A Grammar-Based Index for Matching Business Processes

Bendick Mahleko[1] Andreas Wombacher[2*]
Peter Fankhauser[1]
[1]Fraunhofer Institute, Integrated Publication and Information Systems (IPSI)
D-64293 Darmstadt, Germany, {mahleko|fankhaus}@ipsi.fraunhofer.de
[2]Center for Telematics and Information Technology, University of Twente
500 AE Enschede, The Netherlands, a.wombacher@utwente.nl

Abstract

Complex services are composed of simple services which typically need to be processed in a particular order. Two complex services only match if they agree on both, their simple services and their processing order. This matching semantics can be formalized by means of modelling complex services as finite state automata (FSAs), and analysing the intersection of the FSAs. However, computing the intersection of FSAs is computationally expensive, and thus does not scale for large service repositories. This paper presents an approach for indexing and matching complex services using an abstraction that transforms the underlying FSA via its grammar into a form that can be indexed using available index mechanisms. Evaluation of this approach shows a performance gain of several orders of magnitude as compared to sequential matching.

1. Introduction

The current web service discovery technology is based on UDDI and WSIL which offer limited query support. In particular, query evaluation is limited to attribute/value queries with attributes such as business name, service name, key-ids, category-name, etc., being used to search for services and their providers. More complex descriptions of services such as process aspects, QoS aspects, semantics etc., are not supported. Research by various groups is ongoing to address these limitations and different extensions have already been proposed e.g., [11]. This paper focuses on the process aspect; in particular we address the question, how to efficiently search for services within a service discovery infrastructure to find service providers supporting the business process of a service requestor.

*This work was done during employment at Fraunhofer IPSI.

Business processes in Web Services can be specified using BPEL, where a subset of BPEL can be used to derive a formal model representing a process as a set of message sequences each representing a potential execution sequence of the process [14]. The number of supported message sequences as well as the length of a message sequence may be infinite due to cycles in the process specification. Based on this model matchmaking can be defined as an intersection of message sequences. Since a set of message sequences is potentially infinite, it can not be handled directly by traditional database systems.

This paper presents an indexing approach for querying cyclic business processes using traditional database systems. In particular, we introduce an abstraction function that removes cycles and transforms a potentially infinite set of message sequences into a finite representation, which can be handled by existing database systems. Introducing an abstraction implies an information loss, which may result in not finding all relevant business processes (false misses) and/or returning also processes not being relevant (false matches). The abstraction introduced in this paper guarantees that no false misses occur although false matches are possible. The goal is to minimize the false-hit ratio through the analysis of input data and choosing appropriate input parameters to the abstraction function. Evaluation of this approach shows that a performance gain of significant orders of magnitude is achievable compared to sequential searching.

The rest of the paper is structured as follows: Section 2 discusses the indexing approach, Section 3 discusses the complexity and Section 4 describes the evaluation. Section 5 discusses related work and the conclusion and future work are presented in Section 6.

2. Approach

In this section a formal model for matching business processes is presented. Further, a grammar-based abstrac-

tion of cyclic process specifications to finite message sequences is introduced, which is the basis for indexing the business processes.

2.1. Matching business processes

We assume that business processes are described using the BPEL language specification [1]. Further, we assume that the involved partners are using a common data dictionary, that is the same task descriptions have the same semantics. This is a reasonable assumption because standard bodies already exist to define message formats and their semantics, e.g., RosettaNet, Open Travel Alliance and IOTP. Thus, no semantic reasoning is required when matching business processes, as they derive from the same data dictionary.

To match BPEL specifications, we need a formal model. To this end we have investigated various alternatives and settled on finite state automata (FSA) [8] for a start [1]. The reason for this choice is that it is the easiest model to represent potentially infinite sets of message sequences.

Two FSAs match if their intersection is not empty, i.e., if there exists at least one common path (message sequence) between the start and final states. The definition for matchmaking in this case is non-empty intersection of finite state automata. The language of an FSA represents the set of message sequences accepted by a business process. An FSA state represents the state in which a business process is in e.g., waiting for acknowledgment, and a transition represents a change from one business state to another. Final states in a finite state automaton mark the completion of a message sequence accepted in a business process.

2.2. Indexing FSAs

The basic principle behind our approach is to reduce the complexity of the potentially infinite set of message sequences accepted by an FSA, by making the set finite. If the set is finite, the intersection operator can be replaced by simple string equivalence of finite message sequences encoded as a string, which is readily supported by traditional index structures.

We make the language of an FSA finite by removing cycles in the FSA specification such that the corresponding language is finite. Due to the introduced information loss false misses and false matches may occur. Hence, our indexing approach has the following optimization goals: (i) excluding false misses (ii) minimizing the processing time and (iii) minimizing the number of false matches.

[1] Alternative workflow models have been discussed in [13].

2.2.1 Formal specification

A query performed by a matchmaking engine is evaluated by computing the intersection of the query FSA against FSAs from the data collection; the intersection result is checked for non-emptiness. The final result is the set of FSAs from the data set, that have a non-empty intersection with the query FSA. The definition of an FSA is as described in [8].

Definition 1 (Finite state automata)
An FSA A is represented as a tuple $A = (Q, \Sigma, \Delta, q_0, F)$ where
– Q is a finite set of states,
– Σ is a finite set of messages,
– $\Delta : Q \times \Sigma \times Q$ represents labeled transitions,
– q_0 a start state with $q_0 \in Q$, and
– F a set of final or accepting states with $F \subseteq Q$.

In this paper, we differentiate between three types of FSAs depending on their structure. They are (i) FSAs with no cycles (ii) FSAs with simple cycles and (iii) FSAs with complex cycles. An FSA with no cycles has a finite language. Such FSAs are easy to process and to index because their languages are easily enumerated and indexed using standard techniques for string equivalence. The cycles of an FSA graph are either simple or complex [12]. We begin by formally describing cyclic FSAs.

