# QoS-aware and Behavior-based Approximate Matching of Stateful Web Services

MAHDI SARGOLZAEI
Universiteit van Amsterdam
NETHERLANDS
M.Sargolzaei@uva.nl

FRANCESCO SANTINI
University of Perugia
ITALY
francesco.santini@dmi.unipg.it

FARHAD ARBAB
CWI*
NETHERLANDS
Farhad.Arbab@cwi.nl

HAMIDEH AFSARMANESH
Universiteit van Amsterdam
NETHERLANDS
H.Afsarmanesh@uva.nl

*Abstract:* We present a tool that is able to discover stateful web services in a database, ranked according to a similarity score expressing the affinities between each service and a user-submitted query. To determine these affinities, we also take behavior 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 user's query, so long as they collectively represent similar behavior. We introduce a light extension of WSDL, namely BWSDL to describe the behavior of web services as well, then we develop a GUI to ease this kind of service specification. We use soft constraints to formalize the requirements that a user expresses in her query. We argue that a proper formalization of the behavior of many services that are commonly thought of as stateless, in fact requires a stateful representation. As such, our method and our tool can accommodate discovery of these services better than alternatives that consider them as stateless. Our tool uses a procedure to asses an approximate operational-similarity score among Soft Constraint Automata, which we use as formal models of behavior. The discovery is modelled as a Constraint Optimization Problem. Finally, we enhance our tool by also considering QoS metrics to further meet user's needs, and we present a peer-to-peer implementation to overcome scalability issues.

*Key–Words:* Weighted argumentation frameworks, Coalition formation, Soft constraint satisfaction problems.

## 1 Introduction

Web services (WSs) [2] 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*[1]). In a loosely-coupled environment as SOC, 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., [39]) have achieved good results, very few of them (see Section 9) consider the behavioral 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 behavior 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 environ-

*Centrum Wiskunde & Informatica

[1]Web Services Description Language: `https://www.w3.org/TR/wsdl`.

ment.

The impact of considering stateful behavior in search of services is indeed broader than it may seem at first. As we argue in Section 2, our notion of stateful services in fact covers a much wider class of services, and includes many of those that are commonly considered stateless.

In this paper, we first describe a formal framework (originally introduced in [5]) that, during a search procedure, considers both a description of the requested (stateful) service behavior, and a global similarity score between services and queries. This underlying framework consists of *Soft Constraint Automata* (*SCA*), where semiring-based soft constraints (see Section 3) enhance classical (not soft) CA [7] with a parametric and computational framework that can be used to express the optimal desired similarity between a query and a service. In our work, we use CA as our base formalism for the specification of WS behavior. Since CA models are supported by our related tool in [20], it can readily be extended to support soft constraints; overall, we believe automata models are more understandable and readable for engineers. However, we do not unavoidably depend on CA: in principle, we can use other formalisms and then convert their behavior to CA.

The second and main contribution of the work reported in this paper is an implementation of such a framework using an approximate operational-similarity evaluation between two SCA: we implement this inexact comparison between a query and a service as a *Constraint Optimization Problem* (*COP*), by using JaCoP libraries[2]. We are eventually 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 complexity of search over large databases. To evaluate a similarity score we use different metrics to measure the syntactical distance between operations and between parameter names (see Section 6), e.g., between "*getWeather*" and "*g_weather*". These values are then automatically cast into soft constraints as semiring values (see Section 3), to enable parametric composition and optimization during the process of discovery. Thus, a user may eventually choose a service that adheres to his needs better than the other ones in a database.

The exploitation of the behavior during a search process represents the main feature of our tool. SCA represent the formal model we use to represent behaviors: 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 [5].

Afterwards, we describe a QoS-based service recommendation besides our discovery tool to assist users in service selection. The reason is that non-functional properties as QoS parameters play an important role in user's selection. By already having scores representing functional similarity, using further QoS metrics in the same framework comes at a reduced cost. We adopt a lexicographic composition of soft constraints to capture the trade-off of preferences in selecting the best fitting WS.

Moreover, we develop a peer-to-peer (P2P) implementation for the presented tool, with the aim to improve performance and show the scalability of our approach. The discovery process can be expensive if computed by a single node, especially in presence of thousands of services. To avoid this, we pass the same query to all the available nodes in a network, and each node computes a list of similarity scores for its own content that match the query, producing partial search results. Each node sends such results along with their computed similarity scores to an aggregator, which merges these rankings from multiple nodes to create a single set of ranked results. We can have a number of aggregators organized in a hierarchical way: the results of aggregators can be merged by an aggregator at a higher level.

Finally, we evaluate the quality of the computed results of our proposed tool using two basic informa-

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

tion retrieval's metrics, namely precision and recall. It is not possible to have a meaningful direct comparison against other existing tools due to i) the lack of widely accepted benchmarks using stateful services and ii) the dearth of tools working with such services. Nevertheless, we provide a preliminary study on the precision of our discovery process.

The rest of this paper is structured as follows. Section 2 illustrates behavior description of services. In Section 3 we summarise the background on semiring-based soft constraints [10], as well as the background on SCA [5]. Section 4 shows some examples of how to use SCA to represent the behavior of services and the similarity between their operation and parameter names. In Section 5 we describe the architecture of a tool that implements the search introduced in Section 4. In Section 6 we focus on how we measure the similarity between two different behavioral signatures. In Section 7 we explain our QoS-based service recommendation method to assist users in service selection. The evaluation of the proposed approach is presented in Section 8, while in Section 8.1 we first describe our scalable peer-to-peer implementation of the discovery tool and its performance. In Section 9 we report on the related work. Finally, in Section 10 we draw final conclusions and explain our future work.

## 2 Behavioral Description of Services

Beyond the semantic description of the operations that a service can provide, and the syntax of how they are to be invoked, a specification of the proper order in which those operations can be invoked is a prerequisite for the correct implementation and use of a service. By behavioral specification of a service, we mean the specification of all admissible invocation orders of the operations of that service. The discovery of suitable services matching a query must consider the behavioral specification of candidate matches. What operations can be performed at a given point in time by a client of a service may depend on the history of the previous operations that have already been performed (usually, by the same client) on that service. Therefore, the specification of the behavior of a service is, in general, "stateful". However, these states are not always maintained within the service itself.

We propose the term *exostate* to denote the states of a running service or system that are maintained outside of the implementation of the service, and we use the term *endostate* to capture its internal configuration states. Trading endostates for exostates has important architectural advantages. For instance, because a *RESTful* service (where REST stand for *Representational State Transfer*) [40] has only a single endostate, it never needs to reset itself to recover from a communication failure that disrupts a client's session. RESTful services are commonly referred to as stateless, because they are designed to have a single configuration state, or endostate. However, most of such services are not truly stateless, in the sense that to use them properly, a client must still follow a permissible sequence of invocation of its operations, encoded in its exostates. The exostates of a RESTful service are represented by the values of a set of context parameters that are passed back and forth between the service and each of its clients.

Consider a hotel booking example, such as the RESTful service of [34] illustrated in Figure 1. In our terminology, as a RESTful service, the implementation of this service has a single endostate. However, this service cannot be used properly unless, for instance, *getHotelDetails* operation is invoked only after a *search* operation. Any proper use of this service requires remembering whether or not a search has indeed been performed yet, and perhaps the results of such a search. The REST architecture [40] requires such information to be kept outside of the service implementation itself, on the client/user side (perhaps as cookies), and passed back and forth between the client and the service. From the perspective of a client, however, the stateless service of [34] depicted in Figure 1 cannot

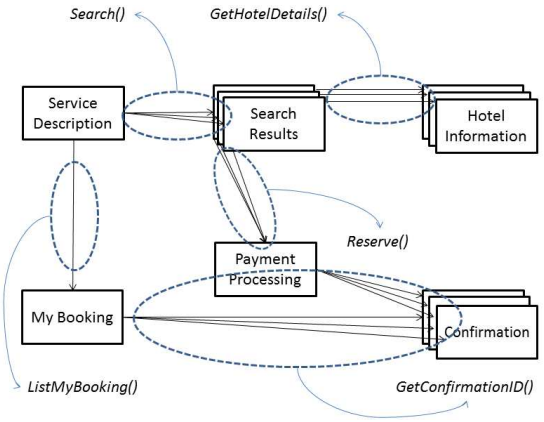


Figure 1: Operations of a restful example: the Hotel booking service.

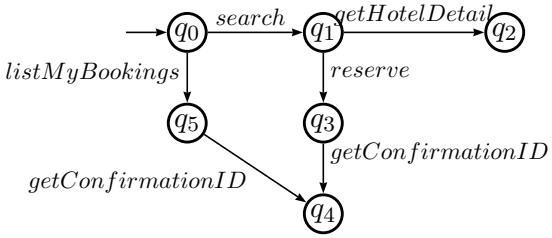


Figure 2: The CA modeling the behavior of the Hotel booking service.

be used without considering the specification of its stateful behavior, which is shown in Figure 2.

The previous example illustrates that many more systems and services than commonly acknowledged are truly stateful, in that the specification of their behavior involves more than a single state, regardless of whether such states are implemented as endostates or exostates. Searching for an appropriate service, as well as its manual or (semi-)automated adaptation, composition, or invocation must take its desired behavior into account. In this paper, we use the term "stateful" comprehensively to refer to any service the specification of whose behavior requires more than a single state. To the best of our knowledge, previous work on matching and retrieval of services does not consider the impact of exostates on suitability of the behavior of services in their search. This fact makes our behavior-based discovery tool not directly comparable with search and retrieval tools, such as the work in [39].

WSDL is the most widely used language to syntactically describe a WS interface. Although WSDL provides required information to establish a connection with a WS, this specification lacks the behavior description of the services [1]. As a consequence, we propose to improve the WSDL through a WS behavior-specification (WSBS). We call such extension as BWSDL, i.e., the WSDL document enriched by the WSBS.

