

CS-2006-28

**Deterministic Simulation of NFA  
with  $k$ -symbol Lookahead**

*Bala Ravikumar and Nicolae Santean*

*Technical Report 28*

David R. Cheriton School of Computer Science  
University of Waterloo

2006

# Deterministic Simulation of NFA with $k$ -symbol Lookahead

Bala Ravikumar<sup>1</sup> and Nicolae Santean<sup>2</sup>

Department of Computer Science<sup>1</sup>  
Sonoma State University, Rohnert Park, CA 94928  
School of Computer Science<sup>2</sup>  
University of Waterloo  
Waterloo, Ontario, Canada N2L 3G1

**Abstract.** We investigate deterministically simulating (i.e., solving the membership problem for) nondeterministic finite automata (NFA), relying solely on the NFA's resources (states and transitions). Unlike the standard NFA simulation, involving an algorithm which stores at each step all the states reached nondeterministically while reading the input, we consider deterministic finite automata (DFA) with lookahead, which choose the “right” NFA transitions based on a fixed number of input symbols read ahead. This concept, known as *lookahead delegation*, arose in a formal study of web services composition and its subsequent practical applications. Here we answer several related questions, such as “*when is lookahead delegation possible?*” and “*how hard is it to find a delegator with a given lookahead buffer size?*”. In particular, we show that only finite languages have the property that all of their NFA's have delegators. This implies, among others, that delegation is a machine property, rather than a language property. We also prove that the existence of lookahead delegators for unambiguous NFA is decidable, thus partially solving an open problem. Finally, we show that finding delegators (even for a given buffer size) is hard in general, and is efficient for unambiguous NFA, and we give an algorithm and a compact characterization for NFA delegation in general.

## 1 Introduction

Finite automata models are ubiquitous in a wide range of applications. The well-known classical applications of automata involve parsing, string matching and sequential circuits. Recently, formal models based on finite automata have been applied in service-oriented computing, a newly emerging framework to harness the power of the World Wide Web [Berardi et al., 2003]. One basic computational problem that arises in this framework is *automated service composition* [Gerede et al., 2004]. Informally, this problem can be described as follows: an activity automaton is a finite state acceptor that accepts a sequence of tasks (each represented by an input symbol). Automated composition involves breaking down a sequence of tasks and assigning them

to individual activity automaton. Formally, a system of finite automata  $\langle A; A_1, A_2, \dots, A_k \rangle$  is said to be composable if every string  $w$  accepted by the DFA  $A$  can be written as a shuffle product of strings  $w_1, \dots, w_j$  where each  $w_i$  is accepted by  $A_j$  for some  $j$ . This formal framework for e-services composition was introduced by [Berardi et al., 2003] and has recently been studied extensively by a number of authors: [Ibarra et al., 2006], [Hull and Su, 2004], [Mecella and Giacomo, 2004], [Gerede et al., 2004], [Gerede et al., 2005] etc.

A requirement more stringent than composability is the existence of  $k$ -lookahead delegators (or  $k$ -delegators for brevity) which is defined as follows. Given a system  $\langle A_1, A_2, \dots, A_k \rangle$  of DFA's or even NFA's, let  $A'$  be the "shuffle-product" of the system. Informally, a DFA  $A$  is said to be  $k$ -delegator for  $A'$  if the states of  $A$  are a subset of the states of  $A'$ . Further, based on the current state and the next  $k$ -input symbols, the transition table of  $A$  makes a deterministic choice among the possible choices of the NFA in such a way that if (and only if) the input string is accepted by  $A'$ , the simulation of  $A$  also results in an accepting state when the simulation is complete. For a given NFA, a basic question is whether it has a  $k$ -delegator for *some* integer  $k$ . One can also ask whether an NFA has a  $k$ -delegator for a *given*  $k$ .

$k$ -Delegators were first introduced informally in [Dang et al., 2004] in their study of e-services composability. In the same paper was established that the existence of  $k$ -delegators is decidable for a give  $k$ . However, the complexity of this problem was not addressed. Moreover, the problem of deciding the existence of a  $k$ -delegator for *some*  $k$  was left as an open problem. In this work, we address these and some related questions, without addressing the implications of our results in e-service applications.

The main results of this work can be summarized as follows. First, we define delegability as a property which can be viewed both as a language and machine property. When viewed as a language property, we characterize the family of regular languages whose all NFA have delegators. Since this family turns out to be that of finite languages, we adopt the second point of view, that of delegability as a machine property. We consider the complexity of determining if a given NFA has a  $k$ -delegator, and we formulate three versions of this problem. The first one involves a fixed  $k$ , the second one includes  $k$  (in unary) as part of the input and the third one involves determining if a  $k$ -delegator exists for some arbitrary  $k$ . When the input is restricted to an *unambiguous* NFA, the first problem is shown to have a polynomial time algorithm, the second one is shown to be in co-NP and the third one is shown to be in PSPACE. When the input may be an ambiguous NFA, even the first version is shown to be PSPACE-complete. We then provide an algorithm for Problem 2 in the general case, that is more efficient than the brute-force algorithm. This algorithm also leads to a simple necessary

and sufficient condition for the existence of a  $k$ -delegator, for some arbitrary  $k$ . Although the decidability of Problem 3 in the general case still remains open, our characterization provides a promising approach towards its resolution. We conclude with some open problems and directions for future work.

## 2 The Delegation Problem

In the following we assume known basic notions of automata theory (see, for example, [Hopcroft and Ullman, 1979] and [Yu, 1997]). Notation-wise, an NFA is a tuple  $M = (Q, \Sigma, \delta, q_0, F)$  with  $Q$  a finite set of states,  $\Sigma$  an alphabet,  $\delta \subseteq Q \times \Sigma \times Q$  a transition relation,  $q_0$  an initial state, and  $F \subseteq Q$  a set of final states.  $M$  is `trim` if each of its states is useful: `accessible` (there exists a computation from the initial state and ending with it) and `co-accessible` (there exists a computation starting from it and ending with some final state). If  $\delta$  is a function (as opposed to a relation), then  $M$  becomes a DFA (deterministic finite automaton). We say that two automata are equivalent if they recognize the same language. In the following we denote by  $\varepsilon$  the empty word, by  $\Sigma^k$  the set of all words over  $\Sigma$  of length  $k$ , by  $\text{pref}(L)$  the set of all prefixes of words in a language  $L$ , and by  $\text{pref}_k(L)$  the set  $\text{pref}(L) \cap \Sigma^k$ .

By a DFA with  $k$ -lookahead buffer we understand a DFA  $A = (Q, \Sigma, f, q_0, F)$  with  $f : Q \times \Sigma^k \rightarrow Q$ , which operates as follows.  $A$  has a buffer with  $k$  cells which initially contains the first  $k$  symbols of the input word (or, if the word has fewer symbols, the entire word). At each computation step,  $A$  consumes one input symbol and stores the following  $k$  symbols of the input tape in its buffer. The function  $f$  decides the next state based on the current state of  $A$  and its buffer content. It is easy to see that DFA with  $k$ -lookahead buffer are equivalent with standard DFA: one can view the buffer content as part of automaton's internal state.

We begin with the definition of a  $k$ -delegator, equivalent with, however different from, that provided in [Dang et al., 2004] – for the reason of improving the formalism.

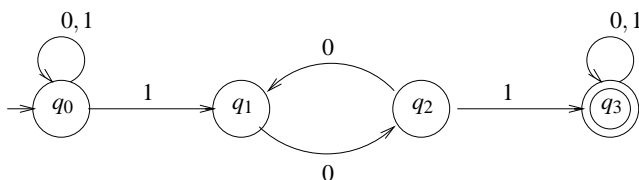
**Definition 1.** *An NFA  $M = (Q, \Sigma, \delta, q_0, F)$  has a  $k$ -delegator if there exists an equivalent DFA with  $k$ -lookahead buffer  $A = (Q, \Sigma, f, q_0, F)$  such that  $f(q, a_1 \dots a_k) \in \delta(q, a_1)$  for all  $(q, a_1 \dots a_k)$  in the domain of  $f$ .*

We say that  $A$  is a  $k$ -delegator for  $M$  or, when the context makes it clear, we denote  $f$  in the above definition to be a  $k$ -delegator for  $M$  (implying that there exists a DFA with  $k$ -lookahead as in the definition, with  $f$  its transition function). Indeed,  $M$  and  $A$  share the same resources (states and transitions) and the pair  $(M, f)$  uniquely identify the  $k$ -delegator  $A$  for  $M$ .

It is clear that any DFA  $M$  has a 1-delegator: simply choose  $f$  in the above definition as being the transition function of  $M$ . There are also NFA's that can have a 1-delegator. On the other hand, for any given  $k$ , it is not hard to construct an example of a NFA that has a  $k$ -delegator, but not a  $(k - 1)$ -delegator.

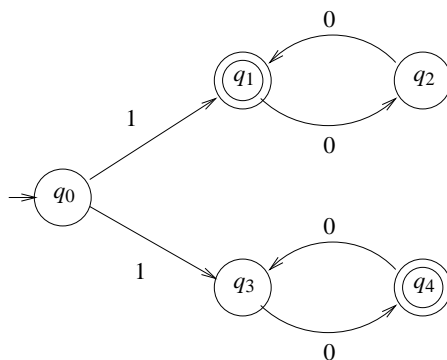
The following example shows that there are NFA's that do not have a  $k$ -delegator for any  $k$ .

*Example 1.* Consider the NFA  $M$  in Figure 1, for the language  $L$  of all words  $w \in \{0, 1\}^*$  in which some pair of successive occurrences of 1 has an odd number of 0's in between them. It is easy to see that  $M$  does not have a  $k$ -delegator for any positive integer  $k$ . The NFA in Figure 2 is an unambiguous NFA (i.e.,



**Fig. 1.** An NFA which has no  $k$ -delegator for any  $k$ .

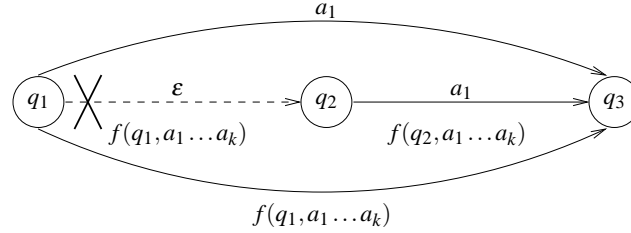
any word is the label of at most one successful computation), and yet, it has no  $k$ -delegator for any  $k$ .



