Web Service Discovery Based on Behavior Signatures

Zhongnan Shen
Department of Computer Science
University of California
Santa Barbara, CA 93106-5110
szn@cs.ucsb.edu

Jianwen Su*
Department of Computer Science
University of California
Santa Barbara, CA 93106-5110
su@cs.ucsb.edu

Abstract

Web service discovery is a key problem as the number of services is expected to increase dramatically. Service discovery at the present time is based primarily on keywords, or interfaces of web services through the use of ontology. We argue that "behavior signatures" as operational level description should play an important role in the service discovery process. In this paper, we propose a new behavior model for web services using automata and logic formalisms. Roughly, the model associates messages with activities and adopts the IOPR model in OWL-S to describe activities. A new query language is developed to express temporal and semantic properties on service behaviors. Query evaluation algorithms are developed; in particular, an optimization approach using RE-tree and heuristics is shown to improve the performance. Specifically, experimental results show that the use of RE-tree reduces query evaluation time by an order of magnitude and with heuristics it enhances the performance by two orders of magnitude. This is clearly an encouraging starting point.

1 Introduction

Web services have received ever increasing interests from e-commerce, science, telecommunication applications, and research communities across different areas, evident by recent activities in the research communities (ICSOC, ICWS, SCC, etc.). A fundamental problem of web services concerns service discovery. For example, service composition may start with locating appropriate existing services that can be used and integrated together. Requirements used to select services may focus on different aspects of web services, varying from service categories to service semantics, from service interfaces to service behaviors. Current work on service discovery focuses on functional descriptions, we argue that behavior signatures [11] as operational level description should play an important role in service discovery.

*Supported in part by NSF grant IIS-0101134.

To motivate the discussion, we consider the *Purchase* service example in Fig. 1. The composite service needs to find a *Purchase* service satisfying the following requirements.

R_1: A service in the purchase service category and provided by some company A.

R_2: A service that provides a WSDL operation *PlaceOrder* with input message *PurchaseRQ* and output message *AcceptRS*.

R_3: A service whose behaviors satisfy all of the following properties:

- (P_1) *Charging* activity is before *Shipping* activity.
- (P_2) To purchase a product, the requester first needs to log into the system and finally log out of the system.
- (P_3) The product can be out of stock, and the credit card is not charged in this case.
- (P_4) The credit card is charged at most once.
- (P_5) The service accepts American Express.

Among the search criteria, R_1 can be expressed in the UDDI framework [23], and R_2 can be dealt with using the service discovery techniques in [13, 17, 4, 8] based on service interfaces described by WSDL[18]. However, conditions P_1 to P_5 in R_3 are properties on the behaviors of a web service and cannot be captured by existing infrastructure for service discovery. Fig. 1(a) shows properties P_1 and P_2, and Fig. 1(b) describes (the flow of) an available *Purchase* service. Symbols on edges outgoing from circles are messages exchanged during the execution of the service (the symbol "!" stands for output messages and "?" for input messages), while symbols on edges outgoing from triangles are activities performed in the service. (Notations will become clear in Section 3.) The available service satisfies P_2, but fails to satisfy P_1 because *Shipping* occurs before *Charging* activity.

Fig. 2 shows another example for service discovery. In this example, a *Plane Ticket Booking* service is needed. Any qualified service must satisfy two requirements concerning the maximum number of tickets

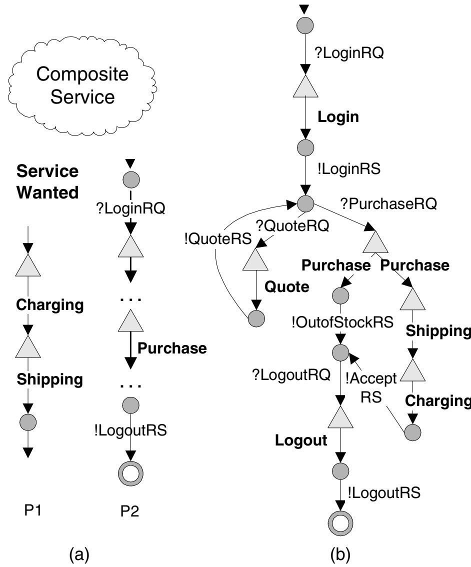

Figure 1. A Purchase Service Example

Service wanted: Plane Ticket Booking
Requirements:
(1) the service can process any number of tickets in one request, and
(2) flight destinations can be anywhere in America.

Services Available:

`BookTicket`	`TicketReservation`
Precondition:	*Precondition:*
NumberOfTickets < 10	NumberOfTickets = any
Destination = West of USA	Destination = any
⋮	⋮

Figure 2. Plane Ticket Booking Service

booked in one request, and flight destinations (respectively). The service `BookTicket` fails to satisfy the constraints because it can only handle book requests with less than 10 tickets and flight destinations are also limited. `TicketReservation` is a candidate service.

In the Plane Ticket Booking example, the requirements on qualified services describe preconditions of these services. However, this type of queries cannot be done by current discovery mechanisms. It is necessary to develop a new service behavior model which is also capable of describing input and output conditions as a basis for service discovery.

