *all* types of the direct subclasses. Besides **exact**, there are two other matching degrees: **plugIn** and **subsumes**. They allow a greater deviation from the original type. In any case, the approach relies on the message flow only, which leads to different problems as we have seen. Moreover, in cases of a non-exact match, the offer cannot be used directly as the messages of request and offer are not compatible. In the *Mind Swap* project [3], a similar approach is pursued.

An extension of this approach is the matcher for *LARKS* [4,5] which was developed by SYCARA et al. In *LARKS*, services are described by four functional parameters **input**, **output**, **inConstraints** and **outConstraints**. The algorithm uses up to five filters to match the description. The first three filters use techniques from information retrieval, the fourth filter is the *Semantic Matchmaker* described above, the fifth filter separately compares the pre- and postconditions. In summary, the approach suffers from the same problems as the *Semantic Matchmaker* as it directly compares the message descriptions, too.

TRASTOUR et al. at *agents@HP Lab* have developed a matching algorithm for service descriptions based on RDF [6]. The descriptions rely on a common RDF schema for an **Advertisement**. Thus, the comparison is performed by a graph matching approach: (1) Two descriptions are similar, if the root elements are similar and (2) two elements  $a$  and  $b$  are similar if (a)  $a$  is a subclass of  $b$  and each property of  $a$  which is also property or subproperty in  $b$  points to a similar element. It is also possible to attach special matching code to the elements. Although the algorithm seems to be very useful, the problems stemming from explicitly matching message descriptions reside. However, we will use this

algorithm as starting point for our matching process (see Section 4.2: ‘The Basic Algorithm’) and adopt it to state oriented service descriptions.

Furthermore, approaches that rely on logical derivation (e.g. on reasoning in description logics) perform their operations on explicit message descriptions, like in [7], [8], [9], and [10].

To summarize, the major drawbacks of existing approaches are: Most of them rely on signature matching. These are not able to find services that offer equivalent functionality with a different interface. Approaches that take the desired functionality into account and restrict the conditions on signature matching are able to find appropriate services. However, they are not able to automatically invoke them, as they do not include mechanisms to map the message flows expected by requestor and provider to one another.

### 3 State Oriented Service Descriptions

The main idea of this paper is to avoid an explicit matching of message descriptions but to derive necessary messages by comparing pre- and postconditions from request and offer. This becomes possible if the description of input and output is integrated into the states of pre- and postcondition. Therefore, in a first step, we will revisit our *purely state oriented description*<sup>1</sup> in the Sections 3.2 and 3.3. The description is based on classic elements like classes, properties, and instances. However, it needs additional description elements: sets and variables. We will introduce them in the next section.

#### 3.1 Additional description elements: Sets and Variables

To describe offers and requests in a state oriented manner, we need enhanced description elements: declarative object sets and variables. **Object sets** are sets containing objects of one (or more) types. They should not be mixed up with classes which contain *all* objects of the corresponding type.

An object set is described declaratively, i.e. by giving conditions that have to be fulfilled in order to be a set member. Graphically, such a set is represented as a rectangle with a small triangle in the left upper corner (see Figure 1). The following three conditions are possible in a definition:

- A *type t*. All objects in the set have to be exactly of this type *t*. This restriction can be weakened by a different type check strategy (see below). We access the type of a set *s* by `type(s)`.
- A list of *direct conditions*. Direct conditions are simple conditions that are restricting the members of the set. Each object in the set has to obey all these conditions. If the type of the set is a primitive datatype (like `Integer`, `String`, `Date` etc.) all typical comparison operations for this type are allowed; if the type is a user defined type (like `Person`, `Format` etc.) only direct comparisons to named objects of this type are allowed (like ‘= pdf’). We access the direct conditions of a set *s* by `directConditions(s)`

---

<sup>1</sup> A predecessor of this description was introduced in [1]