**Fig. 2.** An unambiguous NFA which has no  $k$ -delegator for any  $k$ .

*Remark 1.* We restrict our study to trim  $\varepsilon$ -free NFA, since an  $\varepsilon$ -NFA has no  $k$ -lookahead delegators for some/any integer  $k$  if and only if its  $\varepsilon$ -free equivalent (considering the standard  $\varepsilon$ -elimination) has the same property.

*Proof.* It suffices to notice that a delegator  $f$  which follows an  $\varepsilon$ -transition does not “consume” the input, hence its buffer remains unchanged. Consequently, the  $\varepsilon$ -closure and  $\varepsilon$ -elimination (as shown in Figure 3) can be performed on  $f$  in order to obtain a delegator for the  $\varepsilon$ -free equivalent NFA.  $\square$



**Fig. 3.** The  $\varepsilon$ -removal for a  $k$ -delegator.

The basic idea in Definition 1 is that if a NFA  $M$  has a  $k$ -delegator  $A$  (or equivalently,  $f$ ), then given as input for the delegator the sequence of buffer content

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{pmatrix} \begin{pmatrix} x_2 \\ x_3 \\ \vdots \\ x_{k+1} \end{pmatrix} \cdots \begin{pmatrix} x_{n-k+1} \\ x_{n-k+2} \\ \vdots \\ x_n \end{pmatrix} \begin{pmatrix} x_{n-k+2} \\ x_{n-k+3} \\ \vdots \\ \# \end{pmatrix} \cdots \begin{pmatrix} \# \\ \# \\ \vdots \\ \# \end{pmatrix},$$

$A$  simulates  $M$  by entering a sequence of states of the NFA in such a way that if there is an accepting computation in  $M$  for the string  $x_1x_2\dots x_n$  (i.e., a sequence of states leading to an accepting state) then  $A$  goes through one such sequence of states leading to acceptance as well. Notice that  $A$  is not required to check that the input is in the “correct format”, that is, it does not check that each successive “super symbol”, consisting of a buffer of  $k$  symbols from the original alphabet, is obtained by subsequently dequeuing one symbol from the buffer and enqueueing a new symbol. Furthermore, notice that when the right-end of the input string is reached, a padding symbol  $\#$  is added to the buffer content in order to keep the buffer always filled (always containing  $k$  symbols). We will show later that padding the input is just a matter of formalism, and will be ignored most of the time.

*Remark 2.* Notice that “delegation” can be used as a measure of NFA nondeterminism. If NFA  $M$  has an  $m$ -delegator and  $M'$  has an  $n$ -delegator, with  $m, n$  being the smallest such integers, and if  $n > m$  then the cost of deterministically simulating  $M'$  is greater than that for  $M$ . In this sense,  $M'$  is “more nondeterministic” than  $M$ .

It is clear that every regular language  $L$  is accepted by an NFA that has a 1-delegator, namely a DFA for  $L$ . On the other hand, it may be the case that for some regular languages, every associated NFA may have a  $k$ -delegator for some  $k$ . The next definition is intended to characterize such regular languages.

**Definition 2.** *Let  $L$  be a regular language.*

- (i)  *$L$  is said to be weakly delegable if for any NFA  $M$  for  $L$ , there exists a  $k$  such that  $M$  has a  $k$ -delegator.*
- (ii)  *$L$  is said to be strongly delegable if there exists a  $k$  such that for every NFA  $M$  for  $L$ ,  $M$  has a  $k$ -delegator.*

The next result shows that the classes of regular languages that are weakly delegable and strongly delegable coincide. Let  $M$  be an NFA and let  $p$  be a state of  $M$ . By  $L_p$  we will denote the language accepted by  $M$  if  $p$  is chosen as the start state of  $M$  (with no other change to its definition).

**Theorem 1.** *The following statements are equivalent:*

- (1)  *$L$  is finite.*
- (2)  *$L$  has a strong delegator.*
- (3)  *$L$  has a weak delegator.*

*Proof.*

$1 \Rightarrow 2$  Let  $m$  be the length of the longest string in  $L$ . It is easy to see that any NFA for  $L$  can be “ $m$ -simulated” using a DFA, hence it has an  $m$ -delegator.

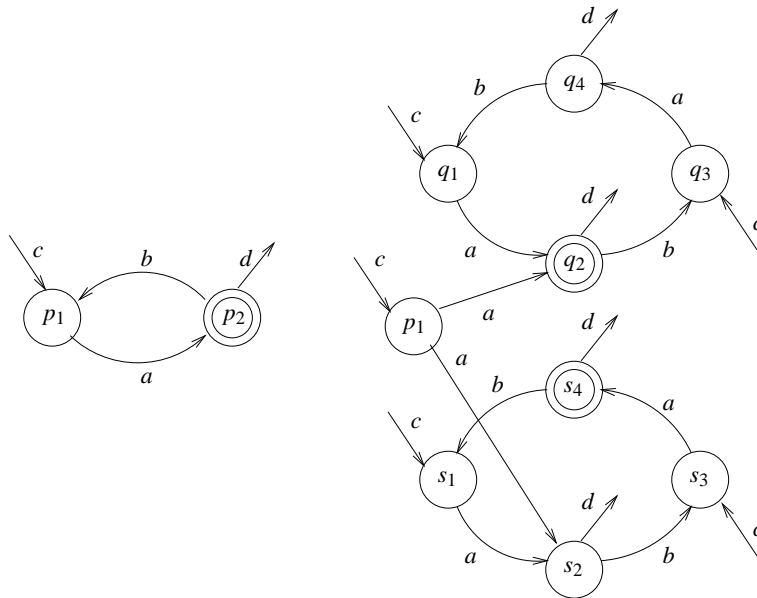
$2 \Rightarrow 3$  is obvious from the definition.

$3 \Rightarrow 1$  We prove the contrapositive, namely, if  $L$  is not finite then there exists an NFA  $M'$  for  $L$  that does not have a  $k$ -delegator for any  $k$ . Let  $M$  be a DFA for  $L$ . We assume that  $M$  is trim, that is, it does not have any useless states. Thus,  $M$  may be incomplete. Since  $L$  is infinite,  $M$  has at least one cycle.

First we consider the simpler case, in which some accepting state lies in a cycle. Fix one such cycle containing the states  $p_1, p_2, \dots, p_r$ . Thus, one of the states in the set  $\{p_1, p_2, \dots, p_r\}$  is an accepting state. Let  $L_i$  be the set of labels on the transition from  $p_i$  to  $p_{(i+1) \bmod r}$ . We define a NFA  $M'$  as follows. We

start with  $M$  and remove the states  $p_2, \dots, p_r$ . Then, we add  $4r$  states  $q_1, \dots, q_{2r}$  and  $s_1, \dots, s_{2r}$ , and add transitions to these states in such way, that for all states  $q_j, q_{r+j}, s_j$  and  $s_{r+j}$ , the equality  $L_{q_j} \cup L_{q_{r+j}} \cup L_{s_j} \cup L_{s_{r+j}} = L_{p_j}$  holds.

For all  $j \in \{1, \dots, r\}$  we consider all transition labels of  $L_j$  and add them to transitions from  $q_j$  to  $q_{j+1}$ , from  $q_{j+r}$  to  $q_{(j+r+1) \bmod 2r}$ , from  $s_j$  to  $s_{j+1}$ , and from  $s_{j+r}$  to  $s_{(j+r+1) \bmod 2r}$ . Next, we add the labels of  $L_1$  to transitions from  $p_1$  to  $q_2$  as well as to  $s_2$ . Finally, for each transition in  $M$  from  $p_j$  to any state not in the cycle, we add in  $M'$  transitions with the same label from  $q_j, q_{r+j}, s_j$  and  $s_{r+j}$  to that state. A transition in  $M$  from a state not in the cycle to a state  $p_j$  in the cycle is replaced in  $M'$  by the transitions from that state to each state  $q_j, q_{r+j}, s_j$  and  $s_{r+j}$ . Finally, consider a transition in  $M$  from  $p_j$  to  $p_t$  where  $t \neq (j+1) \bmod r$ . In  $M'$  we replace these transitions by a corresponding transition from  $q_j$  to  $q_t$ , from  $q_{r+j}$  to  $q_{r+t}$  as well as from  $s_j$  to  $s_t$  and from  $s_{r+j}$  to  $s_{r+t}$ . The accepting states in  $M'$  are chosen as follows. The accepting states of  $M$  that have not been removed will continue to be accepting states. Among the added states, accepting states are determined as follows: If  $p_i$  was an accepting state in  $M$ , then  $q_i$  as well as  $s_{r+i}$  will be chosen accepting states in  $M'$ . This construction is reflected on a small scale in Figure 4. Notice that if  $M'$  is in state

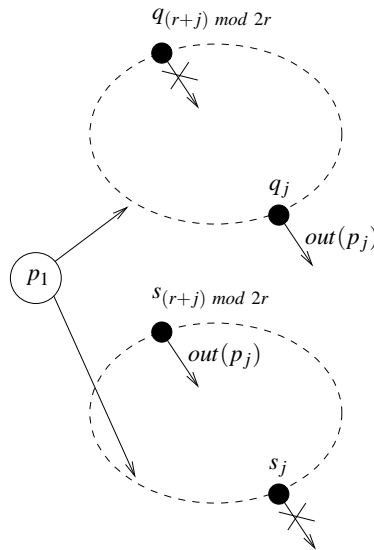


**Fig. 4.** A cycle in  $M$  and its corresponding twin cycles in  $M'$ .



$p_1$  after reading some input symbols, then by reading  $a$  and looking ahead at the next  $k - 1$  symbols does not suffice for predicting which transition should be followed. Indeed, from state  $p_1$  both words  $a(ba)^{2k}$  and  $a(ba)^{2k+1}$  lead to acceptance in  $M$ ; however, if a hypothetical  $k$ -delegator for  $M'$  commits to any particular transition from  $p_1$  on input  $a$ , then one of these two words would lead to a failing computation. It can be shown that  $L(M') = L$  and that  $M'$  does not have a  $k$ -delegator for any  $k$ . The details are straightforward.