In a broad sense, behavior signatures of a web service include conversations (sequences of messages exchanged) of this service, events or activities performed by this service, and semantics of this service (e.g., in terms of input, output, precondition, and result). Behavior signatures of web services are gaining more and more concerns, for example, WSCL [24] and WSCDL [20] attempt to capture conversations between interacting services, WSMO [25] works on semantic web services, and OWL-S [22] introduces preconditions and results to each activity in the flow of a service. With the development of these standards, we envision that more services could become available with their behavior signatures exported. The service behavior model presented in this paper can be abstracted from these standards and be used to facilitate service discovery, maximally reusing existing services. Clearly, such an approach to service discovery is promising and practical.

In this paper, we present a new behavior model for service processes, which associates messages exchanged between participants with activities performed within the service. The "IOPR" (Input, Output, Precondition, Result) model for service activities from OWL-S [22] is used in our model as a first step towards capturing service semantics. The model also extends the finite state automata model of [3, 11] to represent state changes caused by message exchange and activity execution. Our model assumes the existence of a standard vocabulary (domain ontology) for messages and activities and focus on key techniques in the discovery mechanism. We note that domain based ontologies are a practical method being adopted by, e.g., the travel industry, and developing such ontologies is itself a significant task and beyond the scope of our investigation.

Based on the behavior model, a query language is proposed to capture most properties on behavior signatures. These properties include temporal features on sequences of messages or activities, as well as semantics of activities (IOPR). Evaluation algorithms are developed for the query language. Given a set of web services, the RE-tree index structure [5] based on R-tree is constructed to accelerate the search process. A heuristic search algorithm is used for size-constrained search. Preliminary experimental results are also presented in this paper.

The remainder of this paper is organized as follows. The service model is defined in Section 3, after presenting necessary notions in Section 2. Section 4 described the new query language. Evaluation algorithms and the index structure are proposed in Section 5, and experimental results are presented in Section 6. Section 7 discusses the related work and Section 8 concludes the paper.

2 Messages and Activities

In this section, we introduce basic concepts needed for the technical presentation. A web service process is modeled as an extended nondeterministic finite automaton, which exports the behavior signatures of this service. It is a hybrid model in terms of combining messages and activities. This model extends Mealy service model [3] and Roman model [1]: activities in Roman model are associated with input message classes, output message classes, preconditions and results; and a run of a web service process is interpreted in terms of receiving/sending messages and performing activities. This model captures two different types of states in a web service process: states for sending/receiving messages and states for performing ac-

tivities. This separation describes not only the observable conversations (messages exchanged) between cooperating services, but also activity sequences performed inside the service. Before we give the formal definition for this model, we first define message classes and activity profiles.

Messages are data exchanged between web services or between activities within a web service. Intuitively, a message has structured contents and belongs to a message class. We assume the existence of *class names*, *attribute (names)*, and primitive *types* such as integers, strings, and boolean.

DEFINITION A *message class* is an expression $C(A_1{:}\tau_1, ..., A_n{:}\tau_n)$, where C is the class name, $C.A_i$'s are attributes, and τ_i's are types. A *message* in the class $C(A_1{:}\tau_1, ..., A_n{:}\tau_n)$ is a mapping m from $\{C, A_1, ..., A_n\}$ such that $m(C)$ is the identifier of the message, $m(C.A_i)$ is a value in the domain of τ_i for each $1 \leqslant i \leqslant n$.

When the context is clear, a message class $C(\cdots)$ is simply referred to as C. Our model assumes the structure of a message class to be flat. While extension to allow hierarchical structures can always be done, the flat structure simplifies the presentation. A response message class *BookPlaneRS* from the *BookPlaneTicket* service may include attributes *FlightNumber*, *NumberofTickets*, *DateTime*, etc.

Clearly, web services have to access databases or data sources. One approach is to have databases as a part of a web service to store not only the data needed for the process, but also the information (e.g., states) about process itself [7]. The focus of this paper is on web service behaviors, our approach is to allow a web service to consult databases.

A *database class* is an expression $R(A_1{:}\tau_1, ..., A_m{:}\tau_m)$, where R is the class name, $R.A_i$'s are attributes, and τ_i's are types. An *object* in class R is a mapping o from $\{R, A_1, ..., A_m\}$ such that $o(R)$ is the identifier of the object, $o(R.A_i)$ (or $R.o.A_i$) is a value in the domain τ_i for each $1 \leqslant i \leqslant m$.

In OWL-S, the semantics of a web service is described by using a profile. The primary component in a profile concerns the description of the input, output, precondition, and result (IOPR). In our model, we combine IOPRs with Mealy machines for messages.

