A Tool for Behaviour-Based Discovery of Approximately Matching Web Services*,**

Mahdi Sargolzaei[1], Francesco Santini[2],
Farhad Arbab[3], and Hamideh Afsarmanesh[1]

[1] Universiteit van Amsterdam, Amsterdam, Netherlands
{H.Afsarmanesh,M.Sargolzaei}@uva.nl
[2] EPI Contraintes, INRIA - Rocquencourt, France
francesco.santini@inria.fr
[3] Centrum Wiskunde & Informatica, Amsterdam, Netherlands
Farhad.Arbab@cwi.nl

Abstract. We present a tool that is able to discover stateful Web Services in a database, and to rank the results according to a similarity score expressing the affinities between each of them and a user-submitted query. To determine these affinities, we take behaviour into account, both of the user's query and of the services. The names of service operations, their order of invocation, and their parameters may differ from those required by the actual user, which necessitates using similarity scores, and hence the notion of soft constraints. The final tool is based on Soft Constraint Automata and an approximate bisimulation among them, modeled and solved as a Constraint Optimisation Problem.

1 Introduction

Web Services (WSs) [1] constitute a typical example of the *Service Oriented Computing* (SOC) paradigm.WS discovery is the process of finding a suitable WS for a given task. To enable a consumer use a service, its provider usually augments a WS endpoint with an interface description using the *Web Service Description Language* (*WSDL*). In such loosely-coupled environments, automatic discovery becomes even more complex: users' decisions must be supported by taking into account a similarity score that describes the affinity between a user's requested service (the query) and the specifications of actual services available in the considered database.

Although several researchers have tackled this problem and some search tools (e.g., [16]) have achieved good results, very few of them (see Sec. 6) consider the

* This work was carried out during the second author's tenure of the ERCIM "Alain Bensoussan" Fellowship Programme, which is supported by the Marie Curie Cofunding of Regional, National and International Programmes (COFUND) of the European Commission.

** The first and fourth authors are partially supported by the FP7 project GLONET, funded by the European Commission.

R.M. Hierons, M.G. Merayo, and M. Bravetti (Eds.): SEFM 2013, LNCS 8137, pp. 152–166, 2013.

behavioural signature of a service, which describes the sequence of operations a user is actually interested in. This is partly due to the unavoidable limitations of today's standard specifications, e.g., WSDL, which do not encompass such aspects. Despite this, the behaviour of stateful services represents a very important issue to be considered during discovery, to provide users with an additional means to refine the search in such a diverse environment.

In this paper, we first describe a formal framework (originally introduced in [4]) that, during the search procedure, considers both a description of the requested (stateful) service behaviour, and a global similarity score between services and queries. This underlying framework consists of *Soft Constraint Automata* (*SCA*), where semiring-based soft constraints (see Sec. 2) enhance classical (not soft) CA [5] with a parametric and computational framework that can be used to express the optimal desired similarity between a query and a service.

The second and the main contribution of the work reported in this paper is an implementation of such a framework using approximate bisimulation techniques [11] between two SCA: we implement this inexact comparison between a query and a service as a *Constraint Optimisation Problem* (*COP*), by using JaCoP libraries[1]. In the end, we are able to rank all search results according to their similarity with a proposed query. In this way, we can benefit from off-the-shelf techniques with roots in *Artificial Intelligence* (*AI*), in order to tackle the search complexity over large databases. To evaluate a similarity score we use different metrics to measure the syntactical distance between operations and between parameter names (see Sec. 4), e.g., between "*getWeather*" and "*g_weather*". These values are then automatically cast into soft constraints as semiring values (see Sec. 2), with the purpose of being parametrically composed and optimised for the sake of discovery. Thus, a user may eventually choose a service that adheres to her/his needs more than the other ones in a database.

The exploitation of the behaviour during a search process represents the main feature of our tool. SCA represent the formal model we use to represent behaviours: the different states of an SCA represent the different states of a stateful service/query. Relying on SCA allows us to have a framework that comes along with sound operators for composition and hiding of queries [4]. Our plan is to integrate this search tool with the tool presented in [14]. In this comprehensive tool it will be possible, first to search for the desired WSs (or components), and then to compose them into a more complex structured service [14].

The rest of this paper is structured as follows. In Sec. 2 we summarise the background on semiring-based soft constraints [7], as well as the background on SCA [4]. Section 3 shows some examples of how to use SCA to represent the behaviour of services and the similarity between their operation and parameter names. Section 4 describes our tool which implements the search introduced in Sec. 3, while Sec. 5 focuses on how we measure the similarity between two different behavioural signatures. Section 4 introduces the first experimental results evaluating the precision of this tool. In Sec. 6 we report on the related work. Finally, in Sec. 7 we draw our conclusions and explain our future work.

[1] Java Constraint Programming solver (JaCoP): `http://www.jacop.eu`

2 Soft Constraint Automata

Semring-based Soft Constraints. A *c-semiring* [7] (simply semiring in the sequel) is a tuple $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, where A is a possibly infinite set with two special elements $\mathbf{0}, \mathbf{1} \in A$ (respectively the bottom and top elements of A) and with two operations $+$ and $\times$ that satisfy certain properties over A: $+$ is commutative, associative, idempotent, closed, with $\mathbf{0}$ as its unit element and $\mathbf{1}$ as its absorbing element; $\times$ is closed, associative, commutative, distributes over $+$, $\mathbf{1}$ is its unit element, and $\mathbf{0}$ is its absorbing element. The $+$ operation defines a partial order $\leqslant_S$ over A such that $a \leqslant_S b$ iff $a + b = b$; we say that $a \leqslant_S b$ if b represents a value *better* than a. Moreover, $+$ and $\times$ are monotone on $\leqslant_S$, $\mathbf{0}$ is the min of the partial order and $\mathbf{1}$ its max, $\langle A, \leqslant_S\rangle$ is a complete lattice and $+$ is its *least upper bound* operator (i.e., $a + b = lub(a, b)$) [7].