Consider now the case when the cycle containing  $p_1, p_2, \dots, p_r$  does not have an accepting state. Since  $M$  is trim, there are states in this cycle which have transitions to some states that do not belong to the cycle (in order to have successful paths, the states in the cycle should be connected with some final states). Assume one such state is  $p_j$ , with  $j \in \{1, \dots, r\}$  and that the set of labels of transitions from  $p_j$  to states not in the cycle is denoted by  $out(p_j)$ . This state will play the role similar to that of the final states in the previous construction. Without loss of generality, we assume that  $j \neq 1$ . We construct an automaton  $M'$  as before, with the following exception: the states  $s_j$  and  $q_{(j+r) \bmod 2r}$  have no transitions to states which do not belong to the cycle (however,  $s_{(j+r) \bmod 2r}$  and  $q_j$  do have such transitions with labels in  $out(p_j)$ ). This modification is reflected in Figure 5.  $M'$  does not have a  $k$ -delegator for a reason similar to



**Fig. 5.** The construction for a cycle with no final states.

that in the previous construction: committing to a transition out of  $p_1$  would discriminate among paths using an even versus odd number of states  $p_j$ . The details are straightforward.  $\square$

In the following section we investigate machine properties related to the existence of  $k$ -delegators, as a preamble to the algorithmic approach on NFA delegation.

### 3 Basic Results on NFA Delegation

Let  $M = (Q, \Sigma, \delta, q_0, F)$  be a trim NFA and  $q \in Q$ ,  $a \in \Sigma$  such that  $\delta(q, a) = \{q_1, \dots, q_t\}$  with  $t > 1$  ( $q$  has non-deterministic transitions on input  $a$ ). As usual, by  $L_q$  we understand the language obtained by setting  $q$  to be initial state in  $M$ . Notation wise, we denote by  $av$  a word that starts with  $a$  and whose suffix obtained by removing  $a$  is  $v$ .

**Definition 3.** *With the above notations, we say that  $q$  is  $av$ -blind if for all  $i \in \{1, \dots, t\}$  the following inequality holds:*

$$\left( \bigcup_{j \in \{1, \dots, t\}, j \neq i} v^{-1} L_{q_j} \right) \setminus v^{-1} L_{q_i} \neq \emptyset .$$

This definition has the following delegation-related interpretation: if  $M$  has reached a  $w$ -blind state, then reading ahead  $w$  from the input tape does not suffice for deterministically choosing a certain next transition: each transition can potentially lead to non-acceptance for a word that should be accepted by  $M$ .

**Definition 4.** *With the above notations, we denote the blindness of  $q$  (or, the language of blind words for  $q$ ) as being the language  $B_q = \{w \in \Sigma^* / q \text{ is } w\text{-blind}\}$  .*

**Lemma 1.** *State blindness is regular and effectively computable. If  $B_q$  is finite for some  $q \in Q$ , then for every  $w \in B_q$ ,  $|w| \leq (4^{|Q|^2} + 1)^{|\Sigma|}$  .*

*Proof.* Let  $M = (Q, \Sigma, \delta, q_0, F)$  be a trim NFA and  $q \in Q$ . We construct a DFA  $M_q$  that accepts the language  $B_q$  and show that the number of states in  $M_q$  is at most  $(4^{|Q|^2} + 1)^{|\Sigma|}$ . Then, if  $B_q$  is finite, the length of the longest string accepted by  $M_q$  must be bounded by  $(4^{|Q|^2} + 1)^{|\Sigma|}$ , and the claim will follow. The details behind the construction of the DFA  $M_q$  are as follows.

For a symbol  $a \in \Sigma$ , let  $\delta(q, a_1) = \{q_1, q_2, \dots, q_t\}$ . By definition,  $w = a_1 a_2 \dots a_k$  is in  $B_q$  if and only if for each  $i \in \{1, \dots, t\}$ , the following condition holds:

$$\left( \bigcup_{j \in \{1, 2, \dots, t\}, j \neq i} (a_2 a_3 \dots a_k)^{-1} L_{q_j} \right) \setminus (a_2 a_3 \dots a_k)^{-1} L_{q_i} \neq \emptyset .$$

Denote the language on the left-side of the above expression as  $B_{q,a,i}$ . We construct a DFA  $M_{q,a,i}$  to accept  $B_{q,a,i}$  as follows. The states of this DFA are of the form  $\langle S_1, S_2 \rangle$ , where  $S_1, S_2 \subset Q$ . The transition function  $\delta'$  of  $M_{q,a,i}$  is essentially that of the cross-product of the “subset construction” DFA for  $M$  with itself. More precisely,  $\delta'(\langle S_1, S_2 \rangle, a) = \langle S_3, S_4 \rangle$  where  $S_3 = \{q \mid q \in \delta(p, a) \text{ for some } p \in S_1\}$  and similarly  $S_4 = \{q \mid q \in \delta(p, a) \text{ for some } p \in S_2\}$ . The start state of the DFA is chosen to be  $\langle \{q_i\}, \langle \{q_1, \dots, q_{i-1}, q_{i+1}, \dots, q_t\} \rangle$  and the set of accepting states is  $F' = \{\langle S_1, S_2 \rangle \mid (\bigcup_{q \in S_2} L_q) \setminus (\bigcup_{q \in S_1} L_q) \neq \emptyset\}$ .

It can be checked that the above DFA  $M_{q,a,i}$  accepts the language  $B_{q,a,i}$ . The number of states in  $M_{q,a,i}$  is upper-bounded by  $4^{|Q|}$ . Next, we construct a DFA  $M_{q,a}$  accepting the language  $L_{q,a} = \bigcap_{i \in \{1, \dots, t\}} B_{q,a,i}$ . The size of  $M_{q,a}$  is upper-bounded by  $4^{|Q|^2}$ , since  $t \leq |Q|$ . Then, a DFA for  $aB_{q,a}$  has one extra state, and the DFA for  $B_q = \bigcup_{a \in \Sigma^*} aB_{q,a}$  will have a size upper-bounded by  $(4^{|Q|^2} + 1)^{|\Sigma|}$ . This completes the proof.  $\square$

*Remark 3.* One may notice that if the blindness of a state  $q$  of  $M$  is finite, then  $q$  may potentially be used in some  $k$ -lookahead delegator for  $M$ , with  $k$  sufficiently large. Indeed, denoting  $k - 1$  to be the length of a longest word in  $B_q$ , one can observe that a buffer content of size  $k$  allows a delegator to make deterministic decisions on which transition from  $q$  should be followed. The reason is that for any buffer content  $w$ , with  $|w| \geq k$ , the state  $q$  is not  $w$ -blind. Consequently, the “interesting” states are those with infinite blindness.

**Lemma 2.** *For any state  $q$ ,  $B_q$  is prefix-closed, except for the empty word.*

*Proof.* It suffice to prove that if a state  $q$  is  $auv$ -blind then it is  $au$ -blind as well. Let  $\delta(q, a) = \{q_1, \dots, q_t\}$ , and assume by contradiction that  $q$  is not  $au$ -blind. Then, there exists an index  $i \in \{1, \dots, t\}$  such that  $u^{-1}L_{q_i} \supset \bigcup_{j \neq i} u^{-1}L_{q_j}$ . But since  $q$  is  $auv$ -blind, there exists  $z \in \Sigma^*$  such that  $uvz \in \bigcup_{j \neq i} L_{q_j}$  and  $uvz \notin L_{q_i}$ . This contradicts the previous statement, through the word  $vz$ .  $\square$

The following corollary gives a sufficient condition for the existence of lookahead delegators.

**Corollary 1.** *If an NFA  $M$  has all its states finitely blind, then it accepts a lookahead delegator.*

*Proof.* One can construct a  $k$ -lookahead delegator, with  $k$  greater than the maximum length of the words belonging to any blind language of a state in  $M$ . This is a generalization of Remark 3.  $\square$

**Definition 5.** *A state  $q$  is  $k$ -blind if there exists a word  $w \in \Sigma^k$  such that  $q$  is  $w$ -blind.*

The following result is a reflection of Lemma 2.

**Lemma 3.** *If a state  $q$  of an NFA  $A$  is  $k$ -blind,  $k \geq 2$ , then it is  $l$ -blind for all  $l \in \{1, \dots, k-1\}$ .*

*Proof.* (sketch) If a state  $q$  is  $k$ -blind, then there exists a word  $w = av$  with  $|w| = k$  such that  $q$  is  $w$ -blind. Let  $av = ayz$  be a factorization of  $w$  with  $|ay| = l$ . We prove that  $q$  is  $ay$ -blind.

Let  $\delta(q, a) = \{q_1, \dots, q_t\}$ . Since  $q$  is  $w$ -blind we have that for all  $i \in \{1, \dots, t\}$  the following holds:

$$\left( \bigcup_{j \in \{1, \dots, t\}, j \neq i} (yz)^{-1} L_{q_j} \right) \setminus (yz)^{-1} L_{q_i} \neq \emptyset .$$

If we denote by  $\Delta_i$  the above set-difference, we have that

$$\left( \bigcup_{j \in \{1, \dots, t\}, j \neq i} y^{-1} L_{q_j} \right) \setminus y^{-1} L_{q_i} \supseteq z \Delta_i \neq \emptyset$$

holds for all  $i \in \{1, \dots, t\}$ , which shows that  $q$  is  $ay$ -blind, hence it is  $l$ -blind.  $\square$

The following result provides a necessary condition for the existence of NFA delegators.

**Corollary 2.** *If the initial state of an NFA is infinitely blind then the NFA has no  $k$ -lookahead delegator for any integer  $k$ .*

*Proof.* (sketch) Suppose the automaton accepts a  $k$ -lookahead delegator despite the fact that its initial state  $q_0$  is infinitely blind. We choose a word  $w = av$  with  $|w| > k$  such that  $q_0$  is  $w$ -blind. Observe that  $w \in \text{pref}(L)$ , where  $L$  is the language accepted by the NFA. Let  $\delta(q_0, a) = \{q_1, \dots, q_t\}$  and assume that the input word has  $w$  as a prefix. In this case, the lookahead delegator must commit deterministically (regardless on what follows after  $w$ ) to one transition, say,  $(q_0, a, q_i)$ , with  $i \in \{1, \dots, t\}$ . But by the definition of  $av$ -blindness, we know that there exist a word  $z \in \Sigma^*$  such that  $avz \in L$  and  $\delta(q_i, vz)$  does not contain any final state. This word is rejected by the delegator, despite the fact that it belongs to the language.