This paper proposes this lightweight extension of WSDL to incorporate behavioral information. The BWSDL extension follows the rules for extending WSDL [14] to guarantee that any service consumer unaware of the extensions can still parse, validate and use the extended version of WSDL files, i.e., BWSDL documents. A new namespace "BWSDL" should be used to identify the tags part of the extension. Our approach retains the original structure of the WSDL documents and merely enhances them by adding a few new tags to the bottom of the XML-based files. Figure 3 shows an example of the WSDL extension by considering the behavior description represented in Figure 2.

We have implemented a GUI to ease the behavioral specification of services and to allow its visualisation. We have extended *Fizzim*[3], which is an open-source graphical *Finite State Machine* (*FSM*) design tool [46]. Fizzim is written in Java, while its back-end is written in Perl for portability and ease of modification. Our extension of the tool opens a WSDL document as input, and then draws a preliminary graph of its behavioral specification. This preliminary specification assumes each operation of the service as a self-loop transition on a single state,

[3]Fizzim: `http://www.fizzim.com`.

```
…
<BWSDL: Behaviour>
        <transition: Name= “Search”, CurrentState= “q0”, NextState= “q1”>
        <transition: Name= “getHotelDetails”, CurrentState= “q1”, NextState= “q2”>
        <transition: Name= “reserve”, CurrentState= “q1”, NextState= “q3”>
        <transition: Name= “getConfirmationID”, CurrentState= “q0”, NextState= “q5”>
        <transition: Name= “getConfirmationID”, CurrentState= “q3”, NextState= “q4”>
</BWSDL: Behaviour>
…
```

Figure 3: An example of WSDL extension by adding behavior description, i.e., BWSDL.

which means the execution of each operation is independent from other operations.

Figure 4 shows a screen-shot example of a preliminary behavioral specification, obtained by loading a *purchase.wsdl* document. The WSDL document describes that there are two operations provided by the service, namely “sendPurchaseOrder” and “invoiceCallbackPT”, so two self-loop transitions are drawn for them automatically in the preliminary behavioral specification of the service. Then, the user can exploit the tool to assemble SCA and immediately visualise their states and transitions, with the purpose to specify the proper behavior of the represented service. For example, Figure 5 is a revised version of the description specified in Figure 4. Finally, the designed graphical description of the service behavior can be exported as a BWSDL document.

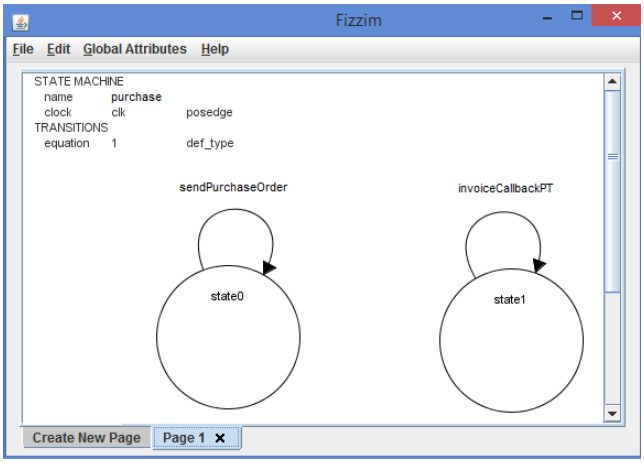


Figure 4: The preliminary behavioral specification of ”Purchase” service.

## 3 Soft Constraint Automata

**Semiring-based Soft Constraints.** A *c-semiring* [10] (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 write $a \leqslant_S b$ 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., $a + b = lub(a, b)$) [10].

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 $\hat{+}$

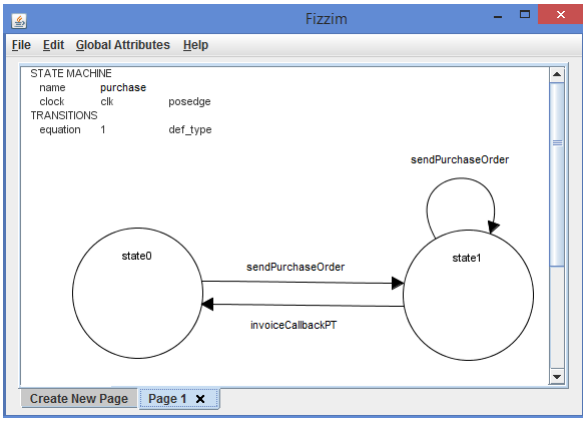


Figure 5: The revised version of the behavioral description of Figure 4.

respectively represent the arithmetic multiplication and addition).

A *soft constraint* [10] 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 Figure 7. 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 $\eta : V \rightarrow D$ of the variables, returns a value of the semiring, i.e., $c : (V \rightarrow D) \rightarrow A$. Let $\mathcal{C} = \{c \mid c : D^{|I \subseteq V|} \rightarrow A\}$ 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 $\eta'$ is $\eta$ modified with the assignment $v = d$. Note also that $c\eta$ is the application of a constraint function $c : (V \rightarrow D) \rightarrow A$ to a function $\eta : V \rightarrow D$; 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 $\otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ is defined as $(c_1 \otimes c_2)\eta = c_1\eta \times c_2\eta$ [10]; $supp(c_1 \otimes c_2) = supp(c_1) \cup supp(c_2)$. Likewise, the combination function $\oplus : \mathcal{C} \oplus \mathcal{C} \rightarrow \mathcal{C}$ is defined as $(c_1 \oplus c_2)\eta = c_1\eta + c_2\eta$ [10]; $supp(c_1 \oplus c_2) = supp(c_1) \cup supp(c_2)$. 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 Section 5 relies on the solution of *Soft Constraint Satisfaction Problems* (*SCSPs*) [10], 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.

**Soft Constraint Automata.** Constraint Automata were introduced in [7] as a formalism to describe the behavior and possible data flow in coordination models (e.g., Reo [7]); they can be considered as acceptors of *Timed Data Streams* (*TDS*) [4, 7]. We now recall the definition of *TDS* from [4], while extending it using the softness notions described in Section 3: we name this result as *Timed Weighted Data Streams* (*TWDS*), which correspond to the languages recognised by *SCA*.[4] For convenience, we consider only infinite behavior and infinite streams that correspond to infinite "runs" of our soft automata, omitting final states, including deadlocks.

**Definition 1 (Timed Weighted Data Streams)** *Let Data be a data set, and for any set $X$, let $X^\omega$ denote the set of infinite sequences over*

[4]TWDSs do not imply time constraints, and thus our (soft) CA are not "timed" [7].

$$\langle \lambda, l, a \rangle \in Data^{\omega} \times \mathbb{R}_{+}^{\omega} \times A^{\omega} \textit{ such that, } \forall k \geqslant 0 : l(k) < l(k+1) \textit{ and } \lim_{k \to +\infty} l(k) = +\infty$$

Thus, a *TWDS* triplet $\langle \lambda, l, a \rangle$ consists of a data stream $\lambda \in Data^{w}$, a time stream $l \in \mathbb{R}_{+}^{\omega}$, and a preference stream $a \in A^{\omega}$; $k$ is a natural number that is used to enumerate the elements of each stream. The time stream $l$ indicates, for each data item $\lambda(k)$, the moment $l(k)$ at which it is exchanged (i.e., being input or output), while the $a(k)$ is a preference score related to $\lambda(k)$.

In [5] we paved the way to the definition of *Soft Constraint Automata* (SCA), which represent the theoretical foundation behind our tool. 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 labelled with pairs consisting of a non-empty subset $N \subseteq \mathcal{N}$ and a soft (instead of crisp as in [7]) data-constraint $c$. Soft data-constraints can be viewed as an association of data assignments with a preference for that assignment. Formally,

**Definition 2 (Soft Data-Constraints)** *A soft data-constraint over a set of port names $\mathcal{N}$ and data values $Data$, is an expression produced by the following grammar, with* $\mathbf{c}$ *as its distinguished symbol:*

$$\mathbf{c} ::= \mathbf{f} \mid \mathbf{f} \oplus \mathbf{f}$$
$$\mathbf{f} ::= \bar{\mathbf{0}} \mid \bar{\mathbf{1}} \mid \mathbf{c} \otimes \mathbf{c} \mid (\mathbf{c}) \mid c$$

*where for $N \subseteq \mathcal{N}$, $c$ represents a function $c : (\{d_n \mid n \in N\} \to Data) \to A$ over a semiring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, the set of variables $\{d_n \mid n \in N\}$ constitute the support of the constraint, and $\{d_n \mid n \in N\} \to Data$ is a function that associates with every variable $d_n$ in this support, a data item $v \in Data$ that passes through the port $n \in N$.*

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 $SDC(N, Data)$, 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 $SDC(\mathcal{N}, Data)$. Note that in Definition 2 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 $\eta$ for the variables $\{d_n \mid n \in N\}$ satisfies $c$ with a preference of $a \in A$, if $c\eta = a$.

In Definition 3 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 [7]. Therefore, CA are subsumed by Definition 3. Note also that weighted automata, with weights taken from a proper semiring, have already been defined in the literature [19]; in SCA, weights are determined by a constraint function instead.

**Definition 3 (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) $c \in SDC(N, Data)$ (see Definition 2). $\mathcal{T}_S$ is called finite iff $\mathcal{Q}, \longrightarrow$ and the underlying data-domain $Data$ are finite.*

The intuitive meaning of an SCA $\mathcal{T}_S$ as an operational model for service queries is similar to the interpretation of labelled transition systems as formal models for reactive systems. The states represent the configurations of a service. The transitions represent the possible one-step behavior, where the meaning of $q \xrightarrow{N,c} p$ is that, in configuration $q$, the ports in

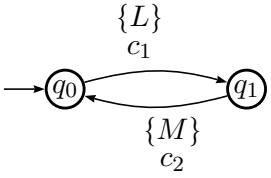


Figure 6: A Soft Constraint Automaton.