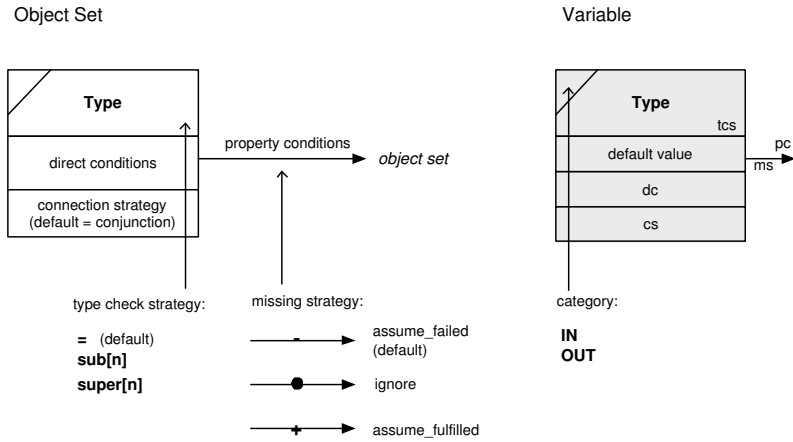
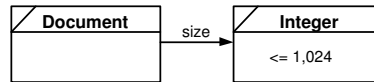


Fig. 1. How to define the enhanced description elements: object sets and variables.

- In case of a complex type: a list of *property conditions* which are depicted as named arrows at the rectangular. Only properties of the corresponding type can be used as property conditions. For this paper, only properties with cardinality  $\langle 0, 1 \rangle$  or  $\langle 1, 1 \rangle$  are allowed. Each of these property conditions  $p$  points to another object set  $y$  and leads to the following restriction for the members of the set  $x$ : An object can only be member of set  $x$  if it has a defined property  $p$  pointing to an instance that is member of  $y$ . By default, all property conditions of the set are connected conjunctively. This can be changed by a different connection strategy (see below).

This example shows two object sets:



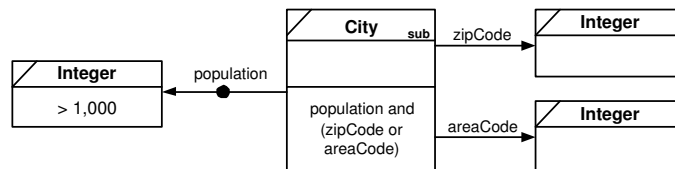
The set on the right hand side contains integer values that are smaller than 1024 (direct condition). The set on the left hand side contains Document objects that have a size which is in the other set (property condition).

The default behavior of a set can be changed by specifying different strategies:

- A *type check strategy* ( $tcs$ ) defines the type the objects in the set have to have. By default, only objects of exactly the given type  $t$  are allowed in the set. This is expressed by the  $tcs$  '='. If also subtypes of  $t$  are allowed, the  $tcs$  **sub[n]** can be used where the optional parameter  $n$  is the maximum distance in the ontology. For example, **sub[1]** means that objects of type  $t$  and  $t$ 's children are allowed. **super** accepts super types of  $t$ . We access the type check strategy of a set  $s$  by  $tcs(s)$ .
- The *connection strategy* ( $cs$ ) changes the way in which the single results of the property conditions are connected. By default, they are connected conjunctively. The  $cs$  is specified as a boolean expressions where the operations

- and**, **or**, and **not** are allowed. We access a single connection operation of the property condition  $p$  by  $cs(p)$ .
- The *missing strategy* ( $ms$ ) specifies the behavior in case of a missing property at an object which is tested for set membership. If for example a property condition for a set of type Document specifies that the document's format has to be pdf or ps, but the current object has not specified that property, the missing strategy decides how to proceed with this object: By default, the missing strategy is *assume\_failed*, which means that in case of a missing property the condition should be regarded as if it had failed. This strategy is depicted by a minus on the property condition arrow (or left blank as it is the default). More strategies are *ignore* depicted by a circle, which skips the condition if it is not defined in the object, and *assume\_fulfilled* depicted by a plus, which means that the condition should be regarded as if it had succeeded. We access the missing strategy of a property condition  $p$  by  $ms(p)$ .

This example shows an object set of Cities:



The strategies change its default semantics: Also objects of subclasses of City are allowed in the set, each object has to have a defined post code *or* area code, and the population has to be larger than 1000 – but objects without a defined population are possible, too.

We have chosen this representation of object sets, because it is very intuitive as it clearly separates different aspects like the conditions themselves, the missing strategy and the connecting strategy. Also, it allows to easily extend or adapt one of the aspects independent of the others and is therefore very suitable for future extensions.