Here we have silently used the fact that if  $q_0$  is  $|w|$ -blind, then it must be also  $k$ -blind, since  $k < |w|$ . This fact ensured the existence of  $z$ .  $\square$

*Remark 4.* Notice that, by Lemma 1, the conditions in Corollary 1 and Corollary 2 are testable. Notice also that a  $k$ -lookahead delegator for an NFA  $M$  must have  $k \geq r$ , where  $r$  is the smallest integer such that the initial state of  $M$  is not  $r$ -blind.

**Definition 6.** A delegator for  $M$ ,  $f : Q \times \Sigma^k \times Q$  is *trim* if all its “predictions” (or, delegations) are used in some successful computations ( $f$  needs not be defined everywhere).

The following results will be used in proving the correctness of Algorithm 1 in Section 4.2, which computes  $k$ -delegators.

**Lemma 4.** If  $f : Q \times \Sigma^k \rightarrow Q$  is a trim delegator for  $M$ , then

$$f(p, av) = q \Rightarrow \left( \forall b \in \Sigma \text{ s.t. } vb \in \text{pref}(L_q) : f(q, vb) \neq \emptyset \right) .$$

*Proof.* Assume that  $f(p, av) = q$ , and take  $vb \in \text{pref}(L_q)$ . There exists a word  $z$  such that  $vbz \in L_q$ . Since  $f$  is trim, there exists a word  $x$  such that while reading  $xav$  the delegator reaches *deterministically*  $p$  while holding  $av$  in its buffer. Observe now that  $xavbz \in L$  and the only way for the delegator to accept it is to make a choice for  $f(q, vb)$ .  $\square$

**Corollary 3.** If  $f : Q \times \Sigma^k \rightarrow Q$  is a trim delegator for  $M$ , then

$$f(p, av) = q \Rightarrow \left( \forall b \in \Sigma \text{ s.t. } vb \in \text{pref}(L_q) : vb \notin B_q \right) .$$

**Corollary 4.** If  $f : Q \times \Sigma^k \rightarrow Q$  is a trim delegator for  $M$ , and if  $v_1 \dots v_k \in B_q$  for some state  $q \in Q$  then  $f(p, av_1 \dots v_{k-1}) \neq q$  for all  $p \in Q, a \in \Sigma$ .

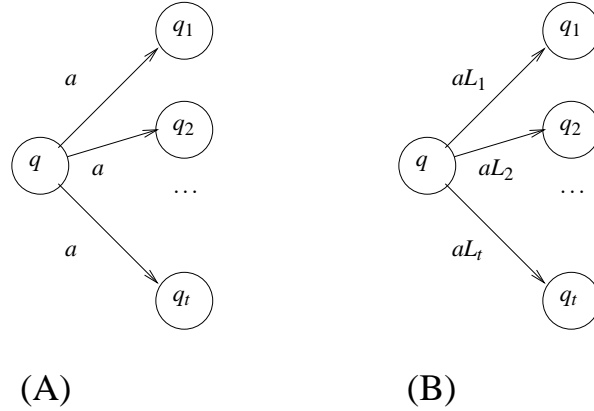
*Proof.* Suppose there exists  $p \in Q$  such that  $f(p, av_1 \dots v_{k-1}) = q$ . By Lemma 4,  $f(q, v_1 \dots v_{k-1}b) \neq \emptyset$ , for all  $b \in \Sigma$  such that  $v_1 \dots v_{k-1}b \in \text{pref}(L_q)$ . But  $v_1 \dots v_{k-1}v_k \in B_q \subseteq \text{pref}(L_q)$ , which implies that  $f(q, v_1 \dots v_k)$  must be defined despite the fact that  $q$  is  $v_1 \dots v_k$ -blind. This is a contradiction.  $\square$

In the following we give another definition (hence, another formalism) for NFA delegation, equivalent to Definition 1. By  $L_M$  we understand the language accepted by  $M$ .

**Definition 7.** Let  $M = (Q, \Sigma, \delta, q_0, F)$  be an NFA. A DFA  $D$  with  $k$ -lookahead buffer is a delegator for  $M$  if

1.  $L_M = L_D$  ,
2.  $M$  and  $D$  have identical transition graphs with the exception of labels, which are in the following relation:

For each transition  $\delta(q, a) = \{q_1, \dots, q_t\}$  in  $M$ , as depicted in Figure 6.A, there correspond  $t$  lookahead transitions in  $D$ , as shown in Figure 6.B, with the following properties: (a) for all  $i \in \{1, \dots, t\}$  the language  $L_i$  has words of length less than  $k$ ; and (b) for all  $i, j \in \{1, \dots, t\}$  with  $i \neq j$  we have  $aL_i \cap aL_j = \emptyset$ .



**Fig. 6.** Transitions in a NFA and its delegator.

In the previous definition we allow  $L_i$  to be  $\emptyset$ , with the meaning that a transition labeled  $a\emptyset = \emptyset$  is “non-existent”, i.e., the delegator chooses to never use it.

Notice that the second condition of Definition 7 implies that  $D$  is a deterministic lookahead automaton. Indeed,  $D$  operates as following: if a state  $q$  is reached and a word  $av$  is in the lookahead buffer, the automaton searches for  $av$  in all languages  $aL_i$ . If it finds it, i.e.,  $av \in aL_i$  for some  $i$ , it will choose the corresponding transition labeled  $aL_i$  and will advance in the next state  $q_i$ .

**Corollary 5.** *With the above notations, if  $M$  has a lookahead delegator, then it has one such that for every state  $q \in Q$  and every letter  $a \in \Sigma$ , we have  $B_q \cap aL_i = \emptyset$ ,  $\forall i \in \{1, \dots, t\}$ .*

*Proof.* Let  $q$  be a state in  $M$  and  $a$  be a symbol with  $\delta(q, a) = \{q_1, \dots, q_t\}$ ,  $t > 1$ . Suppose that the corresponding transitions in a delegator  $D$  for  $M$  are  $(q, aL_1, q_1), \dots, (q, aL_t, q_t)$ . If for some  $i \in \{1, \dots, t\}$  we have  $av \in B_q \cap aL_i$ , then one can easily observe that the delegator can never use the transition  $(q, av, q_i)$  since  $q$  is  $av$ -blind. Hence, one can safely remove  $av$  from the language  $aL_i$ .  $\square$

This corollary gives a “normal form” for lookahead delegators, by discarding label information that is never used.

We now have sufficient tools for investigating algorithmic aspects related to NFA delegation.

#### 4 Complexity of Determining if a $k$ -Delegator Exists

We consider the following computational problems.

**Problem 1.** Let  $k$  be a fixed integer (not part of the input)

*Input:* An NFA  $M$

*Output:* “YES” if and only if  $M$  has a  $k$ -delegator, “NO” otherwise.

**Problem 2.**

*Input:* An NFA  $M$  and an integer  $k$  (in unary)

*Output:* “YES” if and only if  $M$  has a  $k$ -delegator, “NO” otherwise.

**Problem 3.**

*Input:* An NFA  $M$

*Output:* “YES” if and only if  $M$  has a  $k$ -delegator for some  $k$ , “NO” otherwise.

As in the previous sections, we assume that  $M$  is trim. Recall the result in Lemma 1, which turns out to be useful in addressing the complexity of the above problems: for a state  $q$  of an NFA  $M$ , the language  $B_q$ , of blind words for  $q$ , is regular and  $(4^{|Q|^2} + 1)^{|\Sigma|}$  provides an upper-bound on the state complexity of  $B_q$ . In the following section we first tackle the special case in which the input NFA is unambiguous. The subsequent section will deal with the general case of NFA's that may be ambiguous.

#### 4.1 The Case of Unambiguous NFA's

In this subsection we show that in the case of an *unambiguous* NFA as input, Problem 1 is in P, Problem 2 is in co-NP, and Problem 3 is in PSPACE.

*Note 1.* We leave for further work to answer the question whether Problem 2 is co-NP-complete and Problem 3 is PSPACE-complete for unambiguous NFA's.

We begin with a definition, which turned out to be very useful in providing characterizations for NFA delegation in the unambiguous case, and necessary conditions for the general case.

**Definition 8.** Let  $M = (Q, \Sigma, \delta, q_0, F)$  be a NFA, and let  $q \in Q$  and  $w \in \Sigma^*$ . A pair  $(q, w)$  is said to be *crucial* for  $M$  if the following holds: There exist strings  $x$  and  $y$  such that

1.  $xwy$  is in  $L(M)$ , and
2. every accepting computation of  $xwy$  reaches state  $q$  after reading the input  $x$ .

Then, the following lemmas hold for unambiguous NFA.

**Lemma 5.** If  $M$  is unambiguous, then for every state  $q$  and for every string  $w \in \text{pref}(L_q)$ , the pair  $(q, w)$  is crucial.

*Proof.* Since  $M$  is assumed to be trim, every state  $q \in Q$  is useful, i.e., there exists a string  $x$  such that  $q \in \delta(q_0, x)$  and a string  $y$  such that  $\delta(q, wy) \cap F \neq \emptyset$ . Existence of another accepting computation of the string  $xwy$  that does not reach the state  $q$  after reading  $x$  would imply that there are two accepting computations for the string  $xwy$  contradicting the fact that  $M$  is unambiguous.  $\square$

**Lemma 6.** *Let  $M$  be an unambiguous NFA,  $q$  be a state of  $M$  and  $w \in \Sigma^k$  for some integer  $k$ . If  $(q, w)$  is crucial for  $M$  and if  $q$  is  $w$ -blind, then  $M$  cannot have a  $k$ -delegator.*

*Proof.* (sketch) By definition, there exist strings  $x$  and  $y$  such that  $xwy \in L(M)$  and the unique accepting computation on the string  $xwy$  reaches  $q$  after reading the prefix  $x$ . Suppose  $M$  has a  $k$ -delegator. Let  $D$  be a  $k$ -delegator(simulator) for  $M$ , as defined in Definition 7. It is clear that the state reached by  $D$  on reading the prefix  $x$  of the input string  $xwy$  is  $q$ . Now  $D$  will not be able to continue the simulation from the state  $q$  since it is  $w$ -blind.  $\square$

**Lemma 7.** *An unambiguous NFA  $M$  has a  $k$ -delegator if and only if for every state  $q$  of  $M$  there exists no string  $w$  of length greater than or equal to  $k$  such that  $q$  is  $w$ -blind.*

*Proof.* (“ $\Rightarrow$ ”) Let  $M$  have a  $k$ -delegator. Suppose there is a state  $q$  and a string  $w$  of length greater than or equal to  $k$  such that  $q$  is  $w$ -blind. It is clear that  $w \in \text{pref}(L_q)$ ; and by the above lemmas,  $M$  cannot have a  $k$ -delegator - fact which contradicts the hypothesis.