$c_1 : (\{d_L\} \to \mathbb{N}) \to \mathbb{R}^+$ s.t. $c_1(d_L) = d_L + 3$
$c_2 : (\{d_M\} \to \mathbb{N}) \to \mathbb{R}^+$ s.t. $c_2(d_M) = d_M + 5$

Figure 7: $c_1$ and $c_2$ in Fig 6.

$n \in N$ 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 Figure 6 we show an example of a (deterministic) SCA. In Figure 7 we define the *weighted* constraints $c_1$ and $c_2$ that describe the preference (e.g., a monetary cost) for the two transitions in Figure 6, e.g., $c_1(d_L = 2) = 5$.

In [5] 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 performance, e.g., due to the high number of requests. Definition 4 formalises the notion of soft synchronisation-constraint.

**Definition 4 (Soft Synchronisation-constraint)** *A soft synchronisation-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 port, and* $\mathcal{N}$ *is the set of I/O port names of the SCA.*

# 4 Representing the behavior of Services with SCA

In this section we show how the formal framework presented in Section 3 (i.e., SCA) can be used to consider a similarity score between a user's query and the service descriptions in a database, in order 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 (see Section 3) $q_0$ and $q_1$, in Figure 8. Soft constraints $c_1$ and $c_2$ in Figure 9, 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 modelled 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 Figure 9 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 $0.8$,

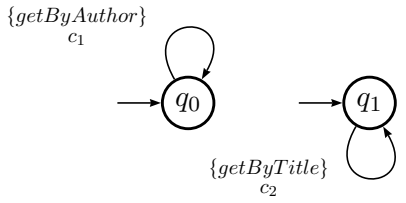


Figure 8: Two soft Constraint Automata representing two different queries.

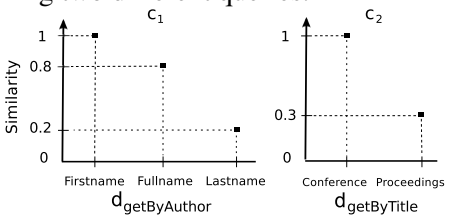


Figure 9: The definitions of $c_1$ and $c_2$ in Figure 8.

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 Figure 8) and $c_2$ in Figure 9. Similarity scores are automatically extracted as explained in Section 5.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Suppose now that our database contains the four services represented in Figure 10. All these services are stateless, i.e., their SCA have a single state each. For instance, service $a$ has only one invokable 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 Figure 9, queries $q_0$ and $q_1$ in Figure 8 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).

We now move from parameter names to operation names, and show that by using soft synchronisation constraints (see Definition 4), we can also compute a similarity score among them. For example, suppose that a user queries $q_0$ in Figure 8. The possible results are services $a$, $b$ and

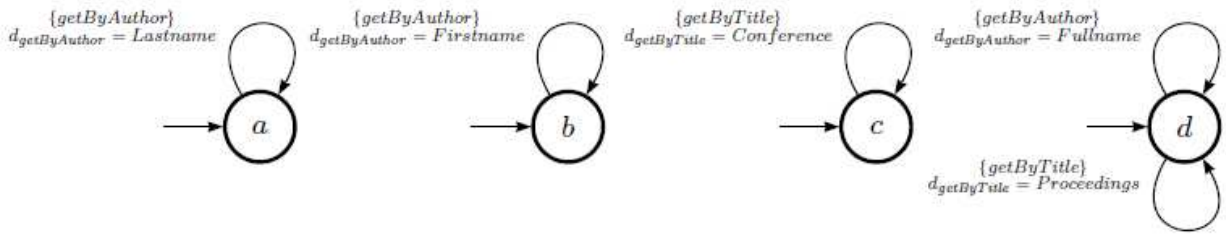


Figure 10: A database of services for the queries in Figure 8; $d$ performs both kinds of search (by author and by title).

$d$ in the database of Figure 10, 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 Figure 11, 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, our main goal is to compute a similarity score considering also the behavior of queries and services. For instance (the query in Figure 12), a user may need to find an on-line purchase service satisfying the following requirements: *i*) shipping activity comes before charging 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 Section 6 we will focus on this aspect.

Constraints on parameters (their data-types as well) and operation names can be straightforwardly

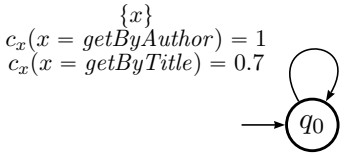


Figure 11: A similarity-based query for the *Author/Title* example.

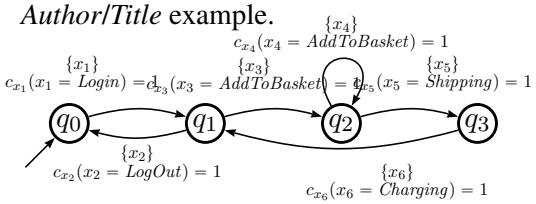


Figure 12: A similarity-based query for the on-line purchase service.

mixed together to represent a search problem where both are taken into account simultaneously for optimization. The tool in Section 5 exploits this kind of search: the similarity functions represented by constraints are computed through the composition of different syntactic similarity metrics.

# 5 Tool Description

Conceptually, our behaviorally-based WS discovery proceeds in four successive steps: *i*) generate a *Web*

*Service Behavior Specification* (*WSBS*) for each registered WS (a WSBS is basically a CA), *ii*) process preference-oriented queries (basically represented as SCA), *iii*) compute an operational similarity-score between a query and our services as an SCSP (see Section 3), and finally, *iv*) solve this problem (see Section 3). Note that we are also able to translate other kinds of behavioral service specification, as WS-BPEL[5], into (S)CA [12].

Step *i* is needed because no standard language or tool exists to specify the behavior of stateful WSs. Therefore, we have to define our own WSBS as a behavioral 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 step *iii*, we set up an SCSP (see Section 3), where soft-constraint functions are assembled by using the similarity scores derived in step *ii*; at the same time, we define those constraints that compare the two behavioral signatures (query/service), and measure their similarity. Finally, we find the best solutions for this SCSP, and we return them to the user. These steps are implemented by different software modules, whose global architecture is defined in Figure 13.

**WSDL Parser.** We rely on a repository of WSDL documents that are captured in a registry, i.e., the *WSDL Registry* (see Figure 13). 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.[6]

**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 behavioual aspects. In fact, a WSDL document only reveals the operation names and the names and data types of their arguments; it does not 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 [29], some of the authors of this paper have already formalised the behavior of a WS (i.e., the WSBS) in terms of CA [7]. Therefore, we adopt the *Extensible Coordination Tools* (*ECT*) [3], which consist of a set of plug-ins for the *Eclipse* platform[7], as the core of the *WSBS Generator*, in order to generate a CA to specify the externally observable behavior of a service. Normally, CA are used as operational semantics for Reo circuits [7]. 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 behavior of WSs in text files, in a simplified form. The file in Figure 15 describes the service represented in Figure 14. In our architecture, all WSBSs are stored in a *WSBS Registry* (see Figure 13).

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 Figure 13) 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

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

[6]http://axis.apache.org/axis2/java/core/.

[7]http://reo.project.cwi.nl/reo/wiki/Tools.

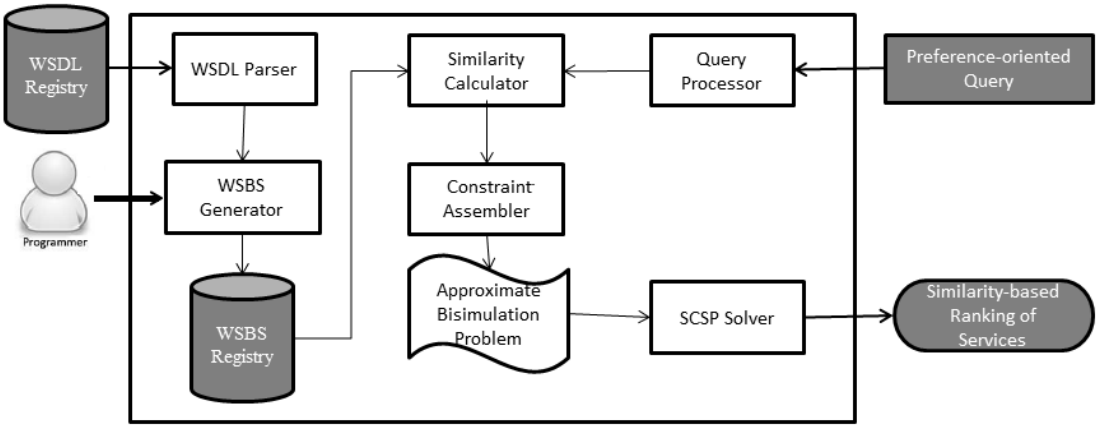


Figure 13: The architecture of the tool.

is represented in Figure 16. 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 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 Figure 16. Each query is represented as an SCA [5] (see Section 3), 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 Figure 16 and Figure 15 for a comparison).

**Similarity Calculator**. As Figure 13 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, the *Levenshtein Distance*, the *Matching Coefficient*, and the *QGrams Distance* [13]. Each of these metrics operates with two input strings, and returns 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

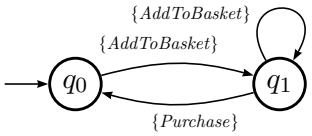


Figure 14: An example of WSBS.