**Variables** are a special kind of object sets which are exclusively needed when describing services. They represent positions where the service offerer or requestor have to insert information before or after service execution. In any case, the information that is inserted has to be a member of the object set, i.e. it has to obey all given conditions. Therefore, variables have the range of the corresponding object set and possibly one single assigned value. Thus, they can be used as if they were an instance. Variables are depicted in the same way as object set, but they use a gray rectangle and have two additional characteristics (see Figure 1, right hand side):

- A *category*. It specifies by whom the information has to be inserted. IN means that the service requestor has to fill in the information, i.e. assign a single value from its range, OUT means that the service offerer will fill in the information.

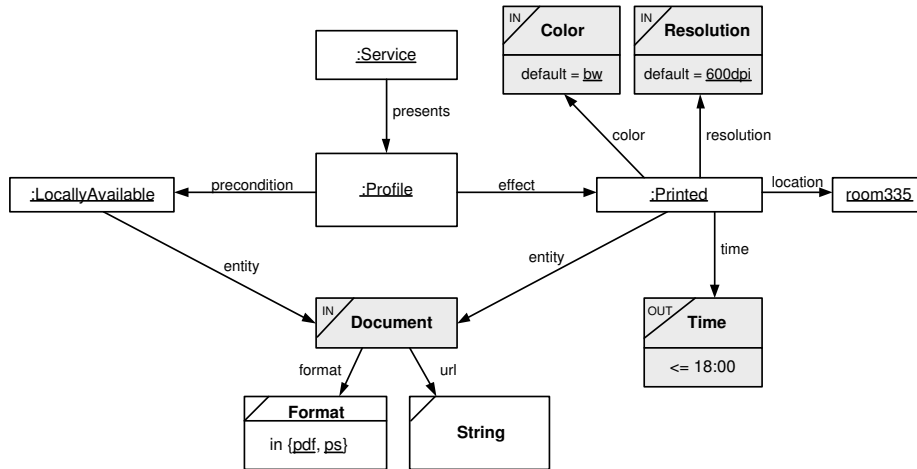


Fig. 2. Example of a state oriented service description of an **offered** printing service.

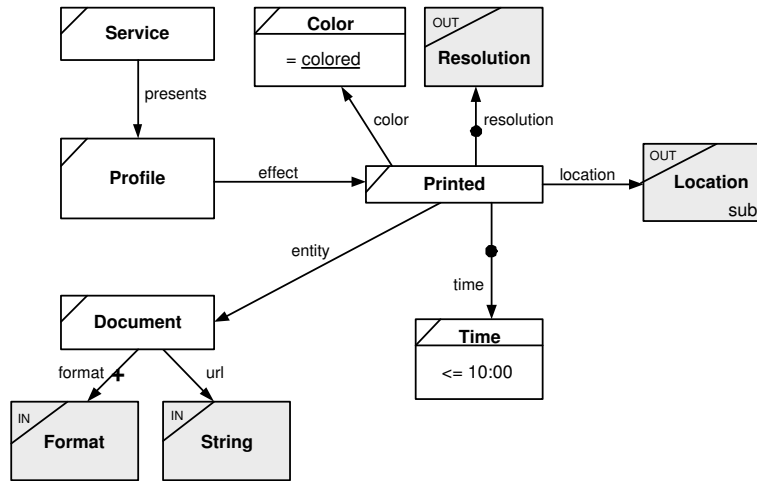
- A *default value*. This value is inserted by default if the service offerer or requestor did not specify a value. Note that the default value has to be a member of the set, too.

### 3.2 Offer Descriptions

The two additional description elements help to overcome the separation of message flow and state transition. Now, it is possible to remove the explicit description of the exchanged messages and integrate it with the help of variables into the states. This leads to state oriented service descriptions (see [1]).

In Figure 2, the offer description of the example printing service is shown as such a state based service description. The input and output properties are removed while the state descriptions are refined by using a common **Document** object. Thus, the described service transforms documents from the state **LocallyAvailable** into the state **Printed**. Moreover, the message flow is integrated as variables (depicted as gray rectangles) in the following manner:

- Values of *incoming messages* (former inputs) are integrated as IN variables of the appropriate type. In the example, inputs are the desired **Color** and **Resolution** of the printout as well as the document itself. It is useful to specify default values for variables, e.g. the default printout will be black-white in a resolution of 600dpi. The two property conditions of the **Document** variable specify that the service expects objects with an arbitrary but defined URL as **String** and with a **format** which is **pdf** or **ps**.
- Values of *outgoing messages* (former outputs) are integrated as OUT variables. They represent the set of objects that can be expected as result when invoking the service. For example, the **Time** variable specifies that the finishing time of the printout is returned, its direct condition says that it will have a value smaller than 18:00.