Some practical instantiations of the generic semiring structure are the *boolean* $\langle\{false, true\}, \vee, \wedge, false, true\rangle$, *fuzzy* $\langle[0..1], \max, \min, 0, 1\rangle$, *probabilistic* $\langle[0..1], \max, \hat{\times}, 0, 1\rangle$ and *weighted* $\langle\mathbb{R}^+ \cup \{+\infty\}, min, \hat{+}, \infty, 0\rangle$ (where $\hat{\times}$ and $\hat{+}$ respectively represent the arithmetic multiplication and addition).

A *soft constraint* [7] may be seen as a constraint where each instantiation of its variables has an associated preference. An example of two constraints defined over the *Weighted* semiring is given in Fig. 2. Given $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$ and an ordered finite set of variables V over a domain D, a soft constraint is a function that, given an assignment $\eta : V \to D$ of the variables, returns a value of the semiring, i.e., $c : (V \to D) \to A$. Let $\mathcal{C} = \{c \mid c : D^{|I \subseteq V|} \to A\}$ be the set of all possible constraints that can be built starting from S, D and V: any function in $\mathcal{C}$ depends on the assignment of only a (possibly empty) finite subset I of V, called the *support*, or *scope*, of the constraint. For instance, a binary constraint $c_{x,y}$ (i.e., $\{x, y\} = I \subseteq V$) is defined on the support $supp(c) = \{x, y\}$. Note that $c\eta[v = d]$ means $c\eta'$ where η' is η modified with the assignment $v = d$. Note also that $c\eta$ is the application of a constraint function $c : (V \to D) \to A$ to a function $\eta : V \to D$; what we obtain is, thus, a semiring value $c\eta = a$.The constraint function $\bar{a}$ always returns the value $a \in A$ for all assignments of domain values, e.g., the $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ functions always return $\mathbf{0}$ and $\mathbf{1}$ respectively.

Given the set $\mathcal{C}$, the combination function $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$ [7]; $supp(c_1 \otimes c_2) = supp(c_1) \cup supp(c_2)$. Likewise, the combination function $\oplus : \mathcal{C} \oplus \mathcal{C} \to \mathcal{C}$ is defined as $(c_1 \oplus c_2)\eta = c_1\eta + c_2\eta$ [7]; $supp(c_1 \oplus c_2) = supp(c_1) \cup supp(c_2)$. Informally, $\otimes/\oplus$ builds a new constraint that associates with each tuple of domain values for such variables a semiring element that is obtained by multiplying/summing the elements associated by the original constraints to the appropriate sub-tuples. The partial order $\leqslant_S$ over $\mathcal{C}$ can be easily extended among constraints by defining $c_1 \sqsubseteq_S c_2 \iff \forall\eta, c_1\eta \leqslant_S c_2\eta$.

The search engine of the tool we present in Sec. 4 relies on the solution of *Soft Constraint Satisfaction Problems* (*SCSPs*) [7], which can be considered as COPs. An SCSP is defined as a quadruple $P = \langle S, V, D, C\rangle$, where S is the adopted semiring, V the set of variables with domain D, and C is the constraint set. $Sol(P) = \bigotimes C$ collects all solutions of P, each associated with a similarity value $s \in S$. Soft constraints are also used to define SCA (see Sec 2).

Soft Constraint Automata. Constraint Automata were introduced in [5] as a formalism to describe the behaviour and possible data flow in coordination models (e.g., Reo [5]); they can be considered as acceptors of *Timed Data Streams* (*TDS*) [3,5]. In [4] we paved the way to the definition of *Soft Constraint Automata* (SCA), which represent the theoretical fundament behind our tool.

SCA [4] use a finite set $\mathcal{N}$ of names, e.g., $\mathcal{N} = \{n_1, \ldots, n_p\}$, where n_i ($i \in 1..p$) is the i-th input/output port. The transitions of SCA are labeled with pairs consisting of a non-empty subset $N \subseteq \mathcal{N}$ and a soft (instead of crisp as in [5]) data-constraint c. Soft data-constraints can be viewed as an association of data assignments with a preference for that assignment. Formally,

Definition 1 (Soft Data-Constraints). *A soft data-constraint is a function* $c : (\{d_n \mid n \in N\} \to \mathit{Data}) \to A$ *defined over a semiring* $S = \langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, *where* $\{d_n \mid n \in N\} \to \mathit{Data}$ *is a function that associates a data item with every variable* d_n *related to port name* $n \in N \subseteq \mathcal{N}$, *and* Data *is the domain of data items that pass through ports in* $\mathcal{N}$. *The grammar of soft data-constraints is:*

$$c_{\{d_n \mid n \in N\}} = \bar{\mathbf{0}} \mid \bar{\mathbf{1}} \mid c_1 \oplus c_2 \mid c_1 \otimes c_2$$

where $\{d_n \mid n \in N\}$ *is the support of the constraint, i.e., the set of variables (related to port names) that determine its preference.*

Informally, a soft data-constraint is a function that returns a preference value $a \in A$ given an assignment for the variables $\{d_n \mid n \in N\}$ in its support. In the sequel, we write $SDC(N, \mathit{Data})$, for a non-empty subset N of $\mathcal{N}$, to denote the set of soft data-constraints. We will use SDC as an abbreviation for $SDC(\mathcal{N}, \mathit{Data})$. Note that in Def. 1 we assume a global data domain *Data* for all names, but, alternatively, we can assign a data domain Data_n for every variable d_n.