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

Figure 15: Text file representing the WSBS in Figure 14.

the *Constraint Assembler* in Figure 13, in order to define the similarity functions that are translated into soft constraints, as explained in Section 4. The representation of the search problem 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 operational similarity-evaluation (see Section 6), as an SCSP (see Section 3). To do so, it represents all preference and similarity requirements as soft constraints. In order to assemble these constraints, we use *JaCoP*, which is a Java library that provides a finite-domain constraint programming paradigm. We have made ad-hoc extensions to the crisp constraints supported 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 optimization problem, by implementing the *fuzzy* semiring, i.e., $\langle [0..1], \max, \min, 0, 1 \rangle$ (see Section 3).

For instance, *SumWeight* is a JaCoP constraint that computes a weighted sum as the following pseudocode: $w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3 = sum$, where $sum$ represents the global syntactic similarity between two operation in terms of the similarity between their operation names ($x_1$), their argument names ($x_2$), and their argument 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 Section 5 we use equal weights. In Section 6, we discuss how to compute how much two behavioral signatures (query/service) are similar, and how we construct the general constraint-based model.

**SCSP solver**. Finally, after the specification of the model in terms of variables and constraints, a search for a solution of the assembled SCSP can be started. This represents the final step (see Figure 13). 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 16 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 have developed a WS crawler to find WSDL documents, and check their validations. We retrieved more than 2000 different WSDL documents, but about 1000 of their corresponding WSs are valid. The validated WSDL documents form our *WSDL Registry* in Figure 13.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo.

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

Figure 16: A single-state query asking for the weather conditions over a `City`, or a `Zipcode`. Different user preference scores are represented within square brackets.

Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Table 1 shows top-ten ranked experiment results, where the other WSs obtained a similarity score less than $0.3$. From left to right the columns respectively show the position in the final ranking, the obtained fuzzy score, the WS name, and, lastly, the matched service operation.

# 6 On Comparing Behavior Signatures

In this section we zoom inside the *Constraint Assembler* component that we introduced in Section 5. We describe how we can approximate the behavior of a posed query with that of a service, since a perfect match can be uncommon.

The basic idea is to compute an operational similarity-score between two automata, respectively representing a query and a WS in a database. The notion of operational similarity 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 behaviors. 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 compute such operational similarity with constraints, we exploit constraint-based graph matching techniques [42]; thus, we are able to "compress" or "dilate" one automaton structure into another. We take advantage of the notion of *sub-graph epimorphism*, corresponding to the application of node delete and merge operations to pass from an SCA to another when checking operational similarity. The existence of an epimorphism from a graph to another is an NP-Complete problem [23].

In the following, we use the query example in Figure 17, and the service example in Figure 18 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 (set $Q$) we define a variable that can be assigned to one or several states of a service (set $S$). 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 Figure 17 and Figure 18 using grey 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

Table 1: The ranking of the top-ten matched WSs, based on the query represented in Figure 16.

| Rank | Score | Name of WS | Interface of the operation |
|---|---|---|---|
| 1 | 0.69 | globalweather | $GetWeather(CityName : string)$ |
| 2 | 0.54 | usweather | $GetWeatherReport(CityName : string)$ |
| 3 | 0.5 | Weather | $Get\_Weather(ZIP : string)$ |
| 4 | 0.48 | WeatherWS | $getWeather(theCityCode : string, theUserID : string)$ |
| 5 | 0.44 | usweather | $GetWeatherReport(ZipCode : string)$ |
| 6 | 0.42 | WeatherForecast | $GetWeatherByZipCode(ZipCode : string)$ |
| 7 | 0.4 | WeatherForecast | $GetWeatherByPlaceName(PlaceName : string)$ |
| 8 | 0.36 | weatherservice | $GetLiveCompactWeatherByStationID(stationid : string,$ $un : UnitType, ACode : string)$ |
| 9 | 0.34 | weatherinfo | $WeatherInfoByPstcode(PostCode : string)$ |
| 10 | 0.30 | weather-area | $GetWeatherArea(id : string)$ |

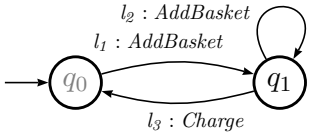


Figure 17: A query example.

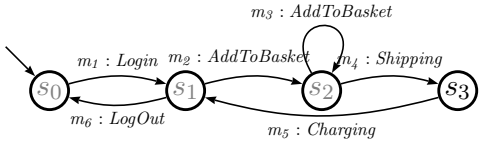


Figure 18: A possible service in a database related to the query in Figure 17.

one or more service-states; on the other hand, different query states can be associated 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., $s_0$, $s_1$ and $s_3$ in Figure 18) 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 behaviors 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 Figure 17, we have three variables $l_1, l_2, l_3$. In Figure 17 and Figure 18 we label each transition with its identifier ($l_1, \ldots, l_3, m_1, \ldots, m_6$), 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.

**Automata Epimorphism.** Algorithm 1 shows our approach to find sub-graph epimorphism of the automata to match behavior specification of the query and services. The idea is that we can merge two or more neighbouring states, i.e. states connected by transitions, to one single state. Every such merged state needs to adjust its transitions. In this way every in-coming transition to one of the combined states, comes to the new state (the merged one). Similarly,

outgoing transitions of both states become outgoing transitions of the combined state.

For example, we can find two automata epimorphisms for the service represented in Figure 18 with the corresponding query-automaton depicted in Figure 17. The state cardinality of the service-automaton is four, while it is two for the query-automaton. Therefore there are two ways to merge: (1) merge three neighbouring states into one state besides the other state, (2) merge two neighbouring states into one state, and merge the remaining two neighbouring states into another single state. Figure 19 shows two possible automata epimorphisms for this example. These two results can be obtained through Algorithm 1 at the following steps (as a reminder, $|Q|$ is 2 and $|S|$ is 4): Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

The first sub-graph epimorphism is the result of step $i = 2$, so three (i.e., $i + 1$) neighbouring states are merged into one state (i.e., state $s_0$ is merged with state $s_1$ and state $s_3$, and then becomes $s_0$'), and the other state, i.e., state $s_2$ forms the other node of the new automaton. By ignoring the dissolved transitions (i.e., internal transitions among the merged nodes), the transition matrix of the new automaton (merged one) is:

$$T = \begin{bmatrix} Null & AddToBasket \\ Shipping & AddToBasket \end{bmatrix}$$

The second sub-graph epimorphism is the result of step $i = 1$, so two (i.e, $i + 1$) neighbouring states are merged into one state (i.e., state $s_0$ is merged with state $s_1$, and then becomes $s_0$'), and the other two states form the other node of the new automaton: state $s_2$ is merged with state $s_3$, and then becomes $s_1$'. By ignoring the dissolved transitions (i.e., internal transitions among the merged nodes) the transition matrix of the new automaton (merged one) is:

$$T = \begin{bmatrix} Null & AddToBasket \\ Shipping & Charging \end{bmatrix}$$

In order to compare the behavior of the query and services, we replace the transition matrix $T$ of each automaton by the adjacency matrix $A$, where $A[i, j] = 0$ if $T[i, j]$ is Null, and 1 otherwise. While the transition matrices of the two possible automata epimorphisms of the above example are different (at $T[1, 1]$), their adjacent matrixes both are the same with the adjacent matrix of the query and it is:

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}$$

It means that new automata epimorphisms are behaviorally matched with the query, but due to the incurring of the penalty factor ($Q/S$), their state similarity-scores would be $0.5$. The transition similarity-scores for these two alternatives, which are derived from a comparison between matched labels, would be respectively $0.36$ and $0.43$. Therefore, the second epimorphism that results in a higher similarity score is considered during the discovery process for the service.

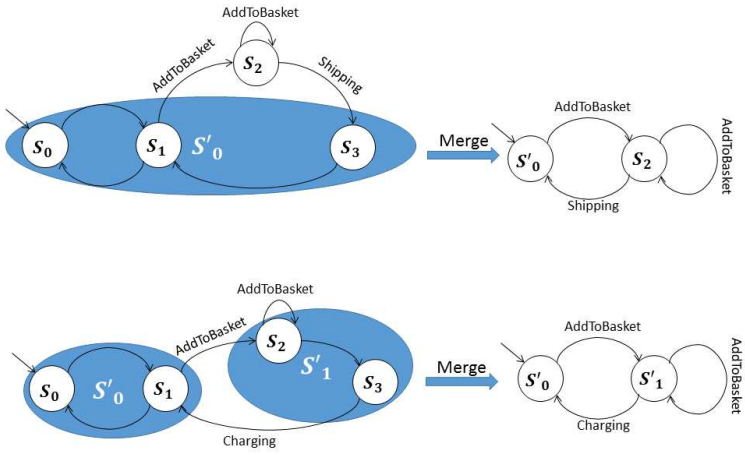


Figure 19: Two possible sub-graph epimorphisms considering Figure 18.

**Operational-similarity of the Match.** In this paragraph we show how to compute a global similarity score $\Gamma$ 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 behavior (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. The score $\theta(\mathcal{M})$ is computed by aggregating the individual *ssim* syntactic similarity-scores (computed by the *Similarity Calculator* proposed in Section 5) obtained for each label match, and then averaging on the number of matched labels. In 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.** As we discussed in Section 2, proper descriptions of the behavior of many of (implementations of) stateless services are stateful. Therefore, we enrich our registered WSs with behavioral descriptions and use the query represented in Figure 20 against this database. According to this stateful query, the ideal service matching the query first performs the Add-to-Basket operation one or more times, and then completes the required shipping processes, and finally charges the costumer for its purchase. 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 behavior of $S_6$ is identical to the behavior of our query, and the behaviors of $S_3$, $S_1$, and $S_2$ are approximately close to the behavior of our query.

```
q0 AddBasket() q1; q1 AddBasket() q1; q1 Shipping() q2; q2 Charge() q0, [1.0]
```

Figure 20: A stateful query asking for a purchase-online scenario including buying, shipping and charging.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

# 7 QoS-aware Service Discovery

While the proposed search engine discovers services according to their functional properties, non-functional properties, e.g., QoS parameters, play an important role in a user's selection. When many web services offer similar capabilities, it is necessary to also consider a set of non-functional properties of services as selection criteria. Functional properties describe what a service can do, while non-functional properties depict how well the service can satisfy its functional properties.