Definition 2 (Cyclic FSAs)
Let $A = (Q, \Sigma, \Delta, q_0, F)$ be an FSA. A is cyclic if there exists at least one cycle $c =< t_1, \cdots, t_n >\in \Delta^n$ with $t_i = (q_i, a_i, q_{i+1})$ for all $1 \leq i \leq n-1$ and $t_n = (q_n, a_n, q_1)$. Let $\mathcal{C}(A)$ be the set of all cycles contained in automaton A.

The above definition describes an ordered set of transitions such that the first and last transitions in the list share a common source and target state respectively. In the next definition we describe simple cycles.

Definition 3 (Simple cycles)
Let $A = (Q, \Sigma, \Delta, q_0, F)$ be an FSA. A contains only simple cycles if all its cycles have disjoint states. Formally, A contains only simple cycles if for all $c_\ell, c_r \in \mathcal{C}(A)$ with $c_\ell =< t_{\ell,1}, \cdots, t_{\ell,n_\ell} >\in \Delta^{n_\ell}$ and $c_r =< t_{r,1}, \cdots, t_{r,n_r} >\in \Delta^{n_r}$ the following holds

$$\nexists \quad i \in \{1, \cdots, n_\ell\}, j \in \{1, \cdots, n_r\}.q_{\ell,i} = q_{r,j} \wedge t_{\ell,i} = (q_{\ell,i}, a_{\ell,i}, q_{\ell,i+1}) \wedge t_{r,j} = (q_{r,j}, a_{r,j}, q_{r,j+1})$$

Every cycle for which there exists another which share the same state is called a complex cycle.

We have already pointed out that FSAs with no cycles are easy to index because we can enumerate their words. Thus, they exhibit the least complexity in terms of number of words. The languages of FSAs with simple or complex cycles are both infinite, however, the languages of FSAs with

complex cycles usually contain a larger number of words up to a certain length than languages associated to FSAs with simple cycles due to the combination of the different complex cycles. In this paper we focus on cyclic FSAs. We shall also show that our indexing approach is ideal for FSAs with simple cycles as the number of words in FSAs with complex cycles quickly explodes.

2.2.2 Abstraction

The reduction of a cyclic FSA to a finite language is achieved by introducing an abstraction function ϕ which takes as input, an FSA A (potentially cyclic) and outputs a finite set of words that can be used for indexing using standard techniques. The index is defined in terms of the abstraction function ϕ as a set of tuples $(A,\ \phi(A))$ and can be implemented using standard data structures e.g., B^+-trees.

Below we describe four possible abstractions to make an infinite FSA language finite, thus indexable with standard approaches. In all the approaches, the same abstractions are applied to both the query and the data.

Ignore message order
A possible abstraction is to ignore order in message sequences and rely on the alphabet of the FSA i.e., individual messages for searching. Based on this approach, two business processes match if the set of messages they support have a non-empty intersection. An index can be constructed from message sets by using standard set indexing techniques such as those described in [7]. The problem

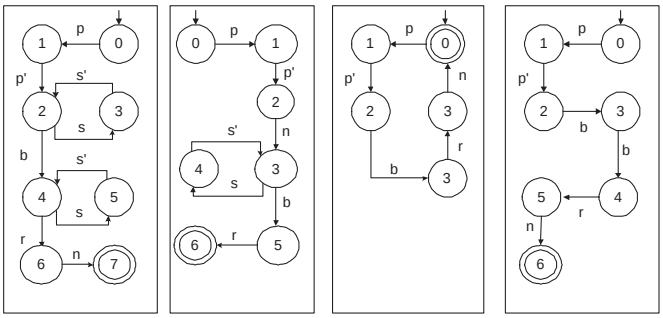


Figure 1. (a) $L(1)$ Seller (b) $L(2)$ Buyer $\#1$ (c) $L(3)$ Buyer $\#2$ (d) $L(4)$ Buyer $\#3$ processes

with this approach is that it gives rise to a high rate of false matches because the order of messages is totally ignored. This is confirmed by Helmer who showed that searching data that is set-indexed using an intersection operator is very inefficient [7].

Example Figure 1 is a simplified, but illustrative example showing a seller (a) and three buyers $(b) - (d)$, represented by their business processes which are modeled as FSAs. The seller and buyers are all using RosettaNet PIPs as basic building blocks for their processes. FSA states represent business states. A state with an incoming transition with no source state is a start state e.g., state 0 is a start state. States with concentric circles represent final states. Arcs connecting two states are called transitions, where a label annotated at an arc represents a message. A path from the start state to the final state represents a sequence of messages accepted by a business process.

The business process of a seller can be explained as follows: the seller expects to receive a purchase order request message p (PIP3A4) from a buyer and he responds with a purchase order confirmation p' message (PIP3A4). Having confirmed the purchase order, the seller can get purchase order status query s (PIP3A5) to which he must respond with a purchase order status response message s' (PIP3A5). This query can be sent several times or not at all. Having confirmed the purchase order, the seller expects to send a billing statement notification message b (PIP3C5) to the buyer. Again, in this state, the buyer may enquire about the status of his purchase order using s message (PIP3A5). The seller expects that after receiving the billing notification b (PIP3C5), the buyer will send a remittance advice notification message r (PIP3C6) showing his payment plan for the items ordered. Finally, the seller sends a notify of advance shipment message n (PIP3B2) informing the buyer about shipment.

The seller $L(1)$ Figure 1 (a) and buyer$\#1$ $L(2)$ Figure 1 (b) business processes have messages in common, but ordered differently. The difference is that the buyer expects to get shipment information n (PIP3B2) soon after his purchse order is confirmed by the seller via p' message (PIP3A4) and do remittance via r message (PIP3C6) later, while the seller process is sending shipping information via n message (PIP3B2) only after remittance advice r message (PIP3C6) is received. Thus from a message level, the two processes match; however, when order is taken into account, the two business processes do not match. Thus ignoring message order and relying on message set intersection results in a false match in this case.