(“ $\Leftarrow$ ”) It follows immediately from the above lemmas.  $\square$

**Lemma 8.** *Let  $M = (Q, \Sigma, \delta, q_0, F)$  be an unambiguous NFA,  $k$  be an arbitrary integer, and let  $Q_1, Q_2 \subseteq Q$  with  $Q_1 \cap Q_2 = \emptyset$  and  $Q_1 \cup Q_2 \subseteq \delta(q_0, w)$  for some word  $w \in \Sigma^*$ . Then testing whether*

$$\left( \bigcup_{q \in Q_1} L_q \right) \setminus \left( \bigcup_{q \in Q_2} L_q \right) \neq \emptyset$$

*can be done in polynomial time.*

*Proof.* The basic idea for such a polynomial time algorithm is due to Stearns and Hunt [Stearns and Hunt, 1985], that containment and equivalence problems are polynomial time decidable for unambiguous NFA's. Their approach was to use linear recurrence equations for designing an efficient algorithm for this problem. Here we use a simpler (but essentially equivalent) approach based on the transfer matrix technique.



We first show that containment problem for unambiguous NFA is solvable in polynomial time. For an unambiguous NFA  $M = (Q, \Sigma, \delta, q_0, F)$ , let us define a  $|Q| \times |Q|$  matrix  $T_M$  as follows. We label the states of  $M$  as  $\{1, 2, \dots, |Q|\}$ . If there are  $k$  transitions from state  $i$  to state  $j$ , then set  $T_M[i, j] = k$ . Denote by  $v$  the column vector  $v = [v_1, v_2, \dots, v_{|Q|}]$  where  $v_i$  is 1 if  $i$  is an accepting state and 0 otherwise. Denote also by  $u$  the row vector  $u = [u_1, u_2, \dots, u_{|Q|}]$  where  $u_i$  is 1 if  $i$  is the start state and 0 otherwise. Now, it is easy to check that the number of strings of length  $m$  (for any integer  $m$ ) accepted by  $M$  is given by  $uT_M^m v$ . It is clear that, for a given  $m$ , the entries of  $T_M^m$  can be computed using  $O(|Q|^3 \log m)$  arithmetic operations by the “repeated squaring” technique. Note also that the bit-size of the integers in the matrix  $T_M^m$  is bounded by  $O(m^c)$  bits for some constant  $c$  - proving that the claim of polynomial time bound is “genuine”, i.e., it holds in the bit complexity model as well. In summary, the number of strings of length  $m$  accepted by an unambiguous NFA  $M$  can be computed in time complexity that is a polynomial in  $|M|$  and  $\log m$ .

Now let  $M_1$  and  $M_2$  be two unambiguous NFA's. We show, using Stearns and Hunt's technique, that the containment problem  $L(M_1) \subseteq L(M_2)$  (or its complement, namely  $L(M_2) \setminus L(M_1) \neq \emptyset$ ) can be solved in time polynomial in  $|M_1| + |M_2|$ . The basic idea is to reduce (in polynomial time) the containment problem to the *conditional equivalence* problem, which is as follows:

*Conditional Equivalence Problem.* Given two unambiguous NFA's  $M_3$  and  $M_4$  such that  $L(M_3) \subseteq L(M_4)$ , determine if  $L(M_3) = L(M_4)$ .

Since  $L(M_1) \subseteq L(M_2)$  if and only if  $L(M_1) = L(M_1) \cap L(M_2)$ , we can choose  $M_4 = M_1$  and  $M_3$  to be an NFA that accepts  $L(M_1) \cap L(M_2)$ . The standard “pair construction” [Hopcroft and Ullman, 1979] for intersection for languages accepted by NFA's results in the size of  $M_3$  being bounded by  $|M_1| \times |M_2|$  and it is also easy to check that  $M_3$  is unambiguous as well. In view of the above reduction, it is enough to show that there exists a polynomial time algorithm for the conditional equivalence problem for unambiguous NFA's. This algorithm is as follows: for every  $k \in \{1, 2, \dots, |Q_3| + |Q_4|\}$  check whether the number of strings of length  $k$  accepted by  $M_3$  and  $M_4$  agree. Then  $L(M_3) = L(M_4)$  if and only if the above check succeeds. It is not hard to show that this check provides a necessary and sufficient condition for the conditional equivalence problem. From the algorithm based on the transfer matrix technique, this check can be done in polynomial time and the claim follows.

We conclude the proof by showing that the given problem can be reduced to the containment problem for unambiguous NFA's. Let us define the NFA's  $M_1$  and  $M_2$  as follows:  $M_1$  ( $M_2$ ) is constructed from a copy of  $M$  by creating a new

start state  $n_1$  ( $n_2$ ) and adding an  $\varepsilon$ -transition from  $n_1$  ( $n_2$ ) to each state in  $Q_1$  ( $Q_2$ ). Finally, we remove the  $\varepsilon$ -transitions and trim  $M_1$  and  $M_2$ . We now show that  $M_1$  and  $M_2$  are unambiguous NFA's. We present an argument only for  $M_1$ , since a similar argument holds for  $M_2$  as well. Suppose  $M_1$  is ambiguous. Then there are two accepting computations for some accepting string in  $M_1$ . Suppose the two accepting paths branch for the first time at state  $s$ . Let the label of the two successful paths branching from  $s$  be  $y$ . If  $s \neq n_1$ , then  $s$  is a state in  $M$ . Let  $x$  be a string that takes the start state  $q_0$  of  $M$  to state  $s$ . It follows that there are at least two accepting computations for the string  $xy$  in  $M$ , contradicting the fact that it is unambiguous. If  $s = n_1$  on the other hand, then it follows that the string  $xy$  can be derived in two ways in  $M$ , again a contradiction. Thus  $M_1$  (and  $M_2$ ) are both unambiguous. It is easy to see that  $L(M_1) \setminus L(M_2) \neq \emptyset$  if and only if

$$\left( \bigcup_{q \in Q_1} L_q \right) \setminus \left( \bigcup_{q \in Q_2} L_q \right) \neq \emptyset$$

and this completes the proof of the lemma.  $\square$

We are now ready to show the first main result of this subsection.

**Theorem 2.** *Problem 1 can be solved in polynomial time when the input NFA is unambiguous.*

*Proof.* The input to the problem are: a (trim) unambiguous NFA  $M = (Q, \Sigma, \delta, q_0, F)$  and an integer  $k$  (in unary). By the Lemmas 5 and 6, it is clear that  $M$  has a  $k$ -delegator if and only if, for every state  $q \in Q$ , all strings in  $B_q$  have a length smaller than  $k$ . To check this condition, we proceed as follows: For a symbol  $a \in \Sigma$ , let  $\delta(q, a) = \{q_1, q_2, \dots, q_t\}$ . Recall that  $w = av_2 \dots v_k$  is in  $B_q$  if and only if for each  $i$ , the following condition holds:

$$\left( \bigcup_{j \in \{1, 2, \dots, t\}, j \neq i} (v_2 v_3 \dots v_k)^{-1} L_{q_j} \right) \setminus (v_2 v_3 \dots v_k)^{-1} L_{q_i} \neq \emptyset .$$

We employ a notation used in Lemma 1, that the language on the left-side of the above expression is denoted  $B_{q,a,i}$ . For each pair  $(q, w)$  where  $w = v_1 v_2 \dots v_k$ , we check whether  $w \notin B_{q,v_1,i}$  as follows. We compute the sets of states  $R_1 = \{p \mid p \text{ is reachable from } q_i \text{ on } v_2 v_3 \dots v_k\}$ , and  $R_2 = \{p \mid p \text{ is reachable from } q_j \text{ for some } j \neq i \text{ on } v_2 \dots v_k\}$ . Note that for a given pair  $(q, w)$ , all these sets can be constructed in time polynomial in  $|M|$ , and use the algorithm of Lemma 8 to test if

$$\left( \bigcup_{q \in R_2} L_q \right) \setminus \left( \bigcup_{q \in R_1} L_q \right) \neq \emptyset$$

If this is true, then we try the next  $i$  from the set  $\delta(q, a)$ . If no  $i$  works for a particular  $w$ , then we return “NO”. Otherwise, we continue with the next string  $w$  of length  $k$  in  $L_q$ . If we find a successful simulating move for every pair  $(q, w)$  where  $q \in Q$  and  $w \in L_q$ , then the algorithm returns “YES”. It is not hard to check that the total time complexity of this algorithm is  $O(2^k P(|M|))$  for some polynomial  $P$  and hence for a fixed  $k$ , the algorithm runs in polynomial time.  $\square$

The proof of the next theorem follows very closely that of the above theorem so we will only present a sketch.

**Theorem 3.** *Problem 2 is in co-NP when the input NFA  $M$  is unambiguous.*

*Proof.* The algorithm is similar to the above - except that the algorithm will guess a pair  $(q, v_1 \dots v_k)$  for some  $q \in Q$  and some string  $w = v_1 \dots v_k \in \Sigma^k$  and will check that  $w \in B_{q, v_1, i}$  for every  $i$ . Note that the sets  $R_1$  and  $R_2$  can be computed in time  $O(k|M|)$  and this is why it is crucial to assume that  $k$  is given in unary. The rest of the details are the same.  $\square$

It is not hard to modify the algorithm(s) described previously for Problems 1 and 2 such that it actually constructs the  $k$ -delegator in the case of an “YES” answer.