Besides the set of functional requirements (i.e. syntactical and behavioral matching of services), which we model as soft constraints, we can also encode any set of QoS requirements as soft constraints in order to assist users in service selection. In principle, we can compute QoS ranking as an extra criterion for search, but doing so may impact the results by giving higher ranking to completely irrelevant services that have high QoS values. To avoid this problem, we use the lexicographic ordering.

We compose all the constraints representing functional and non-functional requirements in the constraint assembler, but in a given lexicographic order: different criteria have different precedence. Assume the user considers three QoS criteria for the service selection including Availability, Reliability, and Response-Time. He/she also states a lexicographic order among these criteria: the first preferred component is Reliability, the second one is Response-Time, and the last one is Availability. In order to combine functional requirements and QoS constraints into a single semiring for match-making of services, we define the lexicographic product of semirings.

Our definition of lexicographic ordering derives from a lattice-based distance function we define in Definition 5. We use this function to exploit the structure of the complete lattice defined by $\langle A, + \rangle$

1. Let $\mid Q \mid$ be the cardinality of the query-automaton states, and $\mid S \mid$ be the cardinality of a service-automaton
2. If $(\mid Q \mid < \mid S \mid)$ then
   *%Find all possible sequences of merges for neighbouring states in the service-automaton to reduce its state's cardinality to* $\mid Q \mid$.
   2.1. For($i = \mid S \mid - \mid Q \mid;\ i \geqslant (|S| - |Q|)/2;\ i = i - 1$)
      2.1.1. Merge $i + 1$ neighbouring states of the service to a single state.
      2.1.2. Merge $(\mid S \mid - \mid Q \mid) - i + 1$ neighbouring states of the service to a single state.
      2.1.3. Adjust the transitions for the merged states.
      2.1.3. If (the new automaton is equal to the query-automaton) then consider it as a sub-graph epimorphism of the service-automaton.
3. If $(|Q| > |S|)$ then
   Find all possible sub-graph epimorphism of the query-automaton to the service-automaton similar to 2.1

**Algorithm 1:** The merging algorithm.

(see in the following why and how). A complete lattice is a partially ordered set in which all subsets have both a supremum ($\mathbf{1}$) and an infimum ($\mathbf{0}$).

**Definition 5 (Lattice-based distance)** *The lattice-based distance is a function* $dist : A \times A \rightarrow \mathbb{N}$ *defined on a semiring* $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$*, that returns the distance between two elements (absolute value) on the lattice defined by* $\langle A, + \rangle$*. In the following we provide two examples, depending if* $A$ *is*

**Finite and partially ordered.** *Given* $a_1, a_2 \in A$*,* $dist(a_1, a_2)$ *is the length of the shortest path between* $a_1$ *and* $a_2$ *on the complete lattice defined by* $\langle A, + \rangle$*.*

**Infinite and totally ordered.** *Given* $a_1, a_2 \in A$ *and* $a_1 <_S a_2$*,* $dist(a_1, a_2)$ *is the distance obtained through a weak inverse of* $\times$*, i.e.,* $a_1 \div a_2$ *[22].*[8]

Since, semirings can be defined over partial order sets, they can be represented as lattice graphs. The distance between two vertices in a lattice graph is the number of edges in a shortest path connecting them. To compute the $dist$ function, we can use a distance matrix that is a square matrix containing the distances, taken pairwise, between the elements of a poset. For example, $dist(\{T\}, \{A, R, T\})$ in Figure 21 is 2.

We now define a lexicographic ordering which takes advantage of Definition 5. The goal is to use the distance of two elements from their least upper bound (obtained through $+$) as additional information to "stretch" the partial order into a "more" total order, to be considered in the lexicographic order.

**Definition 6 (Lexicographic product of semirings)** *The lexicographic product of semirings* $S_1 = \langle A_1, +_1, \times_1, \mathbf{0}_1, \mathbf{1}_1 \rangle$ *and* $S_2 = \langle A_2, +_2, \times_2, \mathbf{0}_2, \mathbf{1}_2 \rangle$*, denoted as* $S_1 \rhd S_2$*, is the semiring* $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ *where:*

- *The carrier of* $S$ *is the Cartesian product of the carries of* $S_1$ *and* $S_2$*, i.e.,* $A = A_1 \times A_2$*.*
- *The selection operator* $+$ *of* $S$ *if given by:*

[8] Please refer to [22] for a formal definition of $\div$, which is outside the scope of this paper.

Table 2: The ranking of the top-ten matched WSs, based on the query represented in Figure 20.

| **Name** | **WSBS** | $\theta$ | $\sigma$ | $\Gamma$ | **Rk** |
|---|---|---|---|---|---|
| S6 | q0 AddToBasket(code:string) q1; q1 AddToBasket(code:string) q1; q1 Shipment(Address:String) q2 ; q2 Charge(AccountOnfo:string) q0 | .88 | 1.0 | .94 | 1 |
| S3 | q0 Login(IdInfo:String) q1; q1 AddToBasket(code:string) q2; q2 AddToBasket(code:string) q2; q2 Shipping(Address:String) q3 ; q3 Charging(AccountInfo:string) q1; q1 Logout() q0 | .89 | .75 | .82 | 2 |
| S1 | q0 AddItem(code:string) q1; q1 AddItem(code:string) q1; q1 Shipping(Address:String) q2 ; q2 Charge(AccountInfo:string) q0 | .56 | 1.0 | .78 | 3 |
| S2 | q0 AddToBasket(code:string) q1; q1 AddToBasket(code:string) q1; q1 Shipment(Address:string) q0 | .77 | .67 | .72 | 4 |
| S7 | q0 AddProduct(Prodcode:string) q1; q1 AddProduct(Prodcode:string) q1; q1 charge(AccountInfo:string) q0 | .49 | .67 | .58 | 5 |
| S4 | q0 AddItemToBasket(code:string) q1; q1 AddItemToBasket(code:string) q1; q1 Charging(AccountInfo:string) | .44 | .67 | .56 | 6 |
| S5 | q0 Login(IdInfo:String) q1; q1 AddProduct(Prodcode:string) q2; q2 AddProduct(Prodcode:string) q2; q2 Shipment(Address:String) q3 ; q3 Payment(AccountInfo:string) q1; q1 Logout() | .32 | .75 | .54 | 7 |
| TrackShipment | q0 ShippingInfo(Info:string) q0; | .67 | .33 | .50 | 8 |
| ItemBasketService | q0 AddItemBasket(Itemcode:string) q1 ; q1 AddItemBasket(Itemcode:String) q1 ; q1 checkout(Itemcode:String) q0 | .31 | .67 | .49 | 9 |
| purchase | q0 sendPurchaseOrder(order:string) q1; q1 sendPurchaseOrder(order:string) q1; q1 shippingPT(Info:String) q0 | .18 | .67 | .43 | 10 |

$$\langle a_1, a_2\rangle + \langle b_1, b_2\rangle = \begin{cases} \langle a_1, a_2\rangle & \textit{if } dist(a_1, lub_1) < dist(b_1, lub_1) \\ \langle a_1, a_2\rangle & \textit{if } a_1 = b_1 \textit{ and } dist(a_2, lub_2) < dist(b_2, lub_2) \\ undefined & otherwise \end{cases}$$

*where $lub_1$ is the least upper bound of $a_1$ and $b_1$ ($a_1 + b_1$), and $lub_2$ is the least upper bound of $a_2$ and $b_2$ ($a_2 + b_2$); $dist()$ is defined in Definition 5. We can even further refine the definition of our selection operator in Definition 6 by choosing the element that is more distant from their common greatest lower bound ($glb$) if two elements are equidistant from their $lub$. Note that two elements in $A$ are always comparable with respect to $dist$, while they may not be comparable in $\langle A, +\rangle$.*

- *The composition operator $\times$ of $S$ is given by:*

$$\langle a_1, a_2\rangle \times \langle b_1, b_2\rangle = \langle a_1 \times_1 b_1, a_2 \times_2 b_2\rangle$$

- $\mathbf{0} = \langle \mathbf{0}_1, \mathbf{0}_2\rangle$ *and* $\mathbf{1} = \langle \mathbf{1}_1, \mathbf{1}_2\rangle$.

The notion of distance that we have defined in the selection operator, is helpful in the case that two elements are not directly comparable. Assume several quality criteria of services are defined to rank WSs. This criterion would be partially ordered because all of its elements are not comparable. Figure 21 shows the lattice graph of this criterion including Availability (A), Reliability (R) and Throughput (T). For elements directly related by the partial order, for example $\{A\} < \{A, R\}$, the choice is clear, so $\{A, R\}$ would be preferred in this comparison. However, we cannot compare some of these elements to each other, for instance $\{A\}$ and $\{R, T\}$. But using the $dist$ function, $\{R, T\}$ is preferred because the $dist(\{R, T\}, \{A, R, T\})$ is 1, which is smaller than $dist(\{A\}, \{A, R, T\})$ (that is 2). In fact, we expect that $\{R, T\}$ would be better than $\{A\}$ because it is closer to the upper bound and satisfies more features.

We consider the lexicographic product operator on semirings to be right associative, i.e., $S_1 \rhd S_2 \rhd S_3 = S_1 \rhd (S_2 \rhd S_3)$. Accordingly, if $A_1$, $A_2$, and $A_3$ are the carriers of semirings $S_1$, $S_2$, and $S_3$, respectively, for simplicity, we skip ordering parentheses and denote the carrier of $S = S_1 \rhd S_2 \rhd S_3$ as $A = A_1 \times A_2 \times A_3$, instead of $A = A_1 \times (A_2 \times A_3)$, and denote the elements of $A$ as $\langle a_1, a_2, a_3\rangle$ instead of $\langle a_1, \langle a_2, a_3\rangle\rangle$.