Prune away cycles
Another possible abstraction is to use the FSA, but ignoring cycles. Unlike the previous approach which ignores message order, in this approach the order of messages in not ignored. While this approach is simple, it violates one of the stated requirements for the indexing approach. It can easily be verified that pruning away cycles from both the query and stored FSA results in false misses, hence is not suitable for our purpose.

Example The seller and buyer#2 processes (Figure 1(a) and (c) respectively) match, since they have message sequences in common. However, when the cycles are pruned away, the two business processes will not match. This is because the originally matching message sequences $pp'brn$ will no longer match when the back edge $(3, n, 0)$ is pruned away from $L(2)$, resulting in a false miss.

Remove duplicates in message sequences

A third possible abstraction is to remove duplicate messages in every sequence as follows: (i) keep the first occurrence of a message in a message sequence (ii) ignore all subsequent occurrences of the same message in the same sequence.

The result of this abstraction on the original language is that a new finite language where no false misses can occur during searches, but false matches are possible. False misses are not possible because intuitively, any sequences that match originally, will still match if the same amount of information is taken away from both automata.

Example The business process for buyer#3 i.e., allows upto two billing statement notification messages b (PIP3C5) to be received. The seller and buyer#3 processes do not match, because in buyer#3's business process, the purchase order confirmation message p' (PIP3A4) is followed by two consecutive billing statement notification messages b (PIP3C5). However, after duplicates are removed in the buyer#3 business process, the extra b (PIP3C5) message is removed and the two business processes will match, thus a false match has occured.

Remove duplicates and record context information through a look-back

This abstraction builds upon the previous one, where duplicate messages are removed. The goal is to reduce the number of false matches by reducing the ambiguity of transitions in different FSAs. One well known approach of resolving ambiguities in languages is to use look-ahead. Alternatively, we are using a look-back, i.e., considering history information of a path traversal to make the transition information more unique. Thus, the context information tells us how we arrive at a given transition by looking backwards a predefined number of transitions in the history.

Like the approach to remove duplicates, this approach allows a finite language to be generated from an infinite one and allows no false misses to occur. However, the approach reduces the rate of false matches by introducing more precision into the language specification through the encoding of context information into the message sequence description. By increasing the amount of context information, we also increase precision, thus increasing the quality of search results.

Example The false match that resulted when duplicates were removed in Figure 1 (d) can be avoided by using context information with a look-back of 1. In particular, the original message sequence $pp'bbrn$ of the buyer#3 business process as depicted in Figure 1 (d) is examined. Rather than simply considering the message itself, we now also consider the previous message. Since the first message p does not have a previous message, the \$ sign is introduced as a placeholder indicating the start of a message sequence, thus, the first element of the sequence is $\$p$. The second message p' has a previous message p, thus, the second element of the sequence is pp'. Following this approach, the sequence above results in $\$p, pp', p'b, bb, br, rn$. However, the corresponding message sequence of the seller as depicted in Figure 1 (a) is $pp'brn$ which maps to $\$p, pp', p'b, br, rn$. The two sequences are different, thus they represent different message sequences, and the two FSAs do not match. Thus, the false match is eliminated by using a look-back of 1.

The rest of the discussion in this section focuses on explaining this approach in more detail, explaining the used algorithms. The approach is based on the construction of a grammar. The regular grammar of an FSA can be constructed using standard techniques e.g., [8]. The regular grammar is converted to a grammar with context information as described below for a look-back of 0:

Definition 4 Introduce context information, where look-back= 0.
Let $A = (Q, \Sigma, \Delta, q_0, F)$ be an FSA. A new grammar, $G_0 = (N_0, T_0, P_0, S_0)$, with context information where look-back amount = 0 is derived from A such that:

- *non-terminals:* $N_0 := \{[q] | q \in Q\} \cup \{[s], [s']\}$, *where s and s' are newly introduced start symbols with $s, s' \notin Q$*
- *terminals:* $T_0 := \Sigma \cup \{\$, \#, ', ' \mid \$, \#, ', ' \notin \Sigma\}$ *, where \$ and # represents the start and termination of a message sequence respectively*
- *start symbol:* $S_0 := [s]$
- *production rules:*

$$P_0 := \bigcup_{([q_1],a,[q_2])\in\Delta} \begin{cases} \{\epsilon[q_1] ::= a, \epsilon[q_2]; \\ \epsilon[q_2] ::= \#\} & \text{if } q_2 \in F \\ \{\epsilon[q_1] ::= a, \epsilon[q_2]\} & \text{else} \end{cases}$$

$$\bigcup \quad \left\{\epsilon[s] \quad ::= \epsilon, \epsilon[s']\right\}$$

$$\bigcup \quad \left\{\epsilon[s'] \quad ::= \$, \epsilon[q_0]\right\}$$

$$\bigcup \quad \left\{\epsilon[q_0] \quad ::= \#|q_o \in F\right\}$$

ϵ represents an empty string. The production $\epsilon[q_1] ::= a, \epsilon[q_2]$ can be expressed in short hand notation $\epsilon[q_1] \xrightarrow{a} \epsilon[q_2]$.

The set N_0 contains non-terminals, and is a union of FSA states where each state is put in square brackets, and a set consisting of two additional symbols [s] and [s']. The symbols [s] and [s'] are symbols for special start productions. T_0 is a set of terminals; it is derived from the union of all input messages to the FSA and a set of special symbols \$, # and $'$,$'$where \$ marks the start of a word, or message sequence, # marks the end of a word or message sequence in the resulting language and $'$,$'$ is a separator symbol. S represents the grammar's start symbol, which is $[s]$, in our representation.