We will finally discuss Problem 3 for unambiguous NFA’s. The following lemma is easy to prove.

**Lemma 9.** *An unambiguous NFA  $M$  has a delegator if and only if  $B_q$  is finite for every state  $q$  of  $M$ .*

*Proof.* (“ $\Leftarrow$ ”) Let  $l$  be the length of the longest string in  $\bigcup_q B_q$ . It is easy to see that, with  $k = l - 1$ ,  $M$  has a  $k$ -delegator.

(“ $\Rightarrow$ ”) Suppose that  $M$  has a  $k$ -delegator for some integer  $k$  and assume by contradiction that the conclusion does not hold. Then there exists a state  $q$  such that  $B_q$  is infinite. This implies that  $B_q$  has a string of length greater than  $k$ . Let  $w$  be such a string. Clearly,  $w$  is also in  $pref(L_q)$ . Since  $M$  is unambiguous, by Lemma 5,  $(q, w)$  is crucial. This leads to a contradiction.  $\square$

**Theorem 4.** *Problem 3 is decidable in PSPACE for unambiguous NFA.*

*Proof.* Given an unambiguous NFA  $M$ , it is enough to check that  $B_q$  is finite for every state  $q$  of  $M$ . Since we can explicitly construct a DFA for each language  $B_q$  and since finiteness of a regular language is decidable, the conclusion follows.

To show that the problem is in *PSPACE*, we have to show that finiteness of each  $B_q$  can be tested in *PSPACE*. One way to show this is by showing that the complement problem is in *PSPACE* (since *PSPACE* is closed under complement.) Recall that in Lemma 1, we described a construction of the DFA for  $B_{q,a,i}$ . Instead of constructing this DFA explicitly, the algorithm guesses a string  $\tau$  of length  $r$  and it checks whether  $\tau$  is in  $B_{q,a,i}$  for each  $a \in \Sigma$  and each  $i$ . Note that  $\tau$  cannot be explicitly written down since it would require exponential space to do so. Instead,  $r = |\tau|$  is written in binary, and the successive symbols of  $\tau$  are guessed, followed by the check whether  $\tau$  is in  $B_{q,a,i}$ . Finally, it is checked whether  $r > 4^{|\mathcal{Q}|^2}$ . Using the fact that [if a DFA  $M$  with  $m$  states accepts a string of length  $m$  or more then  $L(M)$  is infinite], it follows that the algorithm described above verifies “NO” instances correctly. The above algorithm is a non-deterministic polynomial space algorithm, but since  $NPSPACE = PSPACE$ , and since *PSPACE* is closed under complement, the above algorithm can be readily converted into a *PSPACE* algorithm.  $\square$

## 4.2 The Case of Ambiguous NFA’s

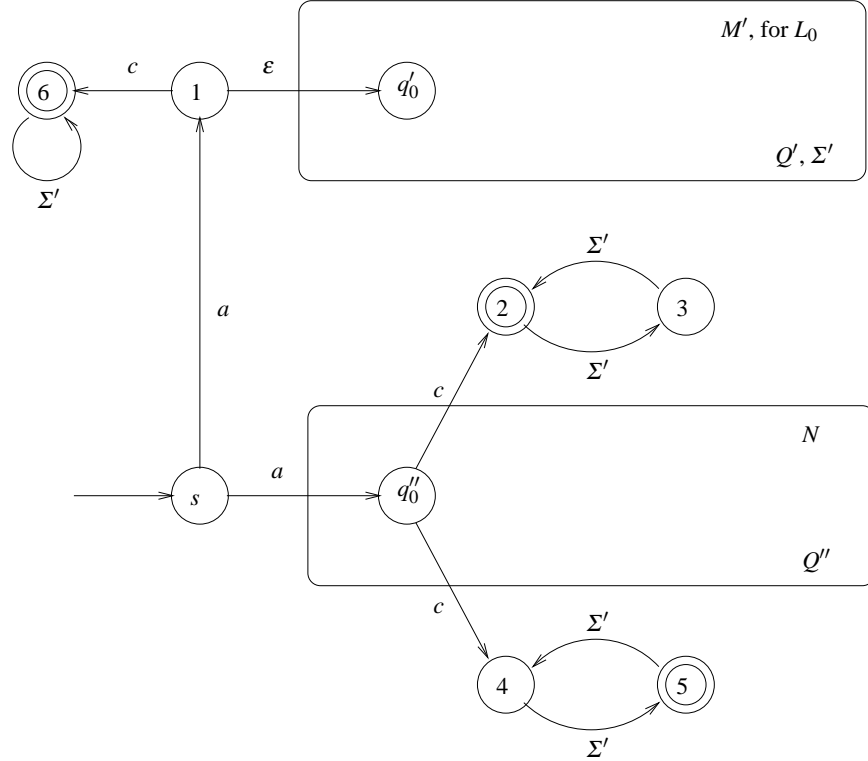
In this section, we describe an algorithm for Problem 1 in the general case, namely the case in which  $M$  can be ambiguous. We will show that the problem is *PSPACE*-complete. This immediately implies that Problems 2 and 3 are *PSPACE*-hard in the general case.

**Theorem 5.** *Problem 1 for the general case is PSPACE-complete. The hardness holds for every fixed  $k = 1, 2, 3, \dots$ .*

*Proof.* To show its membership to *PSPACE*, we will use the brute-force, exhaustive search approach as in [Dang et al., 2004]. Given an NFA  $M = (Q, \Sigma, \delta, q_0, F)$ , we generate all possible  $k$ -delegators and check if one of them is a valid  $k$ -delegator. For a fixed  $k$ , the size of a  $k$ -delegator is bounded by  $O(|M|)$  and thus each one of them can be successively generated in *PSPACE*. Since whether a given  $k$ -delegator  $M'$  correctly simulates a NFA  $M$  can be checked in *PSPACE* (this problem is equivalent to NFA equivalence problem), it follows that Problem 1 is in *PSPACE*.

To show that it is *PSPACE*-hard, we will reduce to it the problem “Is  $L(N) \subseteq L_0$  ?” where  $N$  is a NFA and  $L_0$  is a *fixed* unbounded language. It is known ([Hunt et al., 1976]) that this problem is *PSPACE*-hard when the size of the alphabet over which  $N$  is defined is at least 2 (if the alphabet size of  $N$  is 1, it is easy to see that the test “Is  $L(N) \subseteq L_0$  ?” can be done in *co-NP*). The reduction is as follows: we describe a polynomial time algorithm that, given  $N$ , constructs an NFA  $M$  such that  $M$  has a 1-delegator if and only if  $L(N) \subseteq L_0$ .

Let  $Q''$  be the state set of  $N$ ,  $q_0''$  be its start state, and let  $M'$  be a DFA that accepts the language  $L_0$ ,  $Q'$  be the state set of  $M'$  and  $q_0'$  be its start state. Denote  $\Sigma'$  to be the alphabet over which  $M'$  is defined. We define an automaton  $M = (Q, \Sigma, \delta, s, F)$  as following. We choose the alphabet  $\Sigma$  to be  $\Sigma' \cup \{a, c\}$ , where  $a$  and  $c$  are two new symbols. We set  $Q = \{s, 1, 2, 3, 4, 5, 6\} \cup Q' \cup Q''$ , where  $s, 1, 2, 3, 4, 5$  and  $6$  are new state symbols. The transition relation  $\delta$  is defined as follows:  $\delta(s, a) = \{1, q_0''\}$ ,  $\delta(1, \varepsilon) = \{q_0'\}$ ,  $\delta(1, c) = \{6\}$ ,  $\delta(q_0'', c) = \{2, 4\}$ , and for all  $b \in \Sigma'$  we have  $\delta(2, b) = \{3\}$ ,  $\delta(3, b) = \{2\}$ ,  $\delta(4, b) = \{5\}$ ,  $\delta(5, b) = \{4\}$  and  $\delta(6, b) = \{6\}$ . In addition,  $\delta$  includes all the transitions of  $N$  and  $M'$ . Figure 7 details the construction of  $M$  in terms of  $N$  and  $M'$ . The set of accepting states of  $M$  will be the set of accepting states of  $M'$  and  $N$ , to which we add the states  $2, 5$  and  $6$ . To finalize the construction of  $M$  we remove the only  $\varepsilon$ -transition from state  $1$  using the standard  $\varepsilon$ -removal algorithm. The



**Fig. 7.** The construction of  $M$ , accepting  $a(c\Sigma'^* \cup L_0 \cup L(N))$ .

proof is complete by the following claim:

*Claim.*  $M$  has a 1-delegator if and only if  $L(N) \subseteq L_0$ .

*Proof.* Suppose  $M$  has a 1-delegator, which reads the input  $a$  from the initial state  $s$ . The delegator has two choices, namely 1 and  $q_0''$ . Note that the choice  $q_0''$  is not a valid one since all strings in  $c\Sigma'^*$  are in its blind set. Thus, the delegator is forced to choose  $f(s, a) = 1$ . At this point (being in state 1), in order to correctly simulate  $M$  it is necessary that  $L(N) \subseteq L_0$ . Indeed, if this was not true, then there would be a string  $w \in L(N) \setminus L_0$  with  $aw \in L(M)$ , and the delegator would reject  $aw$  (since there would be no successful computation starting at state 1 and labeled  $w$ ).

Conversely, suppose  $L(N) \subseteq L_0$ . Then, it is easy to see that  $M$  has a 1-delegator. Starting at state  $s$  and on input  $a$ , the delegator chooses  $f(s, a) = 1$ , and from this point it continues the simulation of  $M$  deterministically, since the set of states reachable from 1 have deterministic transitions. This simulation is correct since all the strings that can be accepted by taking the other branch (namely via  $q_0''$ ) can also be accepted from state 1.

This completes the proof for  $k = 1$ . The proof for the other values of  $k$  can be obtained by minor modifications of the above proof, hence the details will be omitted.  $\square$

*Note 2.* In the above construction, we needed a 4-letter alphabet for the construction of  $M$ . It would be interesting to extend the *PSPACE*-completeness proofs to smaller size alphabets.