Intuitively, a precondition for an activity is a set of conjunctive conditions which must be satisfied before the activity is invoked so that this activity can be performed correctly. Preconditions are conjunctive conditions over input message classes and database classes. The syntax of preconditions is defined below, where nonterminals are in italic font, terminals are in typewriter-style.

precondition ::= *andcond*
andcond ::= `(` *orcond* `)` | `(` *orcond* `)` ∧ *andcond*
orcond ::= *condition* | *condition* ∨ *orcond*
condition ::= `type(`*C, A*`)` = *datatype* | *expr comp expr*
expr ::= *term* | *expr op expr*
term ::= *C.A* | *R.o.A* | *dataliteral*

In the definition of precondition, C and R are input message classes and database classes, $C.A$ is the value of message attribute A of input message class C, and $R.o.A$ is a reference to the value of attribute A of object variable o in database class R. The term *dataliteral* is used for data values, i.e., constants. The built-in function `type` is a function returning the type of an attribute in a message class. Comparison operations *comp* include (in)equality and order, operators *op* are $+$, $-$, and multiplication.

A result for an activity is a set of effects that this activity causes on the real world. Results are conjunctive conditional statements over input message classes, output message classes, and database classes.

result ::= `(` *andcond* → *effect* `)` |
`(` *andcond* → *effect* `)` ∧ *result*
effect ::= `(` *pcondition* `)` | `(` *pcondition* `)` ∧ *effect*
pcondition ::= `type(` *C',A* `)` =*datatype* |
`incr(` *R.o.A* `)` *comp expr'* |
`decr(` *R.o.A* `)` *comp expr'* |
expr' comp expr'
expr' ::= *term'* | *expr' op expr'*
term' ::= *C.A* | *C'.A* | *R.o.A* | *dataliteral*

In the definition of result, C' is an output message class and $C'.A$ is the value of message attribute A of output message class C'. The two built-in functions `incr`/`decr` in results are used for quantitative increment/decrement of an object's attribute in the database.

DEFINITION An *activity profile* is defined as $T(I, O, S, P, R)$, where

- T is the name of the activity,
- I is a tuple of input message classes of this activity,
- O is a tuple of output message classes of this activity,
- S is a set of database classes in this activity,
- P is the precondition of this activity,
- R is the result of this activity.

An *activity* is performed when the precondition P holds. It consumes a tuple of input messages which are instances of input message classes in I, and generates another tuple of output messages which are instances of output message classes in O. The result R holds after the activity.

For the *BookPlaneTicket* service, its profile can be
T: BookPlaneTicket.
I: (BookRQ(FltNumber, NumberofTickets, Time,...)).
O: (BookRS(FltNumber, TicketsBooked, Time,...)).
S: {Ticket(FltNumber, TicketAvail,...)}.
P: type(BookRQ, FltNumber) = string
∧ type(BookRQ, NumberofTickets) = int
∧ BookRQ.NumberofTickets $<$ 10.
R: BookRQ.FltNumber=Ticket.o.FltNumber ∧
Ticket.o.TicketAvail $<$ BookRQ.NumberofTickets →
BookRS.TicketBooked=0∧decr(Ticket.o.TicketAvail)=0.

The precondition of this activity describes the datatype of attributes in the input message, and states that it can only handle book requests with number of tickets less than 10. The result says that if the flight information is in the database and there are not enough tickets available, then no ticket should be booked for the requester.

3 Modelling Service Behaviors

In this section we outline the model for web service behaviors. The model combines the state machine approach for handling input and output messages and the OWL-S approach for describing semantics. In particular, the semantics are represented in the form of activity profiles defined in the previous section. A web service under this model may consist of many activities that have input and output. Clearly one has to address the issue of how to pass information between different activities. We use "internal" messages to connect the input and output of activities. Thus we can classify the set of message classes in a web service into three categories: *input* messages that are only consumed by the service, *output* messages that are only generated by the services, and *internal* messages that are both generated and consumed by the service. Note we do not assume any properties [3] (e.g., bounded length) on channels between web services as we are not addressing composition issues here.

DEFINITION A *web service* is a system $\mathcal{A} = (Q, q_0, F, P, \Sigma, M_i, M_o, \Delta)$, where

- Q is a finite set of *states* that is partitioned into *messaging* states Q_m and *activity* states Q_a,
- $q_0 \in Q$ is the *initial* state and $F \subseteq Q$ is the set of *final* states,
- P is a finite set of profiles of activities with distinct names (we will also use P to denote the set of activity names),
- Σ is a finite set of message classes that is partitioned into three sets: $\Sigma_{\text{in}}, \Sigma_{\text{out}}$, and Σ_{var} for the input, output, and internal message classes, resp.,
- M_i, M_o are partial mappings such that for each activity α in P with n input messages and m output messages, and each state $q \in Q_a$, $M_i(\alpha, q) \in (\Sigma_{\text{in}} \cup \Sigma_{\text{var}})^n$ are the input messages for α if defined, and $M_o(\alpha, q) \in (\Sigma_{\text{out}} \cup \Sigma_{\text{var}})^m$ are the output messages for α if defined, and finally
- Δ is a set of transitions of the following form:
  – $(q_1, ?m, q_2)$, where $q_1 \in Q_m, q_2 \in Q$, $m \in \Sigma_{\text{in}}$. The transition changes the state upon receiving a message in class m,
  – $(q_1, !m, q_2)$, where $q_1 \in Q_m, q_2 \in Q, m \in \Sigma_{\text{out}}$. The transition changes the state after sending a message in class m,
  – (q_1, α, q_2), where $q_1 \in Q_a$, $q_2 \in Q$, α is an activity in P. The transition changes the state after performing the activity α.