We state that an assignment η for the variables $\{d_n \mid n \in N\}$ satisfies c with a preference of $a \in A$, if $c\eta = a$.

In Def. 2 we define SCA. Note that by using the *boolean* semiring, thus within the same semiring-based framework, we can exactly model the "crisp" data-constraints presented in the original definition of CA [5]. Therefore, CA are subsumed by Def. 2. Note also that weighted automata, with weights taken from a proper semiring, have already been defined in the literature [10]; in SCA, weights are determined by a constraint function instead.

Definition 2 (Soft Constraint Automata). *A Soft Constraint Automaton over a domain* Data, *is a tuple* $\mathcal{T}_S = (\mathcal{Q}, \mathcal{N}, \longrightarrow, \mathcal{Q}_0, S)$ *where* i) S *is a semiring* $\langle A, +, \times, \mathbf{0}, \mathbf{1}\rangle$, ii) $\mathcal{Q}$ *is a finite set of states,* iii) $\mathcal{N}$ *is a finite set of names,* iv) $\longrightarrow$ *is a finite subset of* $\mathcal{Q} \times 2^{\mathcal{N}} \times SDC \times \mathcal{Q}$, *called the transition relation of* $\mathcal{T}_S$, *and* v) $\mathcal{Q}_0 \subseteq \mathcal{Q}$ *is the set of initial states. We write* $q \xrightarrow{N,c} p$ *instead of* $(q, N, c, p) \in \longrightarrow$. *We call* N *the name-set and* c *the guard of the transition. For every transition* $q \xrightarrow{N,c} p$ *we require that i)* $N \neq \varnothing$, *and ii)* $c \in SDC(N, \mathit{Data})$ *(see Def. 1).* $\mathcal{T}_S$ *is called finite iff* $\mathcal{Q}, \longrightarrow$ *and the underlying data-domain Data are finite.*

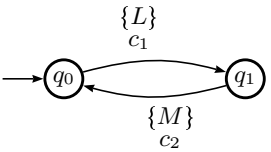

Fig. 1. A Soft Constraint Automaton

$c_1 : (\{d_L\} \to \mathbb{N}) \to \mathbb{R}^+$ s.t. $c_1(d_L) = d_L + 3$

$c_2 : (\{d_M\} \to \mathbb{N}) \to \mathbb{R}^+$ s.t. $c_2(d_M) = d_M + 5$

Fig. 2. c_1 and c_2 in Fig 1

The intuitive meaning of an SCA $\mathcal{T}_S$ as an operational model for service queries is similar to the interpretation of labeled transition systems as formal models for reactive systems. The states represent the configurations of a service. The transitions represent the possible one-step behaviour, where the meaning of $q \xrightarrow{N,c} p$ is that, in configuration q, the ports in $n \in N$ have the possibility of performing I/O operations that satisfy the soft guard c and that leads from configuration q to p, while the ports in $\mathcal{N} \backslash N$ do not perform any I/O operation. Each assignment of variables $\{d_n \mid n \in N\}$ represents the data associated with ports in N, i.e., the data exchanged by the I/O operations through ports in N.

In Fig. 1 we show an example of a (deterministic) SCA. In Fig. 2 we define the *weighted* constraints c_1 and c_2 that describe the preference (e.g., a monetary cost) for the two transitions in Fig. 1, e.g., $c_1(d_L = 2) = 5$.

In [4] we have also softened the synchronisation constraints associated with port names in $\mathcal{N}$ over the transitions. This allows for different service operations to be considered somehow similar for the purposes of a user's query. Note that a similar service can be used, e.g., when the "preferred" one is down due to a fault, or when it offers bad performances, e.g., due to the high number of requests. Definition 3 formalises the notion of soft synchronisation-constraint.

Definition 3 (Soft Synchronization-constraint). *A soft synchronization-constraint is a function* $c : (V \to \mathcal{N}) \to A$ *defined over a semiring* $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$*, where* V *is a finite set of variables for each I/O ports, and* $\mathcal{N}$ *is the set of I/O port names of the SCA.*

3 Representing the Behaviour of Services with SCA

In this section we show how the formal framework presented in Sec. 2 (e.g., SCA) can be used to consider a similarity score between a user's query and the service descriptions in a database, in oder to find the best possible matches for the user.

We begin by considering how parameters of operations can be associated with a score value that describes the similarity between a user's request and an actual service description in a database. We suppose to have two different queries: the first, `getByAuthor(Firstname)`, which is used to search for conference papers using the `Firstname` (i.e., the parameter name) of one of its authors; the name of the invoked service operation is, thus, `getByAuthor`. The second query, `getByTitle(Conference)`, searches for conference papers, using the title of the `Conference` wherein the paper has been published; the name of the invoked operation is `getByTitle`. These two queries are represented as the SCA