Next, we describe a more efficient algorithm for Problem 1 in the general case. We start with a simple, yet important remark.

*Remark 5.* Let  $p$  be a state of  $M$ ,  $av \in \Sigma^k$ , and  $\delta(p, a) = \{q_1, \dots, q_t\}$  ( $t > 0$ ). If a  $k$ -delegator for  $M$  reaches state  $p$  with  $av$  in its buffer, it must/will choose a state  $q_i \in \delta(p, a)$  such that

$$v^{-1}L_{q_i} \supseteq \bigcup_{l \in \{1, \dots, t\}, l \neq i} v^{-1}L_{q_l} .$$

If two such choices,  $q_i$  and  $q_j$ , were possible in two delegator instance, then

$$v^{-1}L_{q_i} = v^{-1}L_{q_j} .$$

Consequently, an algorithm that aims at constructing a  $k$ -delegator would consider all state choices  $q_i$  as above, and test each against a same test set  $W = \{vb/vb \in \text{pref}(L_{q_i})\}$  which is independent of  $q_i$ :

$$\{vb/vb \in \text{pref}(L_{q_i})\} = \{vb/vb \in \text{pref}(L_{q_j})\} .$$

To improve algorithm's formalism, we give the following definition.

**Definition 9.** Let  $q$  be a state in  $M$ ,  $w = a_1 \dots a_k$  and  $\delta(q, a_1, \dots, a_k) = \{q_1, \dots, q_t\}$ . A state  $q_i$  is potential for  $(q, w)$  if it verifies:

$$(a_2 \dots a_k)^{-1} L_{q_i} \supseteq \bigcup_{l \in \{1, \dots, t\}, l \neq i} (a_2 \dots a_k)^{-1} L_{q_l} .$$

Denote  $P(q, w)$  the set of all potential states for  $(q, w)$ .

Notice that the above condition is related to “state blindness”, in the sense that a state  $q$  is  $w$ -blind if and only if  $P(q, w) = \emptyset$ . Notice also that  $P(q, w)$  is obviously computable for any  $q$  and  $w$ .

**Algorithm 1** at page 23 computes a  $k$ -delegator for a given trim NFA  $M$  and an integer  $k > 0$ . It uses a vector  $V$  which stores, for every state  $q$  of  $M$ , a set of words  $w \in \text{pref}_k(L_p)$  for which a hypothetical delegator must not reach  $p$  with  $w$  in its buffer ( $w$  is called a “forbidden” word for  $p$ ). The first part of the algorithm decides whether a  $k$ -delegator for  $M$  exists, by constructing  $V$  and testing whether  $V[q_0] = \emptyset$ , where  $q_0$  is the initial state of  $M$ . If  $V[q_0] = \emptyset$ , the second part of the algorithm constructs a  $k$ -delegator stored in a table  $T[Q, \Sigma^{\leq k}]$ . It does so in two phases: first, it computes the values in  $T[Q, \Sigma^{=k}]$ , which are filled recursively by procedure “*construct*”, after which it completes the table with the values in  $T[Q, \Sigma^{<k}]$  - done by function “*extend*”.

### Correctness of Algorithm 1.

We start by making the following observations:

1. If the algorithm responds “YES”, then the obtained  $k$ -delegator is trim. In some sense, this shows an improvement from the brute-force algorithm which tests for any imaginable  $k$ -delegator whether it is or it is not equivalent with  $M$ .
2. Corollary 4 justifies the step denoted by (\*) in the main algorithm.
3. Lemma 4 and Remark 5 are the theoretical support for the step denoted by (\*\*) in the definition of *construct*.
4. Corollary 3 justifies why after the initialization of  $T$  there is no more need to set cells of  $T$  to *NIL*.

**Definition 10.** For  $w \in \text{pref}(L_q)$ , we say that  $(q, w)$  is forbidden, or that  $w = a_1 \dots a_k$  is a forbidden word for  $q$ , if one of the following two conditions is satisfied, recursively:

1.  $q$  is  $w$ -bind;

---

**Algorithm 1** Computing a  $k$ -delegator.
 

---

**Input:** a trim NFA  $M = (Q, \Sigma, \delta, q_0, F)$  and an integer  $k > 0$

**Output:** “YES” and a  $k$ -delegator ( $T$ ) if it exists, “NO” otherwise

```

for all  $q \in Q$  do
   $V[q] \leftarrow \emptyset$ , compute  $pref_k(L_q)$ , and compute  $P(q, w)$  for all  $w \in pref_k(L_q)$ 

while ( $V$  is updated) do
  for all  $q \in Q$  and  $a_1 \dots a_k \in pref_k(L_q) \setminus V[q]$  do

    if  $P(q, a_1 \dots a_k) = \emptyset$  then
      append  $a_1 \dots a_k$  to  $V[q]$  // (*)
    else
      if ( $\forall p \in P(q, a_1 \dots a_k) : a_2 \dots a_k \Sigma \cap V[p] \cap pref_k(L_p) \neq \emptyset$ ) then
        append  $a_1 \dots a_k$  to  $V[q]$ 

if  $V[q_0] \neq \emptyset$  then
  print “NO”
else
  print “YES”

  for all  $q \in Q$  and  $w \in \Sigma^{<k}$  do
     $T[q, w] = NIL$ 

   $construct(q_0, pref_k(L_{q_0}))$ 
   $extend(T)$ 

  return  $T$ 

```

□

definition of  $construct(q, W)$

```

for all  $a_1 \dots a_k \in W$  do

  if  $T[q, a_1 \dots a_k] = NIL$  then

    choose  $p \in P(q, a_1 \dots a_k)$  s.t.  $a_2 \dots a_k \Sigma \cap pref_k(L_p) \cap V[p] = \emptyset$ 

     $T[q, a_1 \dots a_k] \leftarrow p$ ,  $W' \leftarrow \{a_2 \dots a_k b / a_2 \dots a_k b \in pref_k(L_p)\}$  // (**)

     $construct(p, W')$ 

```

□

definition of  $extend(T)$

```

if  $k > 1$  then
  for all states  $q \in Q$  reachable in  $T$  do

    for all  $w \in L_q \cap \Sigma^{<k}$  do

      find a successful path  $c$  in  $M$  starting with  $q$  and labeled with  $w$ 

      assign to  $T$  values such that the  $k$ -delegator will follow the path  $c$ , once being
      in state  $q$  and having  $w$  in its buffer.

```

□



2. for every state  $p \in P(q, w)$  there exists  $b_p \in \Sigma$  such that  $a_2 \dots a_k b_p \in \text{pref}(L_p)$  and  $(p, a_2 \dots a_k b_p)$  is known to be forbidden.

We denote by  $F_q$  the set of all forbidden words for  $q$ .

*Remark 6.* Let  $q$  be a state in  $M$  and  $a_1 \dots a_k \notin F_q$ . There exists  $p \in P(q, a_1 \dots a_k)$  such that

$$\forall a_2 \dots a_k b \in \text{pref}(L_p) : a_2 \dots a_k b \notin F_p .$$

The set of all state  $p$  verifying this condition will be denoted by  $C(q, a_1 \dots a_k)$ , called the set of chosen states for  $(q, a_1 \dots a_k)$ . Convention-wise, if  $w \in F_p$  then  $C(p, w) = \emptyset$ .

**Lemma 10.** *At the end of the “while-loop” of Algorithm 1 we have  $V[q] = F_q \cap \Sigma^k$ , for all states  $q$  of  $M$ .*

*Proof.* We first notice that Definition 10 establishes recursively that a word is forbidden based on forbidden words of same length (it is “length aware”). We also notice that  $B_q \cap \Sigma^k \subseteq V[q]$ . Indeed, if  $a_1 \dots a_k \in B_q$  then  $P(q, a_1 \dots a_k) = \emptyset$  and  $a_1 \dots a_k$  is among the first words added to  $V[q]$  at the step denoted by (\*). Then it suffices to observe that the test

$$\text{“if } (\forall p \in P(q, a_1 \dots a_k) : a_2 \dots a_k \Sigma \cap V[p] \cap \text{pref}_k(L_p) \neq \emptyset) \text{ then”}$$

used for updating  $V[q]$  checks whether condition 2 of Definition 10 is satisfied.  $\square$

**Lemma 11.** *If the start state  $q_0$  of  $M$  verifies  $F_{q_0} \cap \text{pref}_k(L) = \emptyset$  then  $M$  has a  $k$ -delegator and Algorithm 1 terminates with an “YES” answer and returns a  $k$ -delegator.*

*Proof.* If  $F_{q_0} \cap \text{pref}_k(L) = \emptyset$ , then the algorithm prints “YES” since the test  $V[q_0] \neq \emptyset$  fails as a consequence of Lemma 10. It remains to prove that the procedures *construct()* and *extend()* deliver a delegator. We first make the point that the recursive call to *construct*( $q, W$ ) always verifies  $W \subseteq \text{pref}_k(L_q) \setminus F_q$ . This is true for  $q_0$  and it holds for subsequent calls to *construct*( $p, W'$ ) by virtue of the code lines:

$$\begin{aligned} &\text{choose } p \in P(q, a_1 \dots a_k) \text{ s.t. } a_2 \dots a_k \Sigma \cap \text{pref}_k(L_p) \cap V[p] = \emptyset \\ &T[q, a_1 \dots a_k] \leftarrow p, W' \leftarrow \{a_2 \dots a_k b / a_2 \dots a_k b \in \text{pref}_k(L_p)\} \end{aligned}$$

which ensure that  $W' \cap V[p] = \emptyset$ . By Remark 6,  $p$  is chosen such that  $p \in C(q, a_1 \dots a_k)$ .