**Fig. 3.** Example of a state oriented, set based service description of a **request** for a printing service.

As a result, on the one hand, the IN and OUT variables implicitly define the incoming and outgoing messages while their positions within the states clearly expresses their influence on the functionality of the service. Direct and property conditions restrict the possible values.

### 3.3 Request Descriptions

Request descriptions are defined in a similar way. The explicit specification of the message flow is omitted – instead, the desired interface is integrated into the states as variables like above. However, service requests have a different purpose than service offers, which should be reflected in their descriptions. In [12], we showed that a single instance graph is not suitable to describe the functionality desired by the requestor, because it is not possible to include the information what deviation from this perfect service is tolerated by the requestor. We proposed to use sets of objects in request descriptions instead. Thus, the request is an object set of type **Service** where elements of the sets are suitable services.

Figure 3 shows an example request description in a state oriented, set based form. The requestor asks for service objects that have a profile with an effect of type **Printed**. The printed entity should be the document with the url and format defined by the requestor in a concrete service call (IN variable). In any case the printout should be colored and finished before 10 o'clock (direct conditions in the sets of type **Color** and **Time**). After service execution the requestor wants to get informed about the resolution and location of the printout (OUT variables). Also subclasses of **Location** are allowed (type check strategy **sub**).

Note that request descriptions will typically not contain any conditions about the precondition, but only about the desired effects, as the requestor does not know what preconditions could be needed by a concrete offer. Thus, the checking whether the preconditions of a suitable offer can be fulfilled is done after the matching process and omitted in this paper.

## 4 Coupled Signature and Specification Matching

### 4.1 Introduction

The matching process has two tasks: On the one hand, it has to determine if a offer description fits a given request description, on the other hand, it has to assign concrete values to the variables in the descriptions which is necessary to derive the messages needed later. More precisely, the following variables need to be filled:

- All *IN variables of the offer*, which are called *OffIN* variables in the following. They are needed for generating the message which invokes the offered service.
- All *OUT variables of the request*, which are called *ReqOUT* variables in the following. They are needed for providing the desired return values to the service requestor. The requestor can use the missing strategy *assume\_fulfilled* or *ignore* to specify that he is not needing these values necessarily.

The OUT variables of the Offer (*OffOUT*) don't need to be used in cases where the requestor is not interested in them. Also, the IN variables of the request (*ReqIN*) are unproblematic as they are already replaced with concrete values before the matching process starts. For example, if the service requestor wants to use a service `print("http://domain.de/file.pdf", pdf)`, the corresponding values are assigned to the ReqIN variables before a suitable service is searched.

In principle, the matching process has to check a set inclusion: is the offered service (described as single instance) element of the requested service (described as set of instances)? Or more precisely: What assignment of OffIN and ReqOUT variables is necessary so that the offer is included in the set defined by the request? As both descriptions are graphs stemming from similar ontological concepts (= classes), an obvious basic technique for comparing both descriptions is a graph matching approach. Beginning with the root element of type `Service`, the two descriptions are traversed synchronously and compared step by step.

We will present this matching algorithm in several stages. First, we will explain the basic algorithm that simply assumes default strategies and is not capable of dealing with variables in the descriptions (Section 4.2). The problems with variables are analyzed in more detail in Section 4.3. After introducing necessary set operation, we present the full algorithm in Section 4.4.

### 4.2 The Basic Algorithm

The basic algorithm in pseudo code is shown in Listing 1.1. Parts that are denoted in brackets contain code that (a) is not used in the basic version of the

---

```

1  boolean match(r,o)
2  {
3      [TypeCheckStrategy]: if (type(r) != type(o)) return false;
4
5      [OffOUT]
6      [OffIN]
7
8      if (o not fulfills directConditions(r)) return false;
9
10     matches = true;
11
12     for each defined property condition p of r
13     {
14         if (o.p is defined) singleresult = match(r.p, o.p);
15         else [MissingStrategy]: singleresult = false;
16
17         [ConnectingStrategy]: matches = matches && singleresult;
18     }
19
20     [ReqOUT]
21
22     return matches;
23 }

```

---

**Listing 1.1.** Basic matching algorithm

algorithm because it handles variables (Lines 5, 6, and 20) or (b) is kept very simple by only regarding the default strategies (Lines 3, 15, and 17).