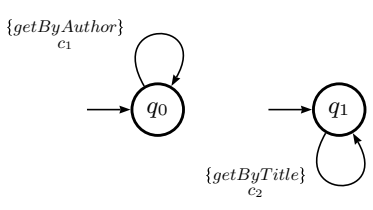

Fig. 3. Two soft Constraint Automata representing two different queries

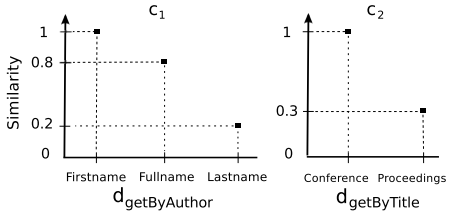

Fig. 4. The definitions of c_1 and c_2 in Fig. 3

(see Sec. 2) q_0 and q_1, in Fig. 3. Soft constraints c_1 and c_2 in Fig. 4, define a similarity score between the parameter name used in a query and all parameter names in the database (for the same operation name, i.e., either `getByAuthor` or `getByTitle`). These similarity scores can be modeled with the *fuzzy* semiring $\langle [0..1], \max, \min, 0, 1 \rangle$ wherein the aim is to maximise the similarity ($+ \equiv \max$) between a request and a service returned as a matching result. Constraint c_1 in Fig. 4 states that similarity is full if a `getByAuthor` operation in the database takes `Firstname` as parameter (since 1 is the top preference of the *fuzzy* semiring), less perfect, that is 0.8, if it takes `Fullname` (usually, `Fullname` includes `Firstname`), or even less perfect, that is 0.2, if it takes `Lastname` only. Similar considerations apply to the operation name `getByTitle` (see Fig. 3) and c_2 in Fig. 4. Similarity scores are automatically extracted as explained in Sec. 4.

Suppose now that our database contains the four services represented in Fig. 5. All these services are stateless, i.e., their SCA have a single state each. For instance, service a has only one invocable operation whose name is `getByAuthor`, which takes `Lastname` as parameter. Service d has two distinct operations, `getByAuthor` and `getByTitle`.

According to the similarity scores expressed by c_1 and c_2 in Fig. 4, queries q_0 and q_1 in Fig. 3 return different result values for each operation/service, depending on the instantiation of variables $d_{getByAuthor}$ and $d_{getByTitle}$. Considering q_0, services a, b, and d have respective preferences of 0.2, 1, and 0.8. If query q_1 is used instead, the possible results are operations c and d, with respective preferences of 1 and 0.3. When more than one service is returned as the result of a search, the end user has the freedom to choose the best one according to his preferences: for the first query q_0, the user can opt for service b, which corresponds to a preference of 1 (i.e., the top preference), while for query q_1 the user can opt for c (top preference as well).

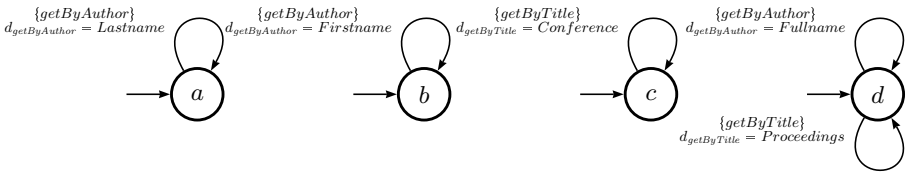

Fig. 5. A database of services for the queries in Fig. 3; d perfoms both kinds of search

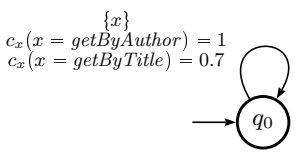

Fig. 6. A similarity-based query for the *Author / Title* example

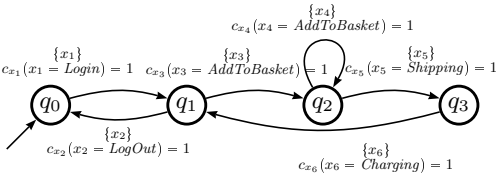

Fig. 7. A similarity-based query for the on-line purchase service

We now move from parameter names to operation names, and show that by using soft synchronisation constraints (see Def. 3), we can also compute a similarity score among them. For example, suppose that a user queries q_0 in Fig. 3. The possible results are services a, b and d in the database of Fig. 5, since service c has an operation named `getByTitle`, different from `getByAuthor`. However, the two services are somehow similar, since they both return a paper even if the search is based either on the author or on the conference. As a result, a user may be satisfied also by retrieving (and then, using) service c. This can be accomplished with the query in Fig. 6, where $c_x(x = getByAuthor) = 1$, and $c_x(x = getByTitle) = 0.7$. Note that we no longer deal with constraints on parameter names, but on operation names. Then, we can also look for services that have similar operations, not only similar parameters in operations.

However, the main goal of this paper is to compute a similarity score considering also the behaviour of queries and services. For instance (the query in Fig. 7), a user may need to find an on-line purchase service satisfying the following requirements: *i*) charging activity comes before shipping activity, *ii*) to purchase a product, the requester first needs to log into the system and finally log out of the system, and *iii*) filling the electronic basket of a user may consist of a succession of "add-to-basket" actions. In Sec. 5 we will focus on this aspect.

Constraints on parameter (their data-types as well) and operation names can be straightforwardly mixed together to represent a search problem where both are taken into account simultaneously for optimization. The tool in Sec. 4 exploits this kind of search: the similarity functions represented by constraints are computed through the composition of different syntactic similarity metrics.

4 Tool Description

Conceptually, our behaviourally-based WS discovery proceeds in four successive steps: *i*) generate a *Web Service Behaviour Specification* (*WSBS*) for each registered WS (a WSBS is basically a CA), *ii*) process preference-oriented queries (basically represented as SCA), *iii*) model an approximate bisimulation between a query and our services as an SCSP (see Sec. 2), and finally, *iv*) solve this problem (see Sec. 2). Note that we are also able to translate other kinds of behavioural service specification, as WS-BPEL[2], into (S)CA [8].