Our model distinguishes two kinds of states: messaging states in which the service can send or receive messages, and activity states in which the service can execute activities. Message sequences describe outside interactions while activity sequences characterize inside executions. Input messages are messages from the service requester, and output messages are messages sent to the service requester. Internal messages are those exchanged internally between activities in a web service. For an activity, its input messages may be from the requester (input messages) or previous activities (internal messages), and its output messages may go to the requester (output messages) or succeeding activities (internal messages).

Fig. 1(b) in Section 1 gives the automaton representation of the *Purchase* service. Internal messages are not shown in the automaton. Circle states stand for message states, and triangle states stand for activity states. Input/output messages are indicated by ?/! resp. The automaton in the example is nondeterministic.

We denote the *language* of a service $\mathcal{A}$ to be the set of all words accepted by $\mathcal{A}$.

4 A Query Language

In this section, we present a query language for web service behaviors. The language is based on first-order logic, and focuses on properties on behavior signatures of web services. The output for a query is a set of services whose behaviors satisfy these constraints. Properties or constraints include temporal properties of messages exchanged or activities performed, and preconditions or results of activities.

DEFINITION Let Σ be a set of message class names, P a set of activities (profiles), and "#" be a special symbol (denoting a wildcard matching any single message/activity). A *behavior pattern* over Σ and P is a regular expression over $\Sigma \cup P \cup \{\#\}$, i.e., expressions built from the set using concatenation (ee'), union ($e|e'$), and closure (e^*).

A behavior pattern specifies a segment of an entire behavior sequence. A behavior pattern composes only of message classes is called a *message pattern*, while a pattern composed only of activity names is called an *activity pattern*. For example, "LoginRQ Login $\#^*$ Logout LogoutRS" is a behavior pattern, whereas "PurchaseRQ (OutofStockRS | AcceptRS)" is a message pattern and "(Purchase Shipping Charging)*" is an activity pattern.

Behavior queries are now defined below.

Pattern Queries. A primitive pattern query is one of the following, where e, e', and e'' are behavior patterns:

- *Starts*(e), *Ends*(e) : pattern e is the starting, (resp.) ending sequence,
- *Contains*(e) : pattern e is a sub-sequence,
- *Before*(e, e'), *After*(e, e') : pattern e occurs before, (resp.) after pattern e', and
- *Between*(e, e', e'') : pattern e'' occurs between pattern e and pattern e'.

A pattern query is either a primitive pattern query, or a boolean combination of primitive pattern queries.

Behavior Queries. A behavior query is an expression in one of the following forms, where ρ is a pattern query, *ALL*ρ and *EXISTS*ρ are universally and existentially quantified behavior queries, and q, q_1, q_2 are behavior queries.

- *ALL*ρ and *EXISTS*ρ are queries,
- $q_1 \wedge q_2$ and $q_1 \vee q_2$ are queries, and
- $q \wedge Precond(\alpha, c)$, $q \wedge Result(\alpha, c)$ are queries.

Predicates $Precond(\alpha, c)$ and $Result(\alpha, c)$ are described as following, where α is an activity (name) and c is a constraint on preconditions or results. c itself is a precondition if it is a constraint on preconditions, and is a result if it is a constraint on results.
– *Precond*(α, c) is a predicate on the precondition of α. *Precond*(α, c) is true if $c \rightarrow p$, where p is the precondition of α;
– *Result*(α, c) is a predicate on the result of activity α. *Result*(α, c) is true if $r \rightarrow c$, where r is the result of α.

The six primitive predicates are used to capture temporal characteristics of behaviors of web services. It describes the order in which messages are exchanged between participating web services and activities are performed within a web service. The other two predicates, $Precond(\alpha, c)$ and $Result(\alpha, c)$, are designed to express requirements on inputs, outputs, preconditions, and results of activities.

Pattern queries in the language are interpreted by enactment skeletons, while behavior queries are interpreted by behavior signatures which are defined below.

DEFINITION Let $\mathcal{A}$ be a web service. An *enactment skeleton* E is a finite sequence of message classes and activity names in a terminating run of the service $\mathcal{A}$. The *behavior signature* B of a web service $\mathcal{A}$ is a 3-tuple (L, M, P), where L is a set of enactment skeletons, M is a finite set of message classes which appear in L, and P is a finite set of activity profiles.

A behavior signature describes the behavior of a web service. It can be mapped to a finite state machine with L being the language accepted by this automaton. The query *ALL*ρ is true if every enactment skeleton in L satisfies the pattern query ρ, and the query *EXISTS*ρ is true if there is an enactment skeleton in L which satisfies ρ.

Recall the five behavioral properties P1 to P5. Now they can be simply expressed in the query language as below.