The algorithm expects two inputs: the root element  $r$  of a request description as well as the root element  $o$  of an offer description. If  $o$  is a semantically fitting offer for the request  $r$ , the method returns true, otherwise it returns false. Furthermore, the variables are filled in as a side effect by assigning concrete values to them. This is done in the hidden code segments and explained later.

The algorithm is recursive and operates in the following manner: At first, it checks in Line 3, if the two provided elements are of the same type. If not, the comparison fails and false is returned. In the full version, the algorithm has to perform this comparison according to the type check strategy of the set in request. In Line 8, it is checked whether  $o$  satisfies all direct conditions of  $r$ . If not, false is returned.

The loop beginning in Line 12 checks the property conditions of the set  $r$ . If the corresponding property is also defined in the offer  $o$ , the algorithm declines to the values  $r.p$  and  $o.p$  and compares them recursively (Line 14). The result of the single comparison is stored in **singleresult**. If  $p$  is not defined in  $o$ , in this basic version, a matching failure is assumed and false is assigned to **singleresult** (Line 15). In the full version, the algorithm has to handle missing values according to the missing strategy specified by the user.

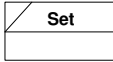
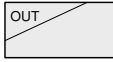
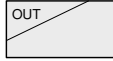
Request r \ Offer o	instance	OUT	IN	-missing-
 or 	Use <b>basic matching</b>	<b>How many</b> values from o are also in r?  all -> true some -> (missing) none -> false  <b>[OffOUT]</b>	Calculate <b>intersection</b> i of values that are in o and in r.  i empty -> false else -> true, assign value x from i to o  <b>[OffIN]</b>	Use <b>missing strategy</b>
additionally for 	matching result: true -> assign o to r  false -> assign null to r	matching result: true -> connect o and r neutral -> assign o later false -> assign null to r	matching result: true -> assign x to r  false -> assign null to r	assign null to r
			<b>[ReqOUT]</b>	

Fig. 4. Problems and solutions while matching variables.

The combination of the single results is done in Line 17. In this basis version, all partial results are combined conjunctively by default. Later, the user defined combination strategy is used. This provides the possibility to use disjunction or negation. After finishing the loop, the calculated result is returned.

Note that the matching process is driven by the structure of the request description, which should not contain any cycles.

### 4.3 Dealing with Variables

This basic version of the matching algorithm can only handle descriptions without variables. Variables are problematic for two reasons:

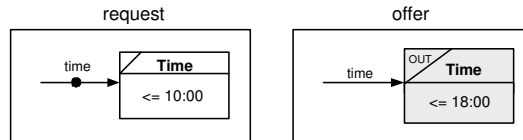
- Variables are undefined parts of the description which leads to problems when matching them.
- ReqOUT and OffIN variables have to be filled with concrete values. This is a prerequisite to allow for automatic service invocation.

The table in Figure 4 analyzes the problems in more detail and shows solutions for them. The elements which can appear in the request  $r$  are listed vertically; the elements of the offer  $o$  are listed horizontally. As the matching process is driven by the structure of the request description, only sets and OUT variables appear here. They can encounter single instances, IN and OUT variables, as well as missing values in the offer. As the ReqOUT variables are sets, too, the matching process is the same as with normal sets. Additionally, it has to be assigned a value to them. Thus, the table is split horizontally: the upper

part is valid for sets and ReqOUT variables, the lower contains the additional parts for the ReqOUT variables.

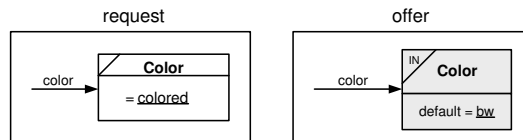
In the following, we will explain three interesting cases from the table which are marked with [OffOUT], [OffIN], and [ReqIN]. The algorithm can be improved by inserting the resulting code at the corresponding tag in the basic version. In the following  $o$  denotes the element from the offer,  $r$  the element from the request:

**[OffOUT]** As the value of the OffOUT variable  $o$  is filled in by the offerer after a successful service execution, it has no assigned value during the matching process. However, as the OffOUT variable is also a set, the offerer has to take a value from this set. Therefore, the matcher has to check if the values of the set  $o$  could be elements of the requested set  $r$ . If all the values of  $o$  are also in  $r$ , the result of the matching result is definitely true, if none of the values of  $o$  is also in  $r$ , the matching result is definitely false. In other cases, i.e. if there are both values of  $o$  in and not in  $r$ , the matching result is undefined – the offerer could return a suitable value, but this is not guaranteed. In this case, the matcher acts conservatively and assumes the value of the OffOUT variable to be missing. Thus, the missing strategy decides the behavior.