P_0 is the set of productions or rules. Each production is represented in Backus Naur form $\epsilon[q_1] ::= a, \epsilon[q_2]$ for each transition $(q_1, a, q_2) \in \Delta$. ϵ represents an empty string. All productions are generated from transitions, plus three special productions, not directly derived from transitions. These productions are (i) $\epsilon[s] ::= \epsilon, \epsilon[s']$ (ii) $\epsilon[s'] ::= \$, \epsilon[q_0]$ and (iii) $\epsilon[q_0] ::= \#$. Productions (i) and (ii) allow a \$ symbol to be used at the beginning of every sequence and keeping the context information length constant even if the sequence length is shorter than the look-back. The rule $\epsilon[q_0] ::= \#$ is added only when the start state is a final state.

Productions in Definition 4 have the form $T^* \times N ::= a, T^* \times N$, which is a context sensitive grammar [3]. However the form of productions is effectively modelled such that the underlying grammar is regular, i.e., context prefix on the right-hand side always starts with the full context prefix of the left-hand side, and all right-hand side always generate terminals to the left of a non-terminal. We give an example to illustrate how the above definition is applied to the seller business process using a grammar with a look-back of 0:

$G_0 = (N_0, T_0, P_0, S_0)$ where
$N_0 := \{[s], [s'], [0], [1], [2], [3], [4], [5], [6], [7]\}$,
$T_0 := \{b, n, p, p', r, s, s', \$, \#, ', '\}$, $S_0 := [s]$, and

$$
\begin{aligned}
P_0 := \{ \quad & \epsilon[s] ::= \epsilon, \epsilon[s']; && \epsilon[s'] ::= \$, \epsilon[0]; \\
& \epsilon[0] ::= p, \epsilon[1]; && \epsilon[1] ::= p', \epsilon[2]; \\
& \epsilon[2] ::= s, \epsilon[3] \mid b, \epsilon[4]; && \epsilon[3] ::= s', \epsilon[2]; \\
& \epsilon[4] ::= s, \epsilon[5] \mid r, \epsilon[6]; && \epsilon[5] ::= s', \epsilon[4]; \\
& \epsilon[6] ::= n, \epsilon[7]; && \epsilon[7] ::= \# \qquad \}
\end{aligned}
$$

The finite language generated from the above grammar is shown below. For brevity we omit ϵ symbols from the language since they represent empty strings:

$\{[\$, p, p', b, r, n, \#], \quad [\$, p, p', s, s', b, r, n, \#],$
$[\$, p, p', b, s, s', r, n, \#] \}$.

Computing grammar for incremented look-back

We now describe a generic algorithm to increase the look-back from n to $n+1$.

Definition 5 ***(Computing context sensitive grammar)*** *Let $G_n = (N_n, T_n, P_n, S_n)$ be a grammar representing an FSA with a look-back of n. A successor grammar $G_{n+1} = (N_{n+1}, T_{n+1}, P_{n+1}, S_{n+1})$ with look-back $n+1$, is generated from G_n such that:*
– $N_{n+1} := N_n$
– $T_{n+1} := T_n$
– $S_{n+1} := S_n$
– let $\bar{a} := a_3 \cdots a_{n-1}$ and
– $q_0 \in T_n \wedge \$^n[s'] ::= \$, \$^n[q_0] \in P_n$
$P_{n+1} :=$

$$
\bigcup_{\substack{a_1 a_2 \bar{a}[q] ::= a', a_2 \bar{a} a'[q_1] \\ \in P_n \setminus \{\epsilon[s] ::= \epsilon, \$^n[s']\}}}
\begin{cases}
\{a_1 a_2 \bar{a} a'[q_1] ::= a'', a_2 \bar{a} a' a''[q_2]\} \\
\quad \textit{if } a_2 \bar{a} a'[q_1] ::= a'', \bar{a} a' a''[q_2] \in P_n \\
\\
\{\, a_1 a_2 \bar{a} a'[q_1] ::= \#\} \\
\quad \textit{if } a_2 \bar{a} a'[q_1] ::= \# \in P_n
\end{cases}
$$

$$
\bigcup \quad \left\{ \epsilon[s] \quad ::= \epsilon, \$^{n+1}[s'] \right\}
$$

$$
\bigcup \quad \left\{ \$^n[s'] \quad ::= \$, \$^{n+1}[q_0] \right\}
$$

where $a_i, a', a'' \in T_n$

The non-terminals, terminals and start symbol for a grammar with look-back $n+1$ are directly derived from corresponding variables in the previous grammar with look-back n, through equivalence.

We will use an example to explain how productions for the grammar with look-back of $n+1$ is generated. We will use productions P_0 given in the previous example to illustrate how productions P_1 for the grammar with look-back of 1 are generated.

The start production is $\epsilon[s] ::= \epsilon, \$[s']$ being derived from $\epsilon[s] ::= \epsilon, \$^1[s']$ since look-back is 1. The next production is $\$[s'] ::= \$, \$[0]$ which is derived from $\$^1[s'] ::= \$, \$^1[q_0]$ in the above definition.

The rest of the productions are derived by iterating over productions of P_0 (excluding $\epsilon[s] ::= \epsilon, \epsilon[s']$). First, we take $\epsilon[s'] ::= \$, \epsilon[0]$ as the pivot production. The set of productions reachable from $\epsilon[s'] ::= \$, [0]$ in a single step is $\{\epsilon[0] ::= p, \epsilon[1]\}$. Thus we can derive a new production $\$[0] ::= p, p[1]$ by combining the pivot production with $\epsilon[0] ::= p, \epsilon[1]$. The new context information for $[0]$ is \$ which is a concatenation of the context information of $[s']$ with that of \$ from $\epsilon[s'] ::= \$, \epsilon[0]$ in P_0. The context information of $[1]$ is similarly obtained from $\epsilon[0] ::= p, \epsilon[1]$ i.e., concatenate context information for $[0]$ and p to obtain p.