[2] WS-Business Process Execution Language, 2.0: `http://tinyurl.com/czkoolw`

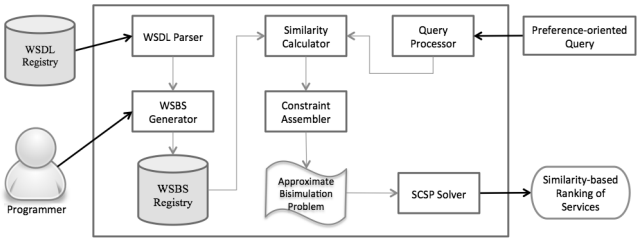

Fig. 8. General architecture of the tool

Step *i* is needed because no standard language or tool exists to specify the behaviour of stateful WSs. Therefore, we have to define our internal WSBS as a behavioural specification for WSs, using WSDL and some extra necessary annotations. In step *ii*, we obtain a query from a user and we process it to find the similarities between the request and the actual services in the database. In the third step, we set up an SCSP (see Sec. 2), where soft-constraint functions are assembled by using the similarity scores derived from step *ii*; at the same time, we define those constraints that compare the two behavioural signatures (query/service), and measure their similarity. Finally, we find the best solutions for this SCSP, and we return them to the user. All these steps are implemented by different software modules, whose global architecture is defined in Fig. 8.

WSDL Parser. We rely on a repository of WSDL documents that are captured in a registry, i.e., the *WSDL Registry* (see Fig. 8). WSDL is an XML-based standard for syntactical representation of WSs, which is currently the most suitable for our purpose. First, we parse these XML-based documents to extract the names and interfaces of service operations using the *Axis2* technology.[3]

WSBS Generator. While a WSDL document specifies the syntax and the technical details of a service interface, it lacks the information needed to convey its behavioural aspects. In fact, a WSDL document only reveals the operation names and the names and data types of their arguments. Hence, we must indicate the permissible operation sequences of a service. If we know that a WS is stateless, then all of its operations are permissible in any order. For a stateful service, however, we need to know which of its operations is (not) allowed in each of its states. In [14], some of the authors of this paper have already formalised the behaviour of a WS (i.e., the WSBS) in terms of CA [5]. Therefore, we adopt the *Extensible Coordination Tools* (*ECT*) [2], which consist of a set of plug-ins for the *Eclipse* platform[4], as the core of the *WSBS Generator*, in order to generate a CA to specify the externally observable behaviour of a service. Normally, the ECT is used to give a semantics to Reo circuits [5]. The resulting CA are captured as XML documents, where the <*states*> and <*transitions*> tags identify the structure of each automaton. It is also possible to indicate the

[3] `http://axis.apache.org/axis2/java/core/`
[4] ECT webpage: `http://reo.project.cwi.nl/reo/wiki/Tools`

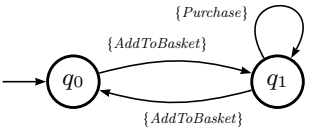

Fig. 9. An example of WSBS

```
q0 AddToBasket q1;
q1 AddToBasket q1;
q1 Purchase q0.
```

Fig. 10. Text file representing the WSBS in Fig. 9

behaviour of WSs in text files, in a simplified form. The file in Fig. 10 describes the service represented in Fig. 9. In our architecture, all WSBSs are stored in a *WSBS Registry* (see Fig. 8).

We can automatically extract a single-state automaton from the operations defined in a WSDL document describing a stateless WS: we use this support-tool to extract the automata for the real-world WSs used in our following experiment. For stateful WSs, we developed an interactive tool that (using a GUI) allows a programmer (see Fig. 8) to visually create the automaton states describing the behavior of a service, and tag its transitions with the operations defined in its WSDL document.

Query Processor. At search time, a user specifies a desired service by means of a text file, and feeds it to this module. An example of our query is represented in Fig. 11. The query format allows to specify all desired transitions among states, including operation names, and the names and data types of their arguments. It enables to search for multiple similar services (separated by "or" operators) at the same time while the tool ranks all the results in the same list. Finally, the tool assigns to each service description a preference score prescribed by the user. A user may use a score (e.g., fuzzy preferences in [0..1]) to weigh all the results, as represented in Fig. 11. Each query is represented as an SCA [4] (see Sec. 2), since preferences can be represented by soft constraints. This textual representation resembles a list of WSBSs, each of them associated with a preference score (see Fig. 11 and Fig. 10 for a comparison).

Similarity Calculator. As Fig. 8 shows, this module requires two inputs: the WSBSs and the processed query. It returns three different kinds of similarity scores, which reflect the similarities between one service and one query *i*) operations names, *ii*) names of input-parameters of operations, and *iii*) data types of input-parameters. We use different string similarity-metrics (also known as *string distance functions*) as the functions to measure the similarity between two text-strings. We have chosen three of the most widely known metrics, including the *Levenshtein Distance*, the *Matching Coefficient*, and the *QGrams Distance*. Each of these metrics operates with two input strings, and return a score estimating their similarity. Since each function returns a value in [0..1], we average these three scores to merge them into a single value still in [0..1].

These similarity scores are subsequently used by the *Constraint Assembler* in Fig. 8, in order to define the similarity functions that are translated into soft constraints, as explained in Sec. 3. The representation of the search problem

```
q0 Weather(City:string) q0, [1.0] or q0 Weather(Zipcode:string) q0, [0.8]
```

Fig. 11. A single-state query asking for the weather conditions over a `City`, or using a `Zipcode`. Different user's preference scores are represented within square brackets.

in terms of constraints is completely constructed by the *Constraint Assembler* module, while the *Similarity Calculator* only provides it with similarity scores.

Constraint Assembler. This module produces a model of the discovery problem, in the form of approximate bisimulation (see Sec. 5), as an SCSP (see Sec. 2). To do so, it represents all preference and similarity requirements as soft constraints. In order to assemble these constraints, we used *JaCoP*, which is a Java library that provides a finite-domain constraint programming paradigm. We have made ad-hoc extensions to the crisp constraints offered by JaCoP in order to equip them with weights, and we have exploited the possibility to minimise/maximise a given cost function to solve SCSPs. Specifically, we have expressed the WSs discovery problem as a fuzzy optimisation problem, by implementing the *fuzzy* semiring, i.e., $\langle [0..1], \max, \min, 0, 1 \rangle$ (see Sec. 2).