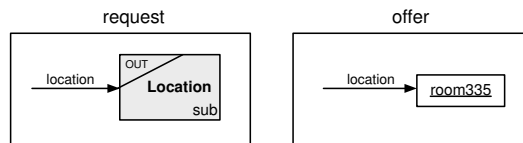
In our example, the Time set in the request is matched against the OUT variable of type Time in the offer. However, the matching result is undefined as the offerer guarantees a finishing time before 6pm while the requestor wants the service to be finished before 10am. Therefore, a missing value is assumed, where the missing strategy *ignore* (black circle on the arrow in the request) decides to skip the property condition completely.

**[OffIN]** As the value of the OffIN variable can be assigned by the requestor himself, it has to be checked if there are values in  $r$  that could also be inserted into  $o$ . Thus, the intersection  $i$  of the sets  $o$  and  $r$  is calculated. If  $i$  is empty, the conditions of requestor and offerer are contradictory, and false is returned as the matching result. However, the default value of  $o$  is assigned to the OffIN variable as the service could be suitable anyway because of a non-standard connecting strategy. If  $i$  is not empty, one arbitrary value from  $i$  is selected and inserted into the OffIN variable  $o$ .



In the example, the Color set with the direct condition '= colored' is matched against the OffIN variable of type Color without conditions. The intersection contains the element colored which is assigned to the OffIN variable  $o$ .

**[ReqOUT]** If  $r$  additionally is a ReqOUT variable, its value is assigned according to the result of the matching process between  $o$  and  $r$ : In case of an unsuccessful matching, null is assigned to  $r$ . This is necessary because a negative matching result of  $r$  and  $o$  does not necessarily lead to an unsuccessful matching result of the whole descriptions since non-default connecting strategies are possible. In case of a successful matching result, if  $o$  has been an OffIN variable, the assigned value is also assigned to  $r$ , in case of an OffOUT variable, the variables are connected, i.e. after a successful service execution the result of the OffOUT variable is assigned to the ReqOUT variable and returned to the service requestor, in case of an instance,  $o$  is assigned to  $r$ .



In our example, the ReqOUT variable of type `Location` is matched against the named instance `room335` of type `Room`. As `Room` is a subtype of `Location`, the match succeeds because of the type check strategy `sub`, and the fixed value `room335` is assigned to the variable. The service requestor will obtain this value after a successful service execution.

As the matching process is guided by the request's structure, it is possible that there are OffIN variables that are not reached by the matcher. However, for a correct service invocation, a concrete value has to be assigned to them. Thus, an arbitrary (preferably the default) value is assigned.

#### 4.4 Needed Set Operations for the Complete Algorithm

The different cases from above show that dealing with variables requires three basic operations on object sets. Given two object sets  $s_1$  and  $s_2$ , the following operations are needed:

- *disjunct*. Is the intersection  $i = s_1 \cap s_2$  empty? I.e., are there any objects that are elements of both sets? We will abbreviate this operation as method `boolean disjunct(Set s1, Set s2)`.
- *subset*. Is  $s_1$  a subset of  $s_2$ ? I.e., is every object in  $s_1$  also in  $s_2$ ? We will abbreviate this operation as method `boolean subset(Set s1, Set s2)`.
- *pickElement*. The task of this operation is to return an arbitrary object  $o$  that is element of the intersection  $s_1 \cap s_2$ . We will abbreviate this operation as method `Instance pickElement(Set s1, Set s2)`.

All these operations are computable with the definition possibilities for sets presented in Section 3.1.

With these possibilities, we can construct the complete algorithm (see Listing 1.2). Its return value is `boolean+` as it can return `true`, `false`, `neutral`, and `missing`. `missing` is only used for partial results and cannot be the result of the whole match.