The next pivot production is $\epsilon[0] ::= p, \epsilon[1]$ and the set of reachable productions is $\{\epsilon[1] ::= p', \epsilon[2]\}$. By applying the same principle, the new production will be $p[1] ::= p', p'[2]$. The full list of productions computable from P_0 is given below:

$$
\begin{array}{lll}
P_1 := \{ & \epsilon[s] ::= \epsilon, \$[s']; & \$[s'] ::= \$, \$[0]; \\
& \$[0] ::= p, p[1]; & p[1] ::= p', p'[2]; \\
& p'[2] ::= s, s[3] \mid b, b[4]; & s[3] ::= s', s'[2]; \\
& b[4] ::= s, s[5] \mid r, r[6]; & s'[2] ::= s, s[3] \mid b, b[4]; \\
& s[5] ::= s', s'[4]; & r[6] ::= n, n[7]; \\
& s'[4] ::= s, s[5] \mid r, r[6]; & n[7] ::= \# \quad \}
\end{array}
$$

Figure 2 illustrates the graphical representation for the grammar corresponding to the above productions.

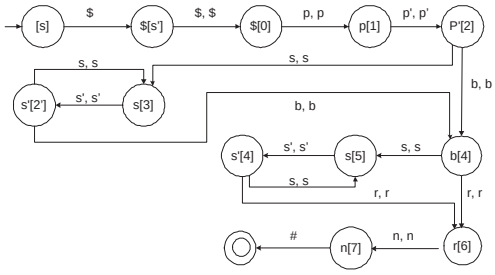


Figure 2. Seller example with look-back of one

In this figure, non-terminals preceded by their context information are represented as states; transitions are labeled with terminals in accordance to the above production rules, i.e., transition labels have the form $a, a_1 \cdots a_{n-1}a$ where a is the label for the current transition, $a_1 \cdots a_{n-1}a$ is context information for the target state reached by this transition. All transitions having $\#$ as a transition label lead into a final state.

We have presented the picture in Figure 2 for illustration purposes only, otherwise it is not necessary for computing the finite language. A finite language is computed from the grammar by deriving words that are generated from the start symbol, such that every cycle is traversed at most twice. During the traversal, duplicates are used as fixpoint. If a subsequence has already been added to the current sequence, the same subsequence is not added again to the sequence if encountered. The following is the finite set of words derived from the productions of P_1:
$[\$\$, \$p, pp', p'b, br, rn, n\#]$,
$[\$\$, \$p, pp', p's, ss', s'b, br, rn, n\#]$,
$[\$\$, \$p, pp', p'b, bs, ss', s'r, rn, n\#]$,
$[\$\$, \$p, pp', p's, ss', s'b, bs, s'r, rn, n\#]$.
Increasing the amount of look-back results in more context information being encoded in subsequences making-up words of the language. This results in a more expressive finite language because potential false matches are minimized as we have already shown. The quality of search results is also increased as a result of the high expressiveness of the languages being intersected. We also want to explicitly point out that both the query and data must undergo the same transformation, hence, the approach is guaranteed to find a match if one exists.

The language resulting from the removal of duplicates and recording of context information is finite as illustrated above. This means we can enumerate words of the language, and use standard indexing techniques such as a B^+-tree to index such languages.

3. Complexity Analysis

Converting an FSA to a grammar with look-back of zero is performed by traversing all transitions of the FSA graph from the start state, visiting each transition only once in a depth-first fashion. The computational complexity is $O(|\Delta|)$ and storage complexity is $O(|\Delta| + |Q|)$ where $|\Delta|$ is the number of transitions and $|Q|$ is the number of states.

Number of productions in an acyclic FSA The best case for the number of words to be stored is when an FSA is non-cyclic. The number of productions $R(n)$, generated for non-cyclic FSAs is given by

$$R(n) \quad = \quad 2 + |\Delta| + |F| \tag{1}$$

where $|\Delta|$ is the number of transitions and $|F|$ is the number of final states. The rational for the above formula is as follows: every FSA transition (q_1, a, q_2) results in a new production $a_1 \cdots a_{n-1}[q_1] \xrightarrow{a} a_2 \cdots a_{n-1}a[q_2]$ where n is the look-back amount, and every FSA transition leading to a final state results in an extra production $a_1 \cdots a_{n-1}[q_2] \xrightarrow{\#} a_2 \cdots a_{n-1}\#$. In addition, there are 2 start productions of the form $\epsilon[s] \xrightarrow{\epsilon} \$^n[s']$ and $\$^n[s'] \xrightarrow{\$} \$^n[q_0]$ where q_0 is a start state.

Number of words in an acyclic FSA The number of generated finite words W cannot exceed the number of transitions $|\Delta|$ plus 1 in the best case, which is when the FSA is acyclic. Hence, the number of generated finite words W is given by

$$W \quad \leq \quad 1 + |\Delta| \tag{2}$$

where 1 represents a special empty word when the start state of an FSA is a final state. The computational and storage complexity for the number of words is therefore $O(|\Delta| + |Q|)$.

3.1. FSAs with simple cycles

The next definition presents a derivation for the number of productions in a simple FSA. Let $A = (Q, \Sigma, \Delta, q_0, F)$

be a simple FSA. For a given look-back amount n, the maximum number of productions is computed recursively as follows:

$$\begin{aligned} R(0) &= 2 + |\Delta| + |F| \quad (3) \\ R(1) &= R(0) * |\Delta| \\ R(2) &= R(1) * |\Delta| \\ &= (R(0) * |\Delta|) * |\Delta| \\ &= R(0) * |\Delta|^2 \\ &\vdots \\ R(n) &= R(0)|\Delta|^n \\ &= (2 + |\Delta| + |F|) * |\Delta|^n \quad (4) \\ &= O(|\Delta|^{n+1}) \quad (5) \end{aligned}$$

The rational for the above equation is as follows: the number of rules when look-back is zero cannot exceed the sum of the number of FSA transitions, final states plus 2. Thus the maximum number of productions $R(0)$ is $R(0) = 2 + |\Delta| + |F|$.