Let $S_{Func}$, $S_{rel}$, $S_{ret}$, and $S_{avl}$ denote the semirings for the functional requirements and the QoS constraints for Reliability, Response-Time, and Availability, respectively. Then, $S = S_{Func} \rhd S_{rel} \rhd S_{ret} \rhd S_{avl}$ is the semiring that we use to find the best match for a query. The fact that $S_{Func}$ appears as the leftmost operand in the lexicographic product that defines $S$ ensures that we consider QoS properties of services in our ranking of candidates only if their functional similarity scores are the same.

# 8 Evaluation

The evaluation of the tool is divided into two parts. First, in subsection 8.1 we evaluate the performance and scalability of the search engine in a distributed implementation of the tool. Then, in subsection 8.2 we validate the adequacy of the tool by measuring its information retrieval metrics.

## 8.1 Performance and Scalability

Distributed computing presents an alternative to traditional centralised systems that can achieve

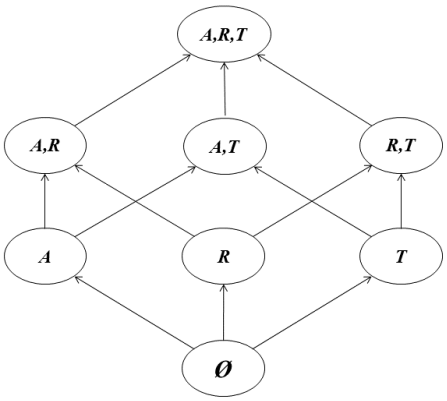


Figure 21: Lattice of subsets of $\{A, R, T\}$, partially ordered by "is subset of".

high-performance in executing heavy-workload tasks [27]. In order to improve the performance of our retrieval tool, we have designed a peer-to-peer (P2P) model for its implementation on a network of equivalent computer nodes. P2P systems offer efficiency, scalability, decentralised control, self-organisation, and symmetric communication [41]. In the P2P model of the tool, every node contains a list of services, as well as their searchable attributes for discovery. A query is passed to all of the nodes, and each node computes a list of similarity scores for its own content that matches the query as search results of our discovery tool. Each node sends its set of computed similarity scores to an aggregator, which merges the sets from multiple nodes to create a single set of ranked search results. Aggregators, too, send their results to other aggregators higher up in a hierarchy, until they reach the top-level aggregator that presents its ranked results to the user. The results of an aggregator node are directly comparable, because it receives the absolute similarity scores from different nodes. As an alternative, each node can send only the ranking of services, and use an aggregation function that weighs the ranking of each service by the number of services in the corresponding sender-node. In our implementation we adopted the former approach, which is based on the absolute similarity score.

The performance statistics that we describe below are calculated for $1000$ registered services. Table 3 shows the execution time and the speed-up to obtain the similarity scores for the registered services, which are distributed among $1$ to $10$ nodes. Table 3 shows that the minimum execution time is obtained with $10$ nodes, which is about $3.5$ times smaller than the execution time with a single node, i.e., without distribution. The processing time needed for parsing the WSDL documents and generating the WSBS specifications is not considered in this performance statistics because this parsing is executed only once (in the registration step). The graph on the left in Figure 22 shows how the execution time varies with the number of nodes (processors). It shows that the execution time is inversely related with the number of distributed nodes. The graph on the right in Figure 22 shows the rate of speed-up, which is equal to

$$\frac{\textit{the execution time using 1 processor}}{\textit{the execution time using n processors}}$$

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien

est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

## 8.2 Validation

We have evaluated the quality of the computed results according to the two most frequent and basic metrics used to measure the effectiveness of information retrieval: precision and recall [28]. Precision (also called positive predictive value) is defined as the number of relevant returned results divided by the number of returned results, while recall (also known as sensitivity) is measured as the number of relevant returned results divided by the total number of relevant entries in the database.

Let $Rel$ be the set of relevant WSs, $Ret$ be the set of returned WSs, $RetRel$ be the set of returned relevant WSs (true positives), and $RetRel_k$ be the set of relevant results in the top $k$ returned WSs. More precisely, the parameters that we have adopted to evaluate the performance of our approach are defined as follows:

$$Precision = |RetRel|/|Ret|$$

$$Precision_k = |RetRel_k|/k$$

$$Recall = |RetRel|/|Rel|$$

We have examined the results based on 20 test queries. For each of these queries, we have manually labelled our WSs in the database as relevant or irrelevant. Since the queries had less than 10 results, we considered the precision at 2, 5 and 10 (parameter $k$). The average $Precision_2$, $Precision_5$ and $Precision_{10}$ for the test queries are respectively $95\%$, $73.3\%$ and $65.4\%$. This preliminary evaluation

Table 3: Evaluation of the distributed model by matching the achieved speed-up with respect to the number of used nodes.

| Number of nodes | Execution time | Speed-up |
|---|---|---|
| 1 | 417 Millisecond | 1 |
| 2 | 213 Millisecond | 1,957746479 |
| 5 | 135 Millisecond | 3,088888889 |
| 10 | 117 Millisecond | 3,564102564 |

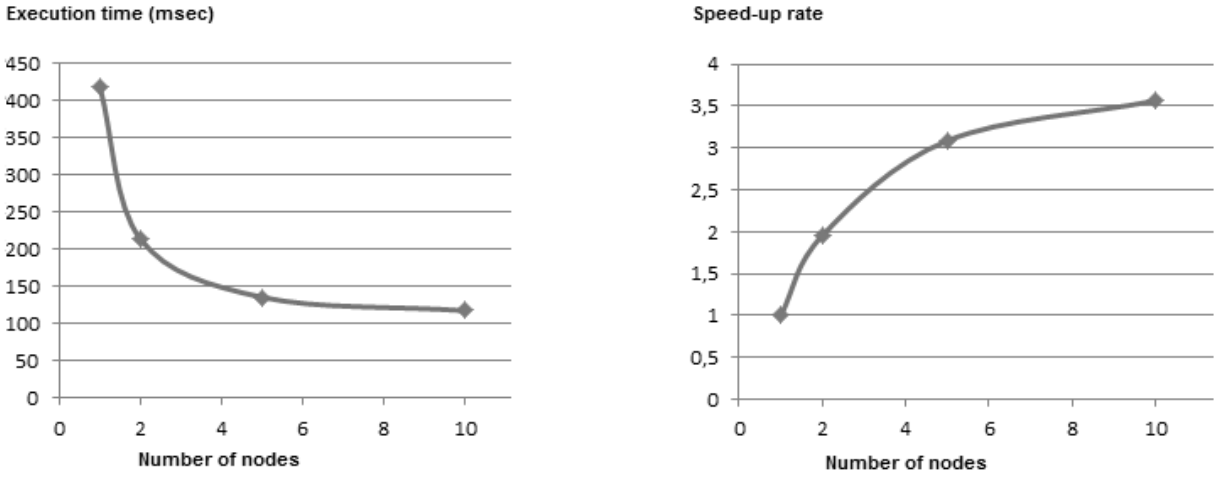


Figure 22: The graphs of execution time and speed-up rate for the example mentioned in Table 3.

is promising especially for stateful queries, although for stateless queries our approach shows no advantage compared with other approaches [39].

The Recall/Precision (R-P) curve is considered as the most informative graph showing the effectiveness of a search engine [18]. An ideal search engine has a horizontal curve with a high precision value; a bad search engine has a horizontal curve with a low precision value. The traditional approach to build a R-P curve is the 11-point interpolated average precision, where precision is measured at the 11 recall levels of $0.0, 0.1, 0.2, \ldots, 1.0$. For each recall level, we then calculate the precision of the results that meet that recall level of the test collection. For example, if the number of relevant WSs ($|Rel|$) is 10, the number of returned WSs ( $|Ret|$ ) at recall level $0.1$ would be the minimum number of the top-ranked search results to see $0.1$ portion of $|Rel|$, i.e. the first relevant WS.

The blue curve in Figure 24 shows the R-P graph of our tool (Beh-Search) for the 20 test queries, where recall level of the graph varies from 0 to 100 percent. For instance, Table 4 shows the results of search for one of the 20 queries to draw the R_P curve.