For instance, *SumWeight* is a JaCoP constraint that computes a weighted sum as the following pseudo-code: $w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 = sum$, where sum represents the global syntactic similarity between two operation names (x_1), considering also their argument names (x_2) and types (x_3). These scores are provided by the *Similarity Calculator*. Moreover, we can tune the weights w_1, w_2, and w_3 to give more or less importance to the three different parameters. In the experiments in Sec. 4 we use equal weights. We leave to Sec. 5 a discussion on how to compute how much two behavioural signatures (query/service) are similar, and how the general constraint-based model is designed.

SCSP Solver. Finally, after the specification of the model consisting of variables and constraints, a search for a solution of the assembled SCSP can be started. This represents the final step (see Fig. 8). The result can be generalised as a ranking of services in the considered database: at the top positions we find the services that are more similar to a user's request.

Experimental Results on a Stateless Scenario. In this section we show the precision results of our tool through a scenario involving stateless real WSs. Figure 11 shows a single-state query that searches for WSs that return the "weather" forecast for a location indicated by the name of a "city" (with a user's preference of 1) or its "zip-code" (preference of 0.8). We retrieved 14 different WSDL documents by querying the word "Weather" on *Seekda*[5], which is a public WS search-engine. These documents list a total of 58 different operations, which populate our *WSDL Registry* (see Fig. 8).

Table 1 reports a part of the experiment results. From left to right the columns respectively report the position in the final ranking, the obtained fuzzy score, the WS name, and, lastly, the matched service operation.

[5] `http://webservices.seekda.com`

5 On Comparing Behaviour Signatures

In this section we zoom inside the *Constraint Assembler* component that we introduced in Sec. 4. We describe how we can approximate the behaviour of a posed query with that of a service, since a perfect match can be uncommon.

The basic idea is to compute an approximate bisimulation [11] between the two automata respectively representing a query, and a WS in a database. The notion of approximate bisimulation relation is obtained by relaxing the equality of output traces: instead of requiring them to be identical, we require that they remain "close". Metrics (represented as semirings, in our case) essentially quantify how well a system is approximated by another based on the distance between their observed behaviours. In this way, we are able to consider different transition-labels by estimating a similarity score between their operation interfaces, and different numbers of states. To model approximate bisimulation with constraints, we exploited constraint-based graph matching techniques [17]; thus, we are able to "compress" or "dilate" one automaton structure into another.

In the following, we use the query example in Fig 12, and the service example in Fig. 13 to describe our constraint-based model for the search. We subdivide this description by considering how we match the different elements of automata (transitions or states), and how we finally measure their overall similarity.

States. To represent our signature-match problem, for each of the query-automaton states (cardinality Q) we define a variable that can be assigned to one or several states of a service (cardinality S). For this purpose, we use *SetVar*, i.e., JaCoP variables defined as ordered collections of integers. Considering our running example, one of the possible matches between these two signatures can be given by $\mathcal{M} \equiv q_0 = \{s_0, s_1, s_3\}, q_1 = \{s_2\}$. This matching is represented in Fig. 12 and Fig. 13 using gray and black labels for states. Clearly, the proposed modelling solution represents a relationship and not a function, since a query state can be associated with one or more service-states; on the other hand, different query states can be associate with the same service state, in case a query has more states than a service. Thus, to match the two automata we allow to "merge" together those states that are connected by a transition (e.g., s_0, s_1

Table 1. The ranking of the top-ten matched WSs, based on the query represented in Fig. 11, out of a database of 14 different WSDL documents

Rank	Score	Name of WS	Interface of the operation
1	0.82	weather	$GetWeather(City : string)$
2	0.69	globalweather	$GetWeather(CityName : string)$
3	0.5	Weather	$Get\_Weather(ZIP : string)$
4	0.48	WeatherWS	$getWeather(theCityCode : string, theUserID : string)$
5	0.47	WeatherWebService	$getWeatherbyCityName(theCityName : string)$
6	0.44	usweather	$GetWeatherReport(ZipCode : string)$
7	0.42	WeatherForecast	$GetWeatherByZipCode(ZipCode : string)$
8	0.4	WeatherForecast	$GetWeatherByPlaceName(PlaceName : string)$
9	0.4	weatherservice	$GetLiveCompactWeathe(cityCode : string, ACode : string)$
10	0.36	weatherservice	$GetLiveCompactWeatherByStationID(stationid : string, un : UnitType, ACode : string)$

and s_3 in Fig. 13) into a single state (e.g., q_0) at the cost of incurring a certain penalty.

Transitions. In our running example, if we match the two behaviours as defined by $\mathcal{M}$, we consequently obtain a match for the transitions (and their labels) as well. Our model has a variable (*IntVar*, in JacoP) for each of the transitions in a query automaton; considering the example in Fig. 12, we have three variables l_1, l_2, l_3. In Fig. 12 and Fig. 13 we label each transition with its identifier ($l_1, \ldots, l_3, m_1, \ldots, m_5$), and a string that represents its related operation-name (in this example, we ignore parameter names and types for the sake of brevity). Thus, the full match-characterisation is now $\mathcal{M} \equiv q_0 = \{s_0, s_1, s_3\}, q_1 = \{s_2\}, l_1 = m_2, l_2 = m_3, l_3 = m_5$. Note that, if a query has more transitions than a service, it may happen to be impossible to match all of them; for this reason, since we need to assign each of the variables in order to find a solution, we assign a mark *NM* (i.e., *Not Matched*) to unpaired transitions.

