Deterministic Simulation of a NFA with k–Symbol Lookahead

Bala Ravikumar[1] and Nicolae Santean[2]

[1] Department of Computer Science, Sonoma State University
Rohnert Park, CA 94928, USA
[2] School of Computer Science, University of Waterloo
Waterloo, ON, 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 [1]. This paradigm is based on so–called e–services composition, concept introduced by [1] and recently studied extensively by a number of scientists: [7], [6], [8], [3], [4], etc.

k–Delegators were first introduced informally in [2] in the study of e–services composability, which involves automatically combining the services of individual agents to accomplish a larger task. In the same paper it was established that the existence of k–delegators is decidable for a given k. However, the complexity of this problem was not addressed. Moreover, the problem of deciding the existence

Jan van Leeuwen et al. (Eds.): SOFSEM 2007, LNCS 4362, pp. 488–497, 2007.

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. Only a sketch of the proof of some results appear in the main text of the paper. Detailed proofs and further explanations on the matters in discussion can be found in the technical report [9], available on the web.

2 The Delegation Problem

In the following we assume known basic notions of automata theory (see, for example, [5] and [12]). Notation–wise, an NFA is a tuple $M = (Q, \Sigma, \delta, q_0, F)$ with Q a finite set of states, Σ 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: i.e., it is `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 δ 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 ε the empty word, by Σ^k the set of all words of length k over Σ (and by $\Sigma^{\leq k}$ the set of all words of length at most k), by $pref(L)$ the set of all prefixes of words in a language L, and by $pref_k(L)$ the set $pref(L) \cap \Sigma^k$.

By a DFA with a k–lookahead buffer we understand a DFA $A = (Q, \Sigma, f, q_0, F)$ with $f : Q \times \Sigma^{\leq k} \to Q$, 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: the buffer content can be viewed as part of automaton's internal state.

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 next example shows that there are NFA's that do not have k–delegators for any k.

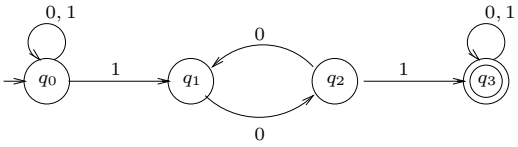

Fig. 1. An NFA which has no k–delegator for any k

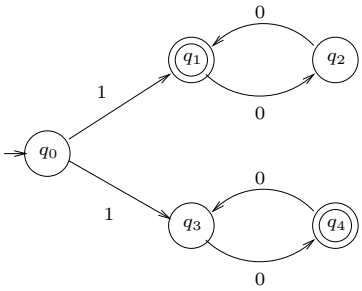

Fig. 2. An unambiguous NFA which has no k–delegator for any k

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. M does not have a k–delegator for any positive integer k. The NFA in Figure 2 is an `unambiguous NFA` (i.e., any word is the label of at most one successful computation), and yet, it has no delegator.

Every regular language L is accepted by a NFA that has a 1–delegator, namely a DFA for L. Nevertheless, there 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 langauge.*

(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 these two classes of regular languages coincide.

Theorem 1. *The following statements are equivalent:*

1. *L is finite