A new grammar for a given look-back n, is computed from the previous grammar (look-back amount $n-1$). This requires iterating over the grammar rules for the previous look-back $n-1$. In each iteration, every grammar rule can generate at most $|\Delta|$ new rules, hence the total number of new rules is the product of total number of previous rules and the number of transitions (assuming that each transition results in the introduction of a new rule). The termination condition is the call to $R(0)$, which is computed non-recursively. Thus the complexity for computing the number of rules is $O(|\Delta|^{n+1})$, being exponential on the look-back for the worst case.

The worst case complexity for the number of finite words is computed from the set of rules. Let $\overline{|\Delta|}$ be the average number of transitions in each FSA, given by

$$\overline{|\Delta|} = \frac{\sum_{i=1}^{N} |\Delta_i|}{N} \quad (6)$$

where $|\Delta_i|$ is the number of transitions in FSA B_i, each B_i is an FSA in the data collection and N is the number of FSAs in the collection. The worst case complexity for the number of productions is $O(\overline{|\Delta|}^{n+1})$. The number of finite words that can be generated from the grammar rules can be approximated as follows:

$$\begin{aligned} W &= 2 * ((2 + \overline{|\Delta|} + |F|) * \overline{|\Delta|}^n) \\ i.e.\ W &= O(\overline{|\Delta|}^{n+1}) \quad (7) \end{aligned}$$

Equation 7 is derived as follows: for an FSA with simple cycles, the worst case complexity is when every production derives a word. In addition, cyclic paths are traversed at most twice, thus we multiply the total number of rules (see Equation 4) by 2 to get an upper bound on the potential number of words that can be derived. Thus the result is, the number of words for the worst case is exponential on the average number of transitions in the FSA.

The effort needed to construct the index is a sum of the effort to compute the grammar, compute a finite set of words and insert into an index structure such as B$^+$-tree. The best case effort needed to construct the index is thus $O(N * \overline{|\Delta|} + log(N * \overline{|\Delta|}))$ where $\overline{|\Delta|}$ is the average number of transitions in an FSA and N is the number of FSAs to be inserted into the collection. $O(log(N * \overline{|\Delta|}))$ is the effort needed to insert into a B$^+$-tree index structure [6]. The worst case complexity is $O\Big(N * \overline{|\Delta|}^{n+1} + log(N * \overline{|\Delta|}^{n+1})\Big)$ where n is the amount of look-back. $\overline{|\Delta|}^{n+1}$ is the number of words generated for each FSA in the worst case as given by equation 7 above.

Index search for FSAs with simple cycles Index-based search is done in three phases: (i) transforming the query FSA into a finite set of words (ii) searching on a standard index structure such as B$^+$-tree (iii) merging intermediate results.

The complexity for transforming an FSA above is $O(|\Delta_q|)$ for the best case and $O(|\Delta_q|^{n+1})$ where $|\Delta_q|$ is the number of transitions in the query FSA (see equation 5).

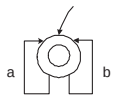
The best case complexity for searching is when the FSAs in the collection are non-cyclic. The search complexity is given by $O(|\Delta_q| * log(N * \overline{|\Delta|}))$ for the best case, where $O(|\Delta_q|)$ is best case complexity for the number of words in the query FSA. The worst case complexity is $O\Big(|\Delta_q|^{n+1} * log(N * \overline{|\Delta|}^{n+1})\Big)$. In general, the number of words from the query FSA is small compared to the number in the collection. Thus, we can omit the $|\Delta_q|$ and $|\Delta_q|^{n+1}$ in the respective complexities for the best and worst case. Thus search complexities can be approximated with $O(log(N * \overline{|\Delta|}))$ and $O\Big(log(N * \overline{|\Delta|}^{n+1})\Big)$ respectively for the best and worst cases.

The standard merge-sort algorithm is used for merging intermediate search results. The complexity for merge-sort is $O\Big(N' log(N')\Big)$ where $N' \leq N$, is the number of FSAs to merge.

The best total search complexity is computed by summing-up the best case complexities for transforming the query FSA, searching and merging which is
$O\Big(|\Delta_q| + |\Delta_q| * log(N * \overline{|\Delta|}) + N' log(N')\Big)$.
The worst case total search complexity is similarly obtained, being
$O\Big(|\Delta_q|^{n+1} + |\Delta_q|^{n+1} * log(N * \overline{|\Delta|}^{n+1}) + N' log(N')\Big)$.

3.2. FSAs with complex cycles

The number of words generated quickly explodes for FSAs with complex cycles. As an example, Figure 3 (b) shows how the number of words exponentially increases for a simple example FSA, Figure 3 (a) with two transitions only. In Figure 3 (b) $|Delta|$ represents the number



(a)

|Delta|	n	|P|	W
2	0	7	5
2	1	15	53
2	2	31	2 555

(b)

Figure 3. FSA with complex cycles

of transitions, n, the look-back amount, $|P|$ the number of productions and W the number of words. Clearly, a different approach is needed to index FSAs with complex cycles e.g., representing FSAs using message sets with context information to increase precision.

4. Evaluation in Web services Domain

The indexing approach described in this paper was implemented and integrated into the IPSI-PF engine - a service discovery engine extending the current UDDI infrastructure with process-aware service discovery capability [15]. There are two parts to the query (i) a UDDI part to support standard UDDI queries (ii) a business process matching engine to support the matching of business processes based on an index approach described in this paper. The business process part is described using standards such as BPEL. The two query parts are passed to the respective engines i.e., the UDDI part of the query is parsed to the UDDI engine such as jUDDI and the business process (BPEL) part is passed to another component that converts BPEL to FSA. We have implemented such a *BPEL to FSA* convertor [14]. After the conversion, the FSA is passed to the business process engine for computing the business process matching. Decomposition of the query is realized as a pipeline e.g., using cocoon, and merging of final results is done using a natural join that is realized using XSLT.