Match Cost. In this paragraph we show how to compute a global similarity score Γ for a match $\mathcal{M}$ (i.e., $\Gamma(\mathcal{M})$). We consider two different kinds of scores, *i*) a state similarity-score, $\sigma(\mathcal{M})$, is derived from how much we need to (de)compress the behaviour (in terms of number of states) to pass from one signature to another, and *ii*) a transition similarity-score, $\theta(\mathcal{M})$, is derived from a comparison between matched labels. In a simple case, we can consider the mean value $\Gamma(\mathcal{M}) = (\sigma(\mathcal{M}) + \theta(\mathcal{M}))/2$, or we can imagine more sophisticated aggregation functions. A rather straightforward function is $\sigma(\mathcal{M}) = \min(\#\mathsf{S}_{\mathcal{M}}, \#\mathsf{Q}_{\mathcal{M}})/\max(\#\mathsf{S}_{\mathcal{M}}, \#\mathsf{Q}_{\mathcal{M}})$ (if $\#\mathsf{S}_{\mathcal{M}} = \#\mathsf{Q}_{\mathcal{M}}$, our match is perfect), but we can think of non-linear functions as well, for instance. The score $\theta(\mathcal{M})$ is computed by aggregating the individual *ssim* syntactic similarity-scores (computed by the *Similarity Calculator* in Sec. 4) obtained for each label match, and then averaging on the number of matched labels. For our example, $\theta(\mathcal{M}) = (ssim(label_{l_1}, label_{m_2}) + ssim(label_{l_2}, label_{m_4}) + ssim(label_{l_3}, label_{m_5}))/3$.

An Experiment with Stateful Services. Since all current WS standards are stateless, for this experiment we hand-crafted four stateful WSs (see Tab. 2). We use the following query against this database. The ideal service matching the query first retrieves the weather forecast for a city based on its name, and then retrieves the forecasts for its neighbouring cities:

`q0 GetWeather(SetCity : string) q1; q1 GetNeighbourhoodWeather() q0 [1.0].`

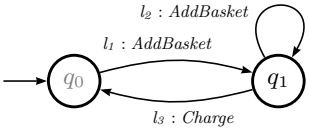

Fig. 12. A query example

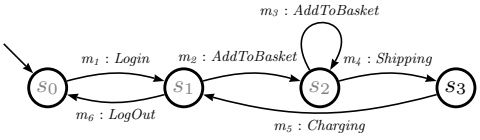

Fig. 13. A possible service in a database related to the query in Fig. 12

Table 2. Our registry of hand-crafted stateful WSs, and the obtained similarity scores

ID	WSBS	θ	σ	Γ	Rk
S_1	q0 getweather(city:string) q0	.76	.5	.63	4
S_2	q0 getweather(city:string) q1 ; q1 getneighborsweather q0	.8	1	.9	1
S_3	q0 login(password:string) q1; q1 getweather(city:string) q2 ; q2 getneighborsweather q0	.8	.66	.73	3
S_4	q0 GetWeather(myCity:string) q1; q1 getNeighWeather q0	.69	1	.84	2

Table 2 shows the results of this experiment: the transition similarity-score $\theta(\mathcal{M})$, the state similarity-score $\sigma(\mathcal{M})$, the global similarity-score $\Gamma(\mathcal{M})$, and the rank *Rk* of each service. These results match our expectations, since the behaviour of S_2 and S_4 each is identical to the behaviour of our query, while the operation interface of S_2 is more similar to the query compared to that of S_4.

6 Related Work

Compared to the work reported in the literature, the solution in this paper seems more general, compact, and comprehensive, because it can encompass any semiring-like metrics, and the whole framework is expressively modeled and solved using Constraint Programming. Moreover, elaborating on a formal framework allows us to easily check properties of services/queries (e.g., to model-check or bi/simulate them [4]), and to have join and hide operators to work on them [4]. Most of the literature seems to report more ad-hoc engineered and specific solutions, instead, which consequently, are less amenable to formal reasoning.

In [18] the authors propose a new behaviour model for WSs using automata and logic formalisms. Roughly, the model associates messages with activities and adopts the IOPR model (i.e., Input, Output, Precondition, Result) in *OWL-S*[6] to describe activities. The authors use an automaton structure to model service behaviour. However, similarity-based search is not mentioned in [18]. In [21] the authors present an approach that supports service discovery based on structural and behavioural service models, as well as quality constraints and contextual information. Behaviours are matched through a subgraph isomorphism algorithm. In [12] the problem of behavioural matching is translated to a graph matching problem, and existing algorithms are adapted for this purpose.

The model presented in [19] relies on a simple and extensible keyword-based query language and enables efficient retrieval of approximate results, including approximate service compositions. Since representing all possible compositions can result in an exponentially-sized index, the authors investigate clustering methods to provide a scalable mechanism for service indexing.

In [6], the authors propose a crisp translation from interface description of WSs to classical crisp *Constraint Satisfaction Problems* (*CSPs*). This work does not consider service behaviour and it does not support a quantitative reasoning on similarity/preference involving different services. In [20], a semiring-based framework is used to model and compose QoS features of WSs. However, no notion of similarity relationship is given in [20].