It is clear that the recursive call to *construct* will end in a finite number of steps, due to the finiteness of  $T$  and to the fact that each subsequent call is preceded by filling an empty(*NIL*) cell of  $T$ . It remains to prove that at the end of Algorithm 1,  $T$  provides indeed a  $k$ -delegator.  $T$  represents the transition table of a  $k$ -lookahead DFA  $A$ , since each cell of  $T$  stores at most one state. We give an informal reason for why  $L(M) = L(A)$ . It is clear that  $L(M) \supseteq L(A)$ . If  $a_1 \dots a_n \in L(M)$  with  $n < k$ , then by definition of procedure *extend* it follows that  $a_1 \dots a_n \in L(A)$ . When  $n \geq k$ , we make the observation (which can be proven by induction) that there is a deterministic computation in  $A$  labeled  $a_1 \dots a_n$ . In order to show that this computation is successful, we notice that after scanning the first  $n - k$  symbols,  $A$  will have in its buffer the word  $a_{n-k+1} \dots a_n$  and will be in a state  $q$  such that  $a_{n-k+1} \dots a_n \notin F_q$ . After scanning another input symbol,  $A$  will be in a state  $p$ , with  $a_{n-k+2} \dots a_n$  in its buffer and  $a_{n-k+2} \dots a_n \in L_p$ . Then, yet again by definition of *extend*,  $A$  will finish the scanning in a final state.  $\square$

**Lemma 12.** *If  $M$  has a  $k$ -delegator, then the start state  $q_0$  of  $M$  verifies  $F_{q_0} \cap \text{pref}_k(L) = \emptyset$  and Algorithm 1 terminates with an “YES” answer and returns a  $k$ -delegator.*

*Proof.* Assume  $f : Q \times \Sigma^{\leq k} \rightarrow Q$  is a delegator for  $M$ . We first prove the following:

*Claim.* With the previous notations, the following implication holds:

$$f(p, w) \neq \emptyset \Rightarrow w \notin F_p .$$

Suppose that  $f$  contradicts the claim for some instance of  $p$  and  $w = a_1 \dots a_k$ , hence  $f(p, a_1 \dots a_k) = p_1$  and  $a_1 \dots a_k \in F_p$ . Since  $f$  is a trim delegator, we have the following sequence:

$$\begin{aligned} f(p, a_1 \dots a_k) = p_1, \quad a_1 \dots a_k \in F_p &\Rightarrow \\ \Rightarrow \exists b_1 : a_2 \dots a_k b_1 \in F_{p_1} \end{aligned}$$

$$\begin{aligned} f(p_1, a_2 \dots a_k b_1) = p_2, \quad a_2 \dots a_k b_1 \in F_{p_1} &\Rightarrow \\ \Rightarrow \exists b_2 : a_3 \dots a_k b_1 b_2 \in F_{p_2} \end{aligned}$$

...

$$\begin{aligned} f(p_{n-1}, a_n) = p_n, \quad a_n \in F_{p_{n-1}} &\Rightarrow \\ \Rightarrow \exists b_n : a_n b_n \in F_{p_n} \end{aligned}$$

...

Notice that by the recursive definition of forbidden words (Definition 10), there exists a choice for the letters  $b_1, \dots, b_n, \dots$  such that in the above sequence there exists a step  $n$  for which  $F_{p_n} = B_{p_n}$ . Observe that there may also exist cycles in this sequence, that is, pairs  $(p, v)$  which are repeated. However, by a proper choice of  $b_1, \dots, b_n, \dots$  we can enforce that the situation  $n : F_{p_n} = B_{p_n}$  appears before any repetition, fact ensured by Definition 10. Now it suffices to notice that  $F_{p_n} = B_{p_n}$  contradicts Corollary 3 by the fact that  $f(p_{n-1}, av) = p_n$  and there exists  $b_n : vb_n \in B_{p_n}$ .

We now use the proven claim as following: since  $f$  is defined in all  $(q_0, w)$  with  $w \in \text{pref}_k(L)$ , we have  $w \notin F_{q_0}$  for all  $w \in \text{pref}_k(L)$ , hence  $F_{q_0} \cap \text{pref}_k(L) = \emptyset$ . The fact that the algorithm terminates with an “YES” answer and it returns a  $k$ -delegator can now be proven similar to the proof of Lemma 11.  $\square$

**Corollary 6.** *If no  $k$ -delegator exists for  $M$  then Algorithm 1 answers “NO”.*

*Proof.* Assume by contradiction that the Algorithm answers “YES”. Then by Lemma 10 we have  $F_{q_0} \cap \Sigma^k = \emptyset$ , and by Lemma 11 the Algorithm returns a  $k$ -delegator, which contradicts that such delegator does not exist.  $\square$

The previous two lemmas prove more than the correctness of Algorithm 1, namely:

**Corollary 7.** *There exists a  $k$ -delegator for  $M$  if and only if*

$$F_{q_0} \cap \text{pref}_k(L) = \emptyset .$$

Consequently, we give the following characterization of NFA delegation:

**Theorem 6.** *There exists a delegator for  $M$  if and only if*

$$|F_{q_0}| < \aleph_0 .$$

*Proof.* The “if” part is straightforward:  $|F_{q_0}| < \aleph_0$  implies that for a  $k$  large enough we have  $F_{q_0} \cap \Sigma^k = \emptyset$ , and by the virtue of Lemma 11 there exists a  $k$ -delegator for  $M$ .

For the “only if” part, we know that there exists  $k$  such that  $M$  has a  $k$ -delegator and, by Corollary 7, that  $F_{q_0} \cap \text{pref}_k(L) = \emptyset$ . But having a  $k$ -delegator implies that for any  $l > k$  there exists an  $l$ -delegator for  $M$ . Then,  $F_{q_0} \cap \text{pref}_{\geq k}(L) = \emptyset$ , and since  $F_{q_0} \subseteq \text{pref}(L)$ , it follows that  $|F_{q_0}| < \aleph_0$ .  $\square$

It is not hard to see that there is an exponential space algorithm to determine membership in  $F_q$  (for any  $q$ ). The reason is as follows: From the foregoing discussion, it is clear that there is a PSPACE algorithm to determine if a string

$w$  is  $q$ -blind. It is also clear that it can be determined in PSPACE the set of states in  $P(q, w)$ . Now, we will describe an algorithm to determine membership of a string  $w_1w_2\dots w_k$  in  $F_q$ .  $w_1w_2\dots w_k$  is in  $F_q$  if and only if condition (1) or (2) is true. (1) can be checked in PSPACE. To check (2), we can create a table (as in memorized version of dynamic programming algorithm) that corresponds to various instances of the form  $(p, x_1x_2\dots x_k)$ . When the decision about an instance is reached, the table entry is filled (with ‘yes’ or ‘no’). When a new instance needs to be solved, the table is checked to see if the decision is already reached. It is clear that the total space of this algorithm is dominated by the table required to maintain the solutions of various instances and hence the resulting algorithm is an exponential space algorithm. Thus,  $F_q$  is recursive.

If we can show that  $F_{q_0}$  is regular or context-free, then clearly, the decidability of Problem 3 will follow since finiteness problem is decidable for both these classes. We do not know if the former is true.

## 5 Conclusion and Future Work

In this paper we have addressed the question of whether a NFA can be simulated deterministically using only its states and transitions, by taking advantage of reading ahead a fixed number of input symbols. This problem complements the extensive prior work on methods of simulating nondeterminism by using exponentially augmented state sets. We have provided a characterization of when this is possible, and have presented an efficient algorithm to determine when such a simulation is possible in restricted cases.

The main problem that remains open is the decidability of Problem 3 for general NFA’s. We believe that this problem is decidable, and Theorem 6 may provide a direction to establish such a result. The complexity of Problems 2 and 3 (in the case of unambiguous NFA’s) have not been completely resolved. Specifically, are the problems complete for co-NP and PSPACE respectively?

## References

- [Berardi et al., 2003] Berardi, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., and Mecella, M. (2003). Automatic Composition of e-Services that Export their Behavior. In Orłowska, E., Papzoglou, M., Weerawarana, S., and Yang, J., editors, *ICSOC 2003*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58. Springer.
- [Dang et al., 2004] Dang, Z., Ibarra, O. H., and Su, J. (2004). Composability of Infinite-State Activity Automata. In Fleischer, R. and Trippen, G., editors, *ISAAC 2004*, volume 3341 of *Lecture Notes in Computer Science*, pages 377–388. Springer.
- [Gerede et al., 2004] Gerede, C., Hull, R., Ibarra, O., and Su, J. (2004). Automated Composition of e-Services: Lookaheads. In [Traverso and Weerawarana, 2004], pages 252–262.

- [Gerede et al., 2005] Gerede, C., Ibarra, O., Ravikumar, B., and Su, J. (2005). On-line and Ad-hoc Minimum Cost Delegation in e-Service Composition. In *IEEE SCC*, pages 103–112. IEEE Computer Society.
- [Hopcroft and Ullman, 1979] Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation - 1st edition*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA.
- [Hull and Su, 2004] Hull, R. and Su, J. (2004). Tools for Design of Composite Web Services. In Weikum, G., König, A. C., and Deßloch, S., editors, *SIGMOD 2004*, pages 958–961. ACM Press.
- [Hunt et al., 1976] Hunt, H., Rosenkrantz, D. J., and Szymanski, T. G. (1976). On the Equivalence, Containment, and Covering Problems for the Regular and Context-Free Languages. *Journal of Computer and Systems Sciences*, 12(2):222–268.
- [Ibarra et al., 2006] Ibarra, O., Ravikumar, B., and Gerede, C. E. (2006). Quality-Aware Service Delegation in Automated Web Service Composition. To appear in *Journal of Automata, Languages and Combinatorics*.
- [Mecella and Giacomo, 2004] Mecella, M. and Giacomo, G. D. (2004). Service Composition: Technologies, Methods and Tools for Synthesis and Orchestration of Composite Services and Processes (tutorial). In [Traverso and Weerawarana, 2004].
- [Rozenberg and Salomaa, 1997] Rozenberg, G. and Salomaa, A. (1997). *Handbook of Formal Languages*. Springer-Verlag, Berlin Heidelberg New York.
- [Stearns and Hunt, 1985] Stearns, R. and Hunt, H. (1985). On the Equivalence and Containment Problems for Unambiguous Regular Expressions, Regular Grammars and Finite Automata. *SIAM Journal on Computing*, 14(3):598–611.
- [Traverso and Weerawarana, 2004] Traverso, P. and Weerawarana, S., editors (2004). *Service Oriented Computing, 2nd International Conference, ICSOC 2004, New York City, NY, USA, November 15-18, 2004, Proceedings*. ACM Press.
- [Yu, 1997] Yu, S. (1997). Regular Languages. In [Rozenberg and Salomaa, 1997], 1:41–110.