1. $Charging$ activity happens before $Shipping$ activity.
   ALL(*Before*($Charging$, $Shipping$)).
2. A service requester needs to log into and log out of the system for a purchase.
   ALL(*Starts*($LoginRQ$)$\wedge$ *Finishes*($LogoutRS$)).
3. Products can be out of stock, and card is not charged.
   EXISTS(*Contains*($OutofStockRS$)$\wedge$ $\neg$*Contains*($Charging$)).
4. The credit card is charged at most once.
   ALL($\neg$*Contain*($Charging$ #* $Charging$)).
5. The service accepts credit card American Express.
   Precond($Purchase$, $PurchaseRQ.cardtype$="AE").

5 Query Evaluation

In this section, we present algorithms for evaluating queries on behavior signatures. Algorithms are developed for checking if a service satisfies a query and for evaluating the query against a large set of web services. First we describe the semantic of saying that a web service satisfying a query. Our focus is on evaluation of skeleton queries which only contain sub-queries of *ALL*ρ and *EXISTS*ρ. We then consider the problem of evaluating queries against a set of services, and illustrate how to use RE-tree [5] and heuristic to speedup the search process. Experimental results show that RE-tree improves significantly the search performance for web services, and that the heuristic yields another significant improvement.

5.1 Evaluation Algorithm

Given a web service $\mathcal{A} = (Q, q_0, F, P, \Sigma, M_i, M_o, \Delta)$ and a behavior query q, we first break q into sub-queries of the form of *ALL*ρ, *EXISTS*ρ, *Precond*(α, c), and *Result*(α, c). Thus q is a boolean combination of these sub-queries. Once each sub-query is evaluated, the final value for q can be obtain trivially. *Precond*(α, c) and *Result*(α, c) can be transformed to constraint satisfaction problem. Formally, $c \rightarrow p$ ($r \rightarrow c$) is true if $c \wedge \neg p$ ($r \wedge \neg c$) is unsatisfiable. For algorithms which check the constraint satisfiability, please refer to [14]. In this paper, we focus on how to effectively evaluate *ALL*ρ and *EXISTS*ρ, which are called skeleton queries evaluation.

The main idea of the evaluation algorithm for skeleton queries is that for each pattern query ρ in it, we generate a regular language $L(\rho)$ such that $L(\rho)$ only contains enactments that satisfy ρ. $L(\rho)$ is then compared with the language $L(\mathcal{A})$ generated by service $\mathcal{A}$. If the intersection of $L(\rho)$ and $L(\mathcal{A})$ is not empty, the query *EXISTS*ρ is satisfied by $\mathcal{A}$. If $L(\mathcal{A})$ is a subset of $L(\rho)$, then the query *ALL*ρ is satisfied.

We don't give the algorithm which generates the language for ρ here due to the space limitation. The language of pattern query $Before(Charging, Shipping)$, for example, can be $\Sigma^* Shipping(\Sigma^* - \Sigma^* charging \Sigma^*)$, where Σ is the entire alphabet. The time complexity for this algorithm is exponential to the length of the skeleton query. The algorithm evaluating skeleton queries is given below.

Algorithm 1 *EvalSkeletonQuery*($L(\mathcal{A})$,*EXISTS*ρ *or ALL*ρ)
Input: $L(\mathcal{A})$ *is the language accepted by the service* $\mathcal{A}$
  ALLρ *or EXISTS*ρ *is a skeleton query ;*
Output: true if $L(\mathcal{A})$ *satisfies the skeleton query;*
Steps:
  ALLρ: *if* $(L(\mathcal{A}) \subset L(\rho))$ $B =$ *true; else* $B =$ *false;*
  EXISTSρ: *if* $(L(\mathcal{A}) \cap L(\rho) = \varnothing)$ $B =$ *false; else* $B =$ *true;*
  return B.

The algorithm requires automata intersection, complement and empty test. It can be shown that the complexity is exponential and can be reduced to PSPACE if automata are deterministic.

5.2 Search Algorithm

With a set of services S, the search results for *EXISTS*ρ and *ALL*ρ are denoted by *EXISTS*$\rho(S)$ and *ALL*$\rho(S)$ respectively. We define a select operation σ:

$$\sigma_\rho(S) = \{s \mid s \in S \wedge L(\rho) \cap L(s) \neq \varnothing\}.$$

The σ operation takes a pattern query, and selects from S all the services whose languages have accepted words which satisfy ρ. Then, we have *EXISTS*$\rho(S) = \sigma_\rho(S)$, and *ALL*$\rho(S) = S - \sigma_{\neg\rho}(S)$. With the number of web services increasing dramatically, it is impractical to do a sequential search in S with a big size. To solve this problem, we use RE-tree to build up index on the select operation.