---

```

1 boolean+ match(r,o)
2 {
3   //TypeCheckStrategy
4   if (not type(r) <tcs(r)> type(o)) return false;
5
6   //OffOUT
7   if (o is OffOUT)
8   {
9     if (disjunct(o,r)) return false;
10    if (subset(o,r)) goto :match;
11    else return missing;
12  }
13
14  //OffIN
15  if (o is OffIN)
16  {
17    if (disjunct(o,r))
18    {
19      o.assign(default(o));
20      return false;
21    }
22    o.assign(pickElement(o,r));
23    goto :match;
24  }
25
26  if (o not fulfills directConditions(r)) return false;
27
28  matches = true;
29  for each defined property condition p of r
30  {
31    if (o.p is defined) singleresult = match(r.p, o.p);
32
33    if ((o.p is undefined) or (singeresult == missing))
34    {
35      //MissingStrategy
36      if (ms(p) == assume_failed) singleresult = false;
37      if (ms(p) == assume_fulfilled) singleresult = true;
38      if (ms(p) == ignore) singleresult = neutral;
39    }
40
41    //ConnectingStrategy
42    matches = matches <cs(p)> singleresult;
43  }
44
45  :match
46
47  //ReqOUT
48  if (r is ReqOUT)
49  {
50    if (matches == true)
51    {
52      if (o is OffIN) r.assign(assignment(o));
53      if (o is OffOUT) r.connect(o);
54      if (o is instance) r.assign(o);
55    }
56    else r.assign(null);
57  }
58
59  return matches;
60 }

```

---

**Listing 1.2.** Complete matching algorithm

## 5 Conclusion

In this paper, we have presented an approach for automatically matching semantic service descriptions. In contrast to existing approaches, it does not rely on an explicit signature matching step, but couples this task with the specification matching. This enables the algorithm to find *all* functionally suitable services even those, whose signatures do not match the request. Moreover, the messages needed for an automatic invocation are derived. To achieve this goal, the matcher compares state based service descriptions. To integrate the message flow into the state descriptions of pre- and postconditions, the concept of declarative object sets and the usage of variables in service descriptions have been introduced.

## References

1. Klein, M., König-Ries, B., Obreiter, P.: Stepwise refinable service descriptions: Adapting DAML-S to staged service trading. In: Proc. of the First Intl. Conference on Service Oriented Computing, Trento, Italy (2003) 178–193
2. Paolucci, M., Kawmura, T., Payne, T., Sycara, K.: Semantic matching of web services capabilities. In: Proc. of the First International Semantic Web Conference, Sardinia, Italy (2002)
3. Sirin, E., Hendler, J., Parsia, B.: Semi-automatic composition of web services using semantic descriptions. In: Proc. of Web Services: Modeling, Architecture and Infrastructure. Workshop in Conjunction with ICEIS2003, Angers, France (2003)
4. Sycara, K.P., Klusch, M., Widoff, S., Lu, J.: Dynamic service matchmaking among agents in open information environments. SIGMOD Record **28** (1999) 47–53
5. Sycara, K., Widoff, S., Klusch, M., Lu, J.: Larks: Dynamic matchmaking among heterogeneous software agents in cyberspace. Autonomous Agents and Multi-Agent Systems **5** (2002) 173–203
6. Trastour, D., Bartolini, C., Gonzalez-Castillo, J.: A semantic web approach to service description for matchmaking of services. In: Proc. of the Intl. Semantic Web Working Symposium (SWWS), Stanford, CA, USA (2001)
7. Gonzalez-Castillo, J., Trastour, D., Bartolini, C.: Description logics for matchmaking services. In: Proc. of the Workshop on Applications of Description Logics at KI-2001, Vienna, Austria (2001)
8. Noia, T.D., Sciascio, E.D., Donini, F.M., Mongiello, M.: A system for principled matchmaking in an electronic marketplace. In: Proc. of the Twelfth Intl. World Wide Web Conference, Budapest, Hungary (2003)
9. Li, L., Horrocks, I.: A software framework for matchmaking based on semantic web technology. In: Proc. of the Twelfth Intl. World Wide Web Conference (WWW 2003), Budapest, Hungary (2003)
10. Li, L., Horrocks, I.: Matchmaking using an instance store: Some preliminary results. In: Proc. of the 2003 Intl. Workshop on Description Logics (DL'2003), poster paper, Rome, Italy (2003)
11. Zaremski, A.M., Wing, J.M.: Specification matching of software components. In: Proc. Of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, ACM Press (1995) 6–17
12. Klein, M., König-Ries, B.: Combining query and preference - an approach to fully automatize dynamic service binding. In: Short Paper at IEEE International Conference on Web Services, San Diego, CA, USA (2004)