We have five relevant services in our database for this query. As we see in Table 4, the first relevant

service (i.e., *serviceSMS*) appeared as the first top-ranked discovered service, so the precision at the recall level $0.2$ (i.e. $1/5$) would be $100\%$. The precision for the second relevant service (i.e. recall level $0.4$) is also $100\%$, but for the third one it is $60\%$ because the third relevant service is the 5th top-ranked service in the list. The precision at the recall levels $0.8$ and $1.0$ is respectively, $66.67$ and $71.43$.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Figure 23 shows one of the other queries that we used for drawing the R-P curve. This query aimed at finding the services that can search and apply for jobs. According to the query behavior represented in Figure 23, the intended service should first register the client, then search for his/her desired jobs, and, finally, apply for one of the retrieved jobs.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Table 5 shows the top ranked results of the query represented in Figure 23. As we see in the table, the first six top-ranked discovered services are relevant, so the precision at recall levels $0.1, 0.2, \ldots, 0.6$ is $100\%$, which is very good. The precision at the next recall levels, i.e., $0.7$, $0.8$ and $0.9$ are still high (respectively, $87\%$, $89\%$, and $90\%$). The precision at

Table 4: The top-ranked matched WSs, based on the query: $q0$ $SendSMS$ $q0$

| Rank | Service Name | WSBS | Relevant | Score | Precision |
|---|---|---|---|---|---|
| 1 | ServiceSMS | q0 SendSMS q0 | R | 1.0 | 100 |
| 2 | BulkSMS | q0 SendSMS q0 | R | 1.0 | 100 |
| 3 | InfoService | q0 SendSMSInfo q0 | | 0.63 | |
| 4 | Server_utf8 | q0 SendSMS_USC2 q0 | | 0.59 | |
| 5 | SmsService | q0 AddSMS q0 | R | 0.52 | 60 |
| 6 | SmsWebService | q0 SMS q0 | R | 0.50 | 66 |
| 7 | BSWS | q0 SubmitSms q0 | R | 0.42 | 71.43 |
| 8 | ClickSmsV4 | q0 GetSMSInfo q0 | | 0.23 | |
| 9 | Price | q0 GetSMSPrice q0 | | 0.21 | |

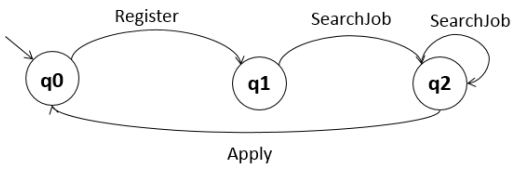


Figure 23: The query for discovering WSs that can search and apply for jobs.

the last recall level, i.e., $1.0$, which is the last relevant service for this query, drops to $59\%$. This service (behaviorally represented by ***q0 LookingForJob q0***) obtained $0.2$ as the global similarity score, and is at the 17th row of the table.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

The top-ten ranked services in Table 2, which are based on the query represented in Figure 20, is another instance of the queries we used to draw our R_P curve. As we see in the table, the first seven top ranked discovered services are relevant, so the precision at recall levels $0.1, 0.2, \ldots, 0.7$ is $100\%$. The precision at the next recall levels is also still high, e.g., it is $89\%$ at the recall level 0.8. We can argue that the precision for multi-state queries are generally better than those for single state queries, which is the biggest advantage of our tool. The reason is that the transition-similarity score is only a part of the global similarity score, which is used for

Table 5: The top-ranked matched WSs, based on the query represented in Figure 23.

| **WSBS** | **Relevant** | $\theta$ | $\sigma$ | $\Gamma$ | **Rk** |
|---|---|---|---|---|---|
| q0 Register q1; q1 Search q2; q2 Search q2; q2 Apply q0 | R | .84 | 1.0 | .92 | 1 |
| q0 Register q1; q1 LookingForJob q2; q2 LookingForJob q2; q2 Apply q0 | R | .62 | 1.0 | .81 | 2 |
| q0 Search q1; q1 Search q1; q1 Apply q0 | R | .79 | .67 | .73 | 3 |
| q0 login q1; q1 Register q2; q2 Search q3; q3 Search q3; q3 Apply q4; q4 logout q0 | R | .84 | .6 | .72 | 4 |
| q0 login q1; q1 SearchJob q2; q2 SearchJob q2; q2 Submit q3; q3 Resubmit q2; q3 logout q0 | R | .53 | .75 | .64 | 5 |
| q0 login q1; q1 Register q2; q2 LookingForJob q3; q3 Looking-ForJob q3; q3 Apply q4; q4 logout q0 | R | .62 | .6 | .61 | 6 |
| q0 Register q1; q1 Vote q2; q2 Vote q2; q2 Submit q0 | | .19 | 1.0 | .59 | 7 |
| q0 LookingForJob q1; q1 LookingForJob q1; q1 Apply q0 | R | .49 | .67 | .58 | 8 |
| q0 login q1; q1 LookingForJob q2; q2 LookingForJob q2; q2 Apply q3; q3 Change q2; q3 logout q0 | R | .33 | .75 | .54 | 9 |
| q0 SearchForJob q0 | R | .74 | .33 | .53 | 10 |
| q0 login q1; q1 Vote q2; q2 Vote q2; q2 Submit q0 | | .02 | 1.0 | .51 | 11 |

ranking of services. Therefore, if a service can be matched behaviorally with a query but uses different labels, it is still expected to obtain a high global similarity score because it earns a good state similarity score. For instance, assume the third ranked service in Table 2 that names the operation of adding an item to the shopping cart "AddItem", while its name is "AddToBasket" in the query. When compared to the other services, e.g. the forth ranked service, the transition-similarity score dropped but the state-similarity score raises the global-similarity score. This means that even in these cases the tool can discover relevant services with a significant global-similarity score.

The R-P graph for our tool (see Figure 24) shows that for the first set of relevant results (i.e. at recall level $10\%$), precision is appropriate. Also precision for the last relevant results (i.e. at recall level $100\%$) is comparable with that of other existing approaches presented in [39]. Figure 24 also shows the Recall/Precision comparison for the studied approaches in [39]. Note that our data set used to derive our R-P curve is different from those used for the approaches studied in [39]. The lack of widely accepted benchmarks using stateful services and the dearth of tools that work with such services makes it not possible to have a meaningful direct comparison of our R-P curve with those of the other tools.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

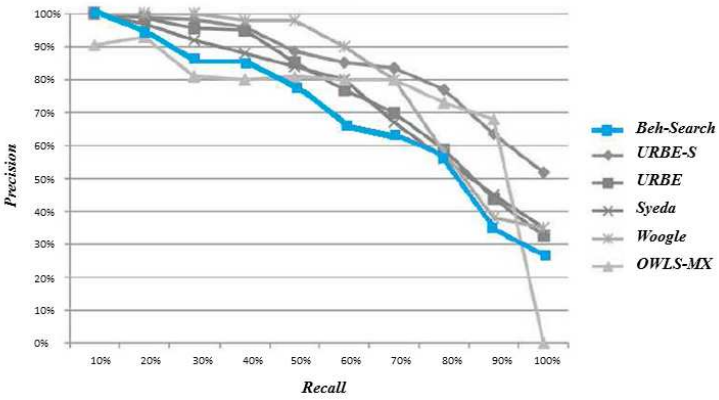


Figure 24: The R-P curve of some of related works.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

# 9 Related Work

Compared to the work reported in the literature, the solution in this paper seems more general, compact, and comprehensive, because it can encompasses any semiring-like metrics, and the whole framework is expressively modelled 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 [5]), or to use join and hide operators for their composition and abstraction [5]. A first step towards this work has been developed in [6]. Modelling and verification of long-running transaction involving composed WSs proposed in [32] and [37] are compatible with our framework. Sharing the same underlying formal model also allows us to apply model-based testing techniques such as in [33], and compliance verification and analysis in [31]. Most of the literature seems to report more ad-hoc engineered and specific solutions, instead, which consequently, are less amenable to formal reasoning.

Formalisms other than constraint automata have been proposed to model the behavior of a service. For example, *session types*, as a formalism for structuring interactions and reasoning over communicating processes, can be applied as a model to describe the behavior of services. A session is defined as a logical unit of information exchanged among participants that specifies the topic of conversation as well as the sequence of the communicated messages [15]. Session types, which can be assigned to end-point processes, describe the user view of an interaction. In [16], the authors specify the behavior of components as session types. As such, they can also be used to describe the behavioral signature of services. We choose to use constraint automata because they can be easily extended to support soft constraints, and, in particular, preferences. Moreover, constraint automata are human readable, and already adopted in related tools, as [30].

In [43] the authors propose a new behavior model for services 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*[9] to describe activities. The authors use an automaton structure to model service behavior. However, similarity-based search is not mentioned in [43].

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

In [47] the authors present an approach that supports service discovery based on structural and behavioral service models, as well as quality constraints and contextual information. Behaviors are matched through a sub-graph isomorphism algorithm, thus vertixes cannot be merged or deleted as in an sub-graph epimorphism.

In [24] the problem of behavioral matching is translated to a graph matching problem, and existing algorithms are adapted for this purpose.

The model presented in [44] 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 [9] the authors propose a crisp translation from interface description of services to classical crisp *Constraint Satisfaction Problems* (*CSPs*). The work in [9] does not consider service behavior and it does not support a quantitative reasoning on similarity/preference involving different services.

In [45] a semiring-based framework is used to model and compose QoS features of WSs. However, no notion of similarity relationship is given in [45].

In [17], the authors propose a novel clustering algorithm that groups names of parameters of service operations into semantically meaningful concepts. These concepts are then leveraged to determine similarity of inputs (or outputs) of service operations.

In [38] 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 [25] 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 [39] show a method of service 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.

Baresi et al. [8] introduce DREAM as an innovative infrastructure for the distributed publication and discovery of WSs. DREAM provides partial solutions for users requests through a set of matchmakers such as WSDL-based Matchmaker and XPath-based matchmaker. The ability to adding new matchmakers gives more flexibility to DREAM, but considering the experimental results, they should improve the precision and recall without affecting the flexibility. Moreover, this version of DREAM could not manage the behavioral and also non-functional aspects of services.

In order to consider QoS metrics as additional criteria to select services from a set of functionally equivalent candidates, we can simply specify QoS properties as meta-attributes. However, the semantics of such schemes is too weak to allow reasoning about QoS properties. In [36], the authors extend constraint automata (CA) with Q-algebras to define *Quantitative Constraint Automata* (*QCA*) to specify the QoS properties of services for an optimized service selection and composition. Also in [36], they introduce *Quasi-Classical Temporal Logic* (*QCTL*) as a logic for reasoning about both behavioral and QoS aspects of services modelled by QCA. Because QCA and our work share constraint automata as their base model, we can extend our soft constraint automata model by adopting the QCA extension of [36], which will enable us to use QCTL logic for a richer form of reasoning about the properties of services.

Preference modelling is an important issue in various fields such as economics, mathematics, infor-

matics, and even psychology. We may have to deal with preferences when we have to make choices on behalf of users [11].

In [21] the authors discuss soft constraint techniques and using the lexicographic order for preferences. Their work is going to be pivotal for the use of soft constraints for dealing with problems, which use some criteria that can be satisfied with a fixed order of importance.