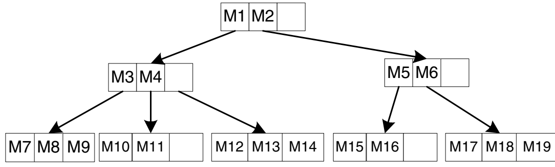

Figure 3. An RE-Tree Example

RE-tree is developed by Chan, Garofalakis, and Rastogi [5] as an index structure for regular expressions. An RE-tree is a R-tree [10] with each node entry to be a regular expression or a finite state automaton. Fig. 3 gives an example of RE-tree. In this example, the following containment properties hold: $L(M3) \cup L(M4) \subseteq L(M1)$, $L(M7) \cup L(M8) \cup L(M9) \subseteq L(M3)$, etc. Automata in internal nodes are called bounding automata. For details about RE-tree, please refer to [5]. Different from [5] in which RE-tree is used to search automata which contain a certain word, we use an automaton as a query, which makes the search algorithm complex but more interesting.

Given a skeleton query *ALL*ρ or *EXISTS*ρ, search begins with the root node, proceeds in a top-down manner, and traverses all the potential paths of the index structure. When searching an internal node N, the search language $L(\rho)$ is computed against each bounding automaton such that if the intersection of these two automata is empty, the search along that path terminates; if the intersection equals to the bounding automaton, then all data in the subtree of this bounding automaton are put in the result set and the search on that path terminates as well, otherwise, the search continues to the node at the next level. When the search reaches a leaf node, the *EvalSkeletonQuery()* algorithm is executed on each data in the node, and if it returns true, this data is put in the result set. The algorithm is not given here.

In the search process mentioned above, we need to check each potential path to get all the data which satisfy the query. However, in a real search engine, it may only need to give the first m services, where m is a reasonable small number. In this case, we don't need to traverse the whole tree to get the search result. Once we have enough number of services, the search can be terminated. Different strategies can be introduced here to speedup the search process. We propose a heuristic search algorithm which gives search priorities to these most potential branches. Specifically, for an internal node N, all the bounding automata in this node are sorted according to a certain metric, the branch whose bounding automaton has the highest rate is searched first, then the second highest rate, and so on. The metric selected should reflect the potentials of containing qualified services.

For each $n \in \mathbb{N}$, the *size* of a language is defined as $|L(M)| = \Sigma_{i=1}^{n}(|L_i(M)|)$, where $|L_i(M)|$ is the number of words of length i in $L(M)$. If M is a bounding automation in node N, for skeleton query *EXISTS*ρ, the metric can be the cardinality of $L(\rho) \cap L(M)$; however, for skeleton query *ALL*ρ, it is more likely that the language of automata under M can be a subset of $L(\rho)$ if $L(\rho)$ is close to $L(M)$. Therefore, the reverse of size of $|L(M) - L(\rho)|$ can be a good metric. The parameter n is preselected.

Algorithm 2 *SizeConstrainedSearch*(N, q, n, t)

Input: N is a node in the RE-tree, q=EXISTSρ or ALLρ, n is the number of services wanted, t is the number of services found;
Output: a set R of automata which satisfy q;
Steps:

```
R = ∅;
if (N is a leaf node)
  for (each entry M) {
    if (EvalSkeletonQuery(L(M), q))  R = R ∪ {M};
    if (|R| + t >= n)  return R; }
if (N is an internal node) {
  if (q =EXISTSρ) {
    compute r_i = |L(M_i) ∩ L(ρ)| for ∀M_i in N;
    L = a descending list of M_i according to r_i; }
  if (q =ALLρ) {
    compute r_i = |L(M_i) − L(ρ)| for ∀M_i in N;
    L = an ascending list of M_i according to r_i; }
  while (L is not empty) {
    remove the first element M_j from L;
    if (L(ρ) ∩ L(M_j) = ∅)   continue;
    if (L(M_j) ⊆ L(ρ)){
      R = R∪ {all data in the subtree M_j points to};
      if (|R| + t >= n)  return R;  continue; }
    R = R∪ SizeConstrainedSearch(N', q, n, t + |R|),
        where N' is the child node of M_j;
    if (|R| + t >= n)  break; }}
 return R.
```

6 Experimental Evaluation

With index structure and search algorithm given in the previous section, we show by experiments that it is realistic to do search discovery on behavior signatures although state machine operations such as language intersection, containment test, are involved. To show the effectiveness of the RE-tree index structure, we examine the performances of two search approaches: RE-tree depth-first approach, and RE-tree big branch first approach. RE-tree depth-first approach searches the RE-tree in depth-first manner and at an

internal node, it traverses each sub-tree in sequential order. RE-tree big branch first approach uses depth-first search as well, but instead of searching each subtree sequentially, it employs the heuristic in the previous subsection. Our experiments shows that the big branch first algorithm outperforms depth-first search algorithm, and both of them can reduce search computation up to an order of magnitude, compared to sequential search.