[6] OWL-S: Semantic Markup for Web Services, 2004: `www.w3.org/Submission/OWL-S/`

In [9], the authors propose a novel clustering algorithm that groups names of parameters of WS operations into semantically meaningful concepts. These concepts are then leveraged to determine similarity of inputs (or outputs) of web-service operations. In [15] the authors propose a framework of fuzzy query languages for fuzzy ontologies, and present query answering algorithms for these query languages over fuzzy *DL-Lite* ontologies. In [13] the authors propose a metric to measure the similarity of semantic services annotated with an *OWL ontology*. They calculate similarity by defining the intrinsic information value of a service description based on the "inferencibility" of each of *OWL Lite* constructs. The authors in [16] show a method of WS retrieval called *URBE* (*UDDI Registry By Example*). The retrieval is based on the evaluation of similarity between the interfaces of WSs. The algorithm used in *URBE* combines the analysis of the structure of a WS and the terms used inside it.

7 Conclusions

We have presented a tool for similarity-based discovery of WSs that is able to rank the service descriptions in a database, in accordance with a similarity score between each of them and the description of a service desired by a user. The formal framework behind the tool consists of SCA [4], which can represent different high-level stateful software services and queries. Thus, we can use SCA to formally reason on queries (e.g., bisimulation for SCA is introduced in [4]). The tool is based on implementing approximate bisimulation [11] with constraints (see Sec. 5), which allows to quantitatively estimate the differences between two behaviours. Defining this problem as an SCSP makes it parametric with respect to the chosen similarity metric (i.e., a semiring), and allows using efficient AI techniques for solving it: subgraph isomorphism is not known to be in P.

Our main intent has been to propose a formal framework and a tool with an approximate bisimulation of behaviours at its heart, not to directly compete against tools such as [16], which although show higher precision than what we have summarised in Sec. 4, do not support behaviour specification in their matching. Nevertheless, in the future we plan to refine the performance of our tool by also evaluating a semantic similarity-score between the operation and parameter names, using an appropriate ontology for services as OWL-S.

References

1. Alonso, G., Casati, F., Kuno, H.A., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Data-Centric Systems and Applications. Springer (2004)
2. Arbab, F., Koehler, C., Maraikar, Z., Moon, Y., Proença, J.: Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. Tool demo session at FACS 8 (2008)
3. Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)

4. Arbab, F., Santini, F.: Preference and similarity-based behavioral discovery of services. In: ter Beek, M.H., Lohmann, N. (eds.) WS-FM 2012. LNCS, vol. 7843, pp. 118–133. Springer, Heidelberg (2013)
5. Baier, C., Sirjani, M., Arbab, F., Rutten, J.J.M.M.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. 61(2), 75–113 (2006)
6. Benbernou, S., Canaud, E., Pimont, S.: Semantic web services discovery regarded as a constraint satisfaction problem. In: Christiansen, H., Hacid, M.-S., Andreasen, T., Larsen, H.L. (eds.) FQAS 2004. LNCS (LNAI), vol. 3055, pp. 282–294. Springer, Heidelberg (2004)
7. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM 44(2), 201–236 (1997)
8. Changizi, B., Kokash, N., Arbab, F.: A Unified Toolset for Business Process Model Formalization. In: Proceedings of FESCA 2010 (2010)
9. Dong, X., Halevy, A., Madhavan, J., Nemes, E., Zhang, J.: Similarity search for web services. In: Proceedings of Very Large Data Bases, vol. 30, pp. 372–383, VLDB Endowment (2004), http://dl.acm.org/citation.cfm?id=1316689.1316723
10. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata, 1st edn. Springer Publishing Company, Incorporated (2009)
11. Girard, A., Pappas, G.J.: Approximation metrics for discrete and continuous systems. IEEE Trans. Automat. Contr. 52(5), 782–798 (2007)
12. Grigori, D., Corrales, J.C., Bouzeghoub, M.: Behavioral matchmaking for service retrieval. In: IEEE International Conference on Web Services (ICWS), pp. 145–152. IEEE Computer Society (2006)
13. Hau, J., Lee, W., Darlington, J.: A semantic similarity measure for semantic web services. In: Web Service Semantics Workshop at WWW (2005)
14. Jongmans, S.-S.T.Q., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Automatic code generation for the orchestration of web services with Reo. In: De Paoli, F., Pimentel, E., Zavattaro, G. (eds.) ESOCC 2012. LNCS, vol. 7592, pp. 1–16. Springer, Heidelberg (2012)
15. Pan, J.Z., Stamou, G., Stoilos, G., Taylor, S., Thomas, E.: Scalable querying services over fuzzy ontologies. In: Proceedings of World Wide Web, WWW 2008, pp. 575–584. ACM, New York (2008), http://doi.acm.org/10.1145/1367497.1367575
16. Plebani, P., Pernici, B.: Urbe: Web service retrieval based on similarity evaluation. IEEE Trans. on Knowl. and Data Eng. 21(11), 1629–1642 (2009), http://dx.doi.org/10.1109/TKDE.2009.35
17. le Clément, V., Deville, Y., Solnon, C.: Constraint-based graph matching. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 274–288. Springer, Heidelberg (2009)
18. Shen, Z., Su, J.: Web service discovery based on behavior signatures. In: Proceedings of the 2005 IEEE International Conference on Services Computing, SCC 2005, vol. 01, pp. 279–286. IEEE Computer Society, Washington, DC (2005)
19. Toch, E., Gal, A., Reinhartz-Berger, I., Dori, D.: A semantic approach to approximate service retrieval. ACM Trans. Internet Technol. 8(1) (November 2007)
20. Zemni, M.A., Benbernou, S., Carro, M.: A soft constraint-based approach to QoS-aware service selection. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 596–602. Springer, Heidelberg (2010)
21. Zisman, A., Dooley, J., Spanoudakis, G.: Proactive runtime service discovery. In: Proceedings of the 2008 IEEE International Conference on Services Computing, SCC 2008, vol. 1, pp. 237–245. IEEE Computer Society, Washington, DC (2008), http://dx.doi.org/10.1109/SCC.2008.60