We carried-out experimental evaluations of the indexing approach. The goal of the evaluation was to characterize indexed search complexity, performance and quality of search results. In particular, we wanted to find out the following:
– index-based search versus sequential search
– data-set size versus index performance
– look-back versus index performance
– look-back versus quality of search results

Specification of data-set

The data-set used for the experiments is business process data based on the RosettaNet specification [10]. We used RosettaNet data to ensure a realistic data-set with a significant level of complexity. We developed a tool to generate complex business processes from the RosettaNet specification as well as analyzing and categorizing them. Using the tool, RosettaNet PIPs were composed into more complex sequences according to RosettaNet rules for combining PIPs. Parameters such as branching, cycle and self-cycle probabilities as well as business process graph depth were used to configure the tool for data generation. The data was partitioned into three sets of sizes 100, 200 and 300 business processes to allow the measuring of query performance-time as a function of data-set size. To measure the influence of look-back on performance, the look-back was varied from 0 through to 5 for each data-set and the resulting number of words recorded. This data also allowed us to measure the influence of look-back on search results quality (i.e., rate of false matches).

Table 1. FSA complexity

N	$\|Q\|$	$\|\Sigma\|$	$\|\Delta\|$
100	7 078	1 463	7 385
200	14 156	2 926	14 770
300	21 234	4 389	22 155

Table 2. data set/ look-back matrix

$N \backslash n$	0	1	2	3	4	5
100	3 152	4 122	4 122	4 726	4 783	5 504
200	6 304	8 244	8 244	9 452	9 566	11 008
300	9 456	12 366	12 366	14 178	14 349	16 512

The structural complexity of input business process data was also measured and recorded. The information recorded includes the number of states, messages, transitions, sequences and cycles. Table 1 summarizes the complexity of the input business processes and Table 2 shows the number of words recorded for each data set as the look-back n, is varied from 0 to 5. The used parameters in the two tables are: N is the data set size, $|Q|$ is the number of state objects, $|\Sigma|$ is the number of transition objects, and $|\Delta|$ is the number of transition objects. We can see from Table 2 that generally, increasing the look-back results in an increase in the number of words. This is because for cyclic FSAs, more iterations are made when the look-back is increased.

The data was used as input to our business process matching engine which was implemented using the approach described in this paper.

Environment of the experiments

The baseline infrastructure is sequential search-based matchmaking engine that scans the data set, computes intersection with the query FSA and checks for non-emptiness

of intersection results. An indexing engine is implemented at the same level of abstraction, based on the approach described in this paper.

The experiments were conducted on a Dell machine, with a Pentium 4 processor 2.00GHz clock speed and 512 MB RAM. The total disk space was 74 GB. The machine was running under Windows XP operating system. MySQL server version 4.0 was the used database engine. The machine was also running the JBOSS 3.2.3 application server which provided the J2EE environment for the IPSI-PF process matchmaking engine. The sequential and index search are implemented on the same data model and level of abstraction using container managed persistency. We allowed no buffering/caching of query results; all tests were run under cold start conditions.

Results

Influence of data set size and look-back The best case, average case and worst case performances were measured for both sequential and indexed-based searches for the three data sets. Index-based search was faster than sequential

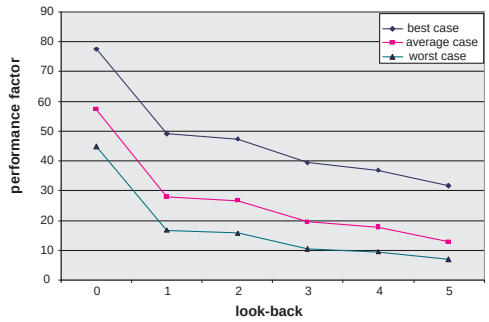


Figure 4. Index performance factor $N = 300$

searching by factors of 78, 58, and 45 for the best, average and worst case respectively, for $N = 300$ data set and $n = 0$ where n is the look-back. The performance factors gradually decrease as n was increased from 0 through to 5, and it was lowest at $n = 5$. Data sets with $N = 100$ and $N = 200$ showed similar patterns. The results of this experiment are shown in Figure 4 for $N = 300$. Figure 5 shows results $N = 100$ and $N = 300$. This behaviour can be explained as follows: increasing the look-back increases the number of words to be compared as shown in Table 2. This increases the overall complexity for both storage and computation, accounting for the decrease in performance gain as n is increased. Figure 5 also shows that index performance gain improves with increase in data set size. Thus our index performs even better with larger data sets.

False matches When $N = 300$ and $n = 0$ a false match rate of 0.26% was observed. When $n \geq 1$, the false match

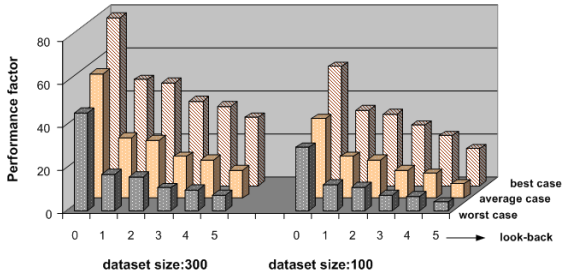


Figure 5. Index performance factor $N = 100$ **and** $N = 300$

rate drops to 0% meaning no false match was recorded. This result shows that the effect of false matches is not that significant. However a full analysis of false match occurence still needs to be carried out.