In the experiments, we use synthetic data set. Each automaton in the data set is randomly generated with the number of states up to 10 because current web services are relatively small with simple operations. Automata are clustered into n groups with different alphabets Σ, which is intended to simulate different service categories. Each automaton in a group is associated with a hot alphabet $\Sigma'(\Sigma' \subset \Sigma)$, such that the symbol on a transition is drawn from Σ' with a probability of η and drawn from Σ with a probability of $(1-\eta)$. This hot alphabet is further to simulate services in different subcategories. Each automaton is mapped to its hot alphabet as follows: Σ is divided into m disjoint hot alphabets with equal number of symbols, then each automaton is randomly mapped to one of the m alphabets. Edges between states are also generated randomly. Queries in the experiments are *EXISTS*ρ and *ALL*ρ. The corresponding automaton for $L(\rho)$ is constructed as follows: 1) use the algorithm of generating data set to produce an automaton M with a relatively small number of states and edges; 2) extend $L(M)$ to $\Sigma' \cdot L(M)$ which simulates the *Ends*(e) predicate, or $L(M) \cdot \Sigma'$ which simulates the *Starts*(e) predicate, or $\Sigma' \cdot L(M) \cdot \Sigma'$ which simulates the other four predicates. Table 1 summarizes the parameters used in our experiments.

Table 1. Experimental Parameters

Parameter	Value
Size of alphabet	$\lvert\Sigma\rvert = 200$
Number of groups	$n = 10$
Number of hot alphabets in a group	$m = 4$
Number of states	up to 10
Hot probability	$\eta = 0.6$
Size of data set	1000, 2000
Number of queries	1000
Number of tree levels	5 to 6

We first test the performance of the RE-tree depth-first search by varying the data set size. The performance results are presented in terms of number of automata comparisons incurred by the RE-tree depth-first search. The number of comparisons is measured with respect to the result percentage, which is defined as the ratio of the result set size to the data set size, i.e., $n_{\text{result}}/n_{\text{dataset}}$.

Fig. 4 and 5 show the performance results of *EXISTS*ρ and *ALL*ρ queries respectively as the data set size is varied. We get similar results as [5]. The RE-tree approaches can reduced the number of automata comparisons by a factor from 5 up to 25 for both kinds of queries, compared to sequential search. This shows that the RE-tree can prune search space effectively. When the result set is small, say

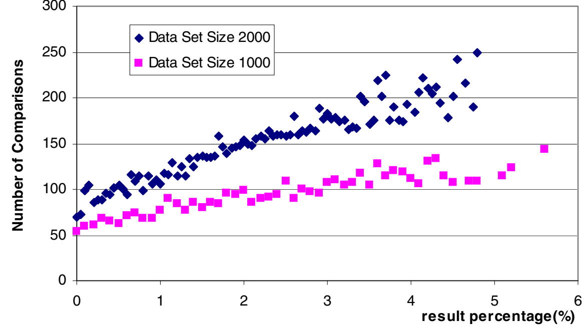

Figure 4. Comparisons for *EXISTS*ρ queries

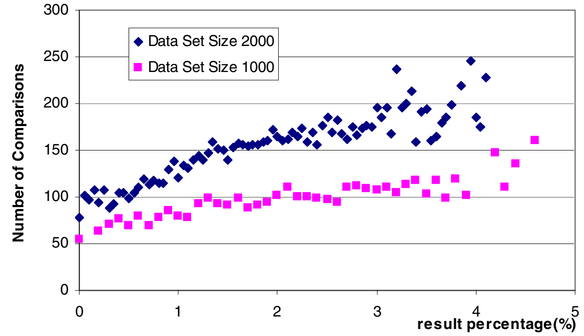

Figure 5. Comparisons for *ALL*ρ queries

1% of the data set, the speedup gained from the RE-tree index structure makes our search mechanism realistic and practical (less than 50 comparisons out of 1000 data).

We also test the effectiveness of the heuristic size-constrained search algorithm. We compare its performance with depth-first size-constrained search approach, and the performance is presented in terms of search speedup defined as $N_{\text{depth}}/N_{\text{heuristic}}$. N_{depth} and $N_{\text{heuristic}}$ are the average number of comparisons in the depth-first search and in the big branch first search respectively. The size of the result set is constrained to a relatively small number m which varies from 1 to 100 in our experiments. The size of the data set is 2000, and for each m, 100 queries are performed.

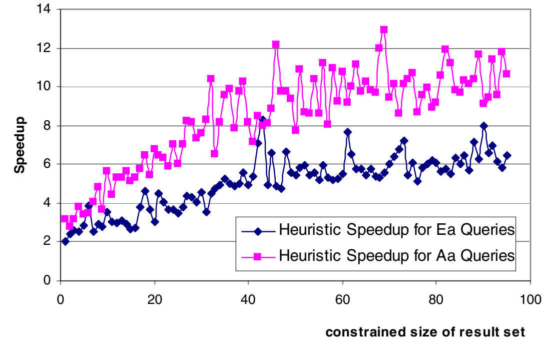

Figure 6. Heuristic vs. Depth-first search

Fig. 6 shows the performance results of $E\alpha$ and $A\alpha$ queries as the constrained size of result set is varied. As we can see from this figure, the heuristic algorithm outperforms

the normal depth-first algorithm by a factor of up to 10. The number of automata comparisons for depth first search first increases with constrained size m, then becomes relatively stable. This is because when you want more services with a small number m, the algorithm needs to search more space to get the services. However, when m is big enough such that the algorithm needs to traverse the whole tree to get the result set, the number of comparisons becomes stable. The heuristic algorithm has a better performance because of two reasons. When m is small, the algorithm only needs to go through a few big branches to get the result set, and when m is big, the algorithm doesn't need to go over other branches if the most potential branch has zero result for the query. The experimental result shows that the heuristic is effective in pruning search space.