Managing tradeoffs of the QoS preferences has been addressed by [26] using a lexicographic based specification language for expressing the QoS preferences.

It is also addressed in [35] based on a model of lexicographic semi-order. Although lexicographic semi-order seems to work better in our context by considering threshold for each criterion, unfortunately it is not associative.

# 10 Conclusion

We have presented a tool for similarity-based discovery of WS that is able to rank the service descriptions in a database, in accordance with a similarity score matching each with the description of a service desired by a user. The formal framework behind the tool consists of SCA [5], which can represent different high-level stateful software services and queries. We can use SCA to formally reason on queries (e.g., operational similarity for SCA introduced in [5]). The tool is based on implementing approximate operational-similarity evaluation with constraints (see Section 6), which allows to quantitatively estimate the differences between two behaviors. 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: sub-graph isomorphism is not known to be in $P$.

The presented tool has been developed using Java and it can be integrated with the tool presented in [30] with the ultimate goal to also automatically orchestrate the discovered services. We also present a distributed model of our tool to improve performance and scalability of the tool over large scale distributed data sets describing service description data sets.

Our main intent has been to propose a formal framework and a tool with an approximate operational-similarity of behaviors at its heart, not to directly compete against tools such as [39]. Although such tools show higher precision than what we have summarised in Section 5, they do not support behavior specification in their matching. Nevertheless, in the future we plan to refine the performance of our tool by incorporating a semantic similarity-score for operation and parameter names, using an appropriate ontology for services, such as OWL-S.

*References:*

[1] Afsarmanesh, H., Sargolzaei, M., Shadi, M.: Semi-automated software service integration in virtual organisations. Enterprise Information Systems 9(5-6), 528–555 (2015)

[2] Alonso, G., Casati, F., Kuno, H.A., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Data-Centric Systems and Applications, Springer (2004)

[3] 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)

[4] Arbab, F., Rutten, J.J.M.M.: A coinductive calculus of component connectors. In: Recent Trends in Algebraic Development Techniques

(WADT) Revised Selected Papers. LNCS, vol. 2755, pp. 34–55. Springer (2002)

[5] Arbab, F., Santini, F.: Preference and similarity-based behavioral discovery of services. In: ter Beek, M.H., Lohmann, N. (eds.) WS-FM. Lecture Notes in Computer Science, vol. 7843, pp. 118–133. Springer (2012)

[6] Arbab, F., Santini, F., Bistarelli, S., Pirolandi, D.: Towards a similarity-based web service discovery through soft constraint satisfaction problems. In: Proceedings of the 2nd International Workshop on Semantic Search over the Web, Istanbul, Turkey, August 27, 2012. p. 2. ACM (2012)

[7] 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)

[8] Baresi, L., Miraz, M., Plebani, P.: A distributed architecture for efficient web service discovery. Service Oriented Computing and Applications 10(1), 1–17 (2016)

[9] Benbernou, S., Canaud, E., Pimont, S.: Semantic web services discovery regarded as a constraint satisfaction problem. In: Flexible Query Answering Systems, 6th International Conference. LNCS, vol. 3055, pp. 282–294. Springer (2004)

[10] Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. J. ACM 44(2), 201–236 (1997)

[11] Brafman, R., Domshlak, C.: Preference handling-an introductory tutorial. AI magazine 30(1), 58 (2009)

[12] Changizi, B., Kokash, N., Arbab, F.: A Unified Toolset for Business Process Model Formalization. In: Proceedings of FESCA 2010 (2010)

[13] Cheatham, M., Hitzler, P.: String similarity metrics for ontology alignment. In: The Semantic Web - ISWC 2013 - 12th International Semantic Web Conference. Lecture Notes in Computer Science, vol. 8219, pp. 294–309. Springer (2013)

[14] Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web services description language (wsdl) version 2.0 part 1: Core language. W3C recommendation 26, 19 (2007)

[15] Dezani-Ciancaglini, M., DeLiguoro, U.: Sessions and session types: an overview. In: Web Services and Formal Methods, pp. 1–28. Springer (2010)

[16] Dezani-Ciancaglini, M., Padovani, L., Pantovic, J.: Session type isomorphisms. In: PLACES. pp. 61–71 (2014)

[17] 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)

[18] Dong, X., Halevy, A., Madhavan, J., Nemes, E., Zhang, J.: Similarity search for web services. In: Proceedings of the Thirtieth international conference on Very large data bases-Volume 30. pp. 372–383. VLDB Endowment (2004)

[19] Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer Publishing Company, Incorporated, 1st edn. (2009)

[20] Fox, E.A., Shaw, J.A.: Combination of multiple searches. NIST SPECIAL PUBLICATION SP pp. 243–243 (1994)

[21] Gadducci, F., Hölzl, M., Monreale, G.V., Wirsing, M.: Soft constraints for lexicographic orders. In: Mexican International Conference on Artificial Intelligence. pp. 68–79. Springer (2013)

[22] Gadducci, F., Santini, F.: Residuation for bipolar preferences in soft constraints. Inf. Process. Lett. 118, 69–74 (2017)

[23] Gay, S., Fages, F., Martinez, T., Soliman, S., Solnon, C.: On the subgraph epimorphism problem. Discrete Applied Mathematics 162, 214–228 (2014)

[24] 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)

[25] Hau, J., Lee, W., Darlington, J.: A semantic similarity measure for semantic web services. In: Web Service Semantics Workshop at WWW (2005)

[26] Iordache, R., Moldoveanu, F.: A conditional lexicographic approach for the elicitation of qos preferences. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems". pp. 182–193. Springer (2012)

[27] Iosup, A., Sonmez, O., Anoep, S., Epema, D.: The performance of bags-of-tasks in large-scale distributed systems. In: Proceedings of the 17th international symposium on High performance distributed computing. pp. 97–108. ACM (2008)

[28] Järvelin, K., Kekäläinen, J.: Ir evaluation methods for retrieving highly relevant documents. In: Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval. pp. 41–48. ACM (2000)

[29] 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: Paoli, F.D., Pimentel, E., Zavattaro, G. (eds.) ESOCC. Lecture Notes in Computer Science, vol. 7592, pp. 1–16. Springer (2012)

[30] Jongmans, S.S.T., Santini, F., Sargolzaei, M., Arbab, F., Afsarmanesh, H.: Orchestrating web services using reo: from circuits and behaviors to automatically generated code. Service Oriented Computing and Applications pp. 1–21 (2013)

[31] Kokash, N., Arbab, F.: Formal behavioral modeling and compliance analysis for service-oriented systems. In: de Boer, F.S., Bonsangue, M.M., Madelain, E. (eds.) FMCO. Lecture Notes in Computer Science, vol. 5751, pp. 21–41. Springer (2008)

[32] Kokash, N., Arbab, F.: Formal design and verification of long-running transactions with extensible coordination tools. IEEE T. Services Computing 6(2), 186–200 (2013)

[33] Kokash, N., Arbab, F., Changizi, B., Makhnist, L.: Input-output conformance testing for channel-based service connectors. In: Aceto, L., Mousavi, M.R. (eds.) PACO. EPTCS, vol. 60, pp. 19–35 (2011)

[34] Kopecky, J., Gomadam, K., Vitvar, T.: hrests: An html microformat for describing restful

web services. In: Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on. vol. 1, pp. 619–625. IEEE (2008)

[35] Mariotti, M., Manzini, P., et al.: Choice by lexicographic semiorders. Theoretical Economics 7(1) (2012)

[36] Meng, S., Arbab, F.: QoS-driven service selection and composition using quantitative constraint automata. Fundam. Inform. 95(1), 103–128 (2009)

[37] Meng, S., Arbab, F.: A model for web service coordination in long-running transactions. In: SOSE. pp. 121–128. IEEE (2010)

[38] Pan, J.Z., Stamou, G., Stoilos, G., Taylor, S., Thomas, E.: Scalable querying services over fuzzy ontologies. In: Proceedings of World Wide Web. pp. 575–584. WWW '08, ACM, New York, NY, USA (2008)

[39] Plebani, P., Pernici, B.: Urbe: Web service retrieval based on similarity evaluation. IEEE Trans. on Knowl. and Data Eng. 21(11), 1629–1642 (2009)

[40] Richardson, L., Ruby, S.: RESTful web services. " O'Reilly Media, Inc." (2008)

[41] Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Middleware 2001. pp. 329–350. Springer (2001)

[42] le Clément de Saint-Marcq, V., Deville, Y., Solnon, C.: Constraint-based graph matching. In: Gent, I.P. (ed.) CP. Lecture Notes in Computer Science, vol. 5732, pp. 274–288. Springer (2009)

[43] Shen, Z., Su, J.: Web service discovery based on behavior signatures. In: Proceedings of the 2005 IEEE International Conference on Services Computing - Volume 01. pp. 279–286. SCC '05, IEEE Computer Society, Washington, DC, USA (2005)

[44] Toch, E., Gal, A., Reinhartz-Berger, I., Dori, D.: A semantic approach to approximate service retrieval. ACM Trans. Internet Technol. 8(1) (Nov 2007)

[45] Zemni, M.A., Benbernou, S., Carro, M.: A soft constraint-based approach to QoS-aware service selection. In: Service-Oriented Computing - 8th International Conference, ICSOC 2010. LNCS, vol. 6470, pp. 596–602 (2010)

[46] Zimmer, P., Zimmer, M., Zimmer, B.: Fizzim an open-source fsm design environment. Enterprise Information Systems 9(5-6), 528–555 (2014)

[47] Zisman, A., Dooley, J., Spanoudakis, G.: Proactive runtime service discovery. In: Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 1. pp. 237–245. SCC '08, IEEE Computer Society, Washington, DC, USA (2008)