5. Related Work

The approach presented in this paper to index FSA is to use an grammar-based abstraction to reduce complexity of the finite state automata, hence making it "indexable". The only approach that allows full indexing of FSA to support FSA intersection queries to our knowledge is the RE-Tree [2]. It relies on finite state automata theory to build a hierarchical tree structure like the B+tree, where the language containment relationship between finite state automata is used to guide the search. The RE-tree approach requires several optimizations to be performed to construct the index, e.g., constructing an optimal bounding automaton, finding an optimal node split when a node is full (NP hard problem), and selecting an optimal insertion node (maximizes language intersection). The index has high complexity in terms of construction, as compared to our approach. Here we will compare only the search complexities because search operations are done more often on a database than maintenance and construction operations.

The best case complexity for FSA-based intersection searching using RE-trees is $O\left(|\Delta_q| * \overline{|\Delta|}' * log(N)\right)$ where $|\Delta_q|$ is the number of transitions in the query FSA, $\overline{|\Delta|}'$ is the average number of transitions in the bounding FSA and N is the number of FSA in the database. This complexity formula is obtained as follows: the search tree is pruned by intersecting the query FSA with the bounding FSA and takes quadratic time $O\left(|\Delta_q| * \overline{|\Delta|}'\right)$. The intersection operation must be performed for the entire height h of the tree where $h = log_k(\frac{N+1}{k})$, accounting for the $log(N)$ in the complexity formula where k is the maximum degree of the RE-tree. On the contrary, the best case complexity for

searching using our approach is $O\Big(|\Delta_q| + |\Delta_q| * log(N * \overline{|\Delta|}) + N^{'} log(N^{'})\Big)$. The RE-tree best case search complexity is more than quadratic due to the multiplying factor of the number of transitions in query FSA, the average number of transitions in the bounding FSA and the logarithm of the number of FSA in the collection. On the other hand, our best case search complexity is more than linear, but less than quadratic as can be verified from the formula. Thus our approach has better best case complexity than Re-trees.

The worst case complexity for searching using an RE-tree is when every branch of the RE-tree is traversed. The complexity is given by $O\Big(|\Delta_q| * \overline{|\Delta|}^{'} * N * log(N) + N^{'} * log(N^{'})\Big)$. The number of intersection operations to be performed is given by $\sum_{i=1}^{h} k^i$ which has $O(N)$ complexity after substituting h with $log_k(\frac{N+1}{k})$ and expanding the summation formula. Thus the $O(N)$ complexity accounts for the N multiplying factor in the worst case complexity formula and $N^{'} * log(N^{'})$ is the complexity for doing merge-sort. In our approach, the worst case complexity for searching can be verified to be $O\Big(|\Delta_q|^{n+1} + |\Delta_q|^{n+1} * log(N) + |\Delta_q|^{n+1} * (n+1) * log(\overline{|\Delta|}) + N^{'} * log(N^{'})\Big)$. It can be shown that for small values of n and large data sets, our index has better worst case search performance than the RE-tree.

There exists other approaches like dataGuides [5], 1/2/T-indexes [9] and GraphGrep [4]. These approaches do not have mechanisms to handle infinite languages as well as the intersection operator on FSA. They are all based on clustering of paths, to reduce the number of paths to be searched.

Set-based approaches also exist, although they have not been applied to this kind of problem [7]. Their main limitation is that order is ignored in sets, hence the notion of a word is lost. Moreover, performance analysis on set-based structures done by Helmer [7], indicated that they support point queries very well, but are very poor when it comes to intersection queries. The set data structure itself is also lossy, hence, introduces lots of false matches.

6. Conclusion and Future Work

The paper presented an approach for indexing business processes for efficient matching in web service infrastructures. The approach uses a grammar-based abstraction to reduce the number of message sequences to be compared. The paper presented a description of the approach along with complexity analysis and experimental evaluation. Evaluation results show search performance improvement by an order of magnitude against sequential searching for the best case, average case and worst case. Analytical results also show that our approach is better than the RE-tree approach for FSA with simple cycles. Future work addresses the indexing of FSA with complex cycles. We also want to extend the approach to more complexity types of FSA e.g., annotated FSA, considering optional and mandatory messages within a business process specification. A full analysis of false matches will also be carried out.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for web services. version 1.1. http://www.ibm.com/developerworks/library/ws-bpel, May 2003.

[2] C. Chan, M. Garofalakis, and R. Rastogi. Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12(2):102–119, 2003.

[3] S. Ginsburg. *Algebraic and automata-theoretic properties of formal languages*. North-Holland/ American elsevier, 1975.

[4] R. Giugno and D. Shasha. Graphgrep: A fast and universal method for querying graphs. In *16th International Conference in Pattern recognition (ICPR)*, Quebec, Canada, August 2002. IEEE Computer Society.

[5] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[7] S. Helmer and G. Moerkotte. A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal*, 12(3):244–261, 2003.

[8] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001.

[9] T. Milo and D. Suciu. Index structures for path expressions. *Lecture Notes in Computer Science*, 1540:277–295, 1999.

[10] RosettaNet. http://www.rosettanet.org. last visited: 15 November 2004.

[11] A. ShaiklAli, O. F. Rana, R. Al-Ali, and D. W. Walker. Uddie: An extended registry for web services. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops (SAINT-w03)*. IEEE Computer Society, 2003.

[12] H. WEINBLATT. A new search algorithm for finding the simple cycles of a finite directed graph. *Journal of the Association for Computing Machinery*, 19(1), 1972.

[13] A. Wombacher, P. Fankhauser, B. Mahleko, and E. Neuhold. Matchmaking for business processes based on choreographies. *International Journal of Web Services*, 1(4):14–32, 2004.

[14] A. Wombacher, P. Fankhauser, and E. Neuhold. Transforming bpel into annotated deterministic finite state automata for service discovery. In *IEEE International Conference on Web Services (ICWS 2004)*, Los Alamitos, California, 2004. IEEE Computer society.

[15] A. Wombacher, B. Mahleko, and E. Neuhold. Ipsi-pf: A business process matchmaking engine. In *Proceedings of the CEC 2004, San Diego, California, USA*, 2004.