7 Related Work

Existing models capture either messages or activities but not both. Mealy service model [3, 9], BPEL [19], and WSCL [24] are message-based models, while the Roman model [1], PSL [21], CTR-S [6], and OWL-S [22] are mainly event or activity based. In this sense, our hybrid automata model is more "expressive" in terms of exporting service behaviors with both messages and activities.

Most work done in service discovery concentrates on the matchmaking of service interfaces [13, 17, 4, 8]. This includes the lexical match and semantical match of input messages, service operations, and service categories. Service behaviors are not considered in these discovery approaches. Reference [16] attempts to find wanted business processes by directly matching message sequences, lack of a way to describe temporal properties. Activity sequences are considered in [2], and service discovery relies on pattern matching of two graphes which represent services. Our query language takes both message sequences and activity sequences into account, and tries to capture temporal properties of these sequences.

RE-tree is developed in [5] for finding languages that contain a given word; in our work, a search query is another automaton. This makes the search more complicated. In [**?**], cycles are removed from FSA representation of services, which transforms an infinite set of message sequences into a finite one. Although B+ tree can be used, the approach is rather limited.

8 Conclusions

We present a new behavior model for web services, in which messages are associated with activities in a web service, and preconditions and results for activities are concerned to facilitate service discovery at the semantic level. A new query language is proposed, which captures properties on web service behavior signatures exported by its behavior model. Evaluation algorithms for the query language are developed, and index structure based on RE-tree is used to improve search performance, along with additional heuristics.

References

[1] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini and M. Mecella. Automatic Composition of e-Services that Export their Behavior. ICSOC, 2003.

[2] A. Bernstein, and M. Klein. Discovering Services: Towards High Precision Service Retrieval. CaiSE workshop on Web Services, e-Business, and the Semantic Web, 2002.

[3] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. WWW, May 2003.

[4] J Cardoso, and A. Sheth. Semantic e-workflow composition. Journal of Intelligent Info. Systems, vol.21, no.3, Nov. 2003.

[5] C.Y. Chan, M. Garofalakis, and R. Rastogi. RE-Tree: An Efficient Index Structure for Regular Expressions. VLDB, 2002.

[6] H. Davulcu, M. Kifer, I.V.Ramakrishnan. CTR-S:A Logic for Specifying Contracts in Semantic Web Services. WWW, 2004.

[7] A. Deutsch, L. Sui and V. Vianu. Specification and Verification of Data-driven Web Services. PODS 2004.

[8] X. Dong, A.Y. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity Search for Web Services. VLDB, 2004.

[9] X. Fu, T. Bultan, and J. Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. CIAA, 2003.

[10] A. Guttman. R-Trees:a dynamic index structure for spatial searching. In Proc. of SIGMOD, Boston, June 1984.

[11] R. Hull, and J. Su. Tools for Design of Composite Web Services. ACM SIGMOD, June 2004.

[12] B. Mahleko, and A. Wombacher. A grammar-based index for matching business processes. To appear in ICWS'05.

[13] A. Patil, S. Oundhakar, A. Sheth, K. Verma. Meteor-s Web Service Annotation Framework. WWW, 2004.

[14] J.Pearson, and P.Jeavons. A Survey of tractable constraint satisfaction problems. Technical Report CSD-TR-97-15, Oxford University, Computing Laboratory, 1997.

[15] R. Reiter. Knowledge in Action. Sep. 2001, MIT Press.

[16] A. Wombacher , P. Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for business processes. IEEE Int. Conference on E-Commerce, 24-27 June 2003.

[17] L. Zhang, B. Li, T. Chao, and H. Chang. On demand Web Services-based Business Process Composition. IEEE Int. Conf. on Systems, Man and Cybernetics, Vol: 4, 2003.

[18] W3C. WSDL1.1. March 2002. http://www.w3.org/TR/wsdl.

[19] BPEL4WS Version 1.1. May 2003. http://www-106.ibm.com/developerworks/library/ws-bpel/.

[20] W3C. WSCDL Version 1.0. April 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.

[21] Process Specification Language. http://www.mel.nist.gov/psl/ontology.html.

[22] The OWL Coalition. OWL-S:Semantic Markup for Web Services. http://www.daml.org/services/owl-s/1.1B/owl-s.pdf.

[23] OASIS. UDDI v3.0. 19 July 2002. http://uddi.org/pubs/uddi-v3.0.1-20031014.htm.

[24] W3C. WSCL v1.0. 14 March 2002. http://www.w3.org/TR/2002/NOTE-wscl10-20020314/.

[25] WSMO. Web Service Modeling Ontology. http://www.wsmo.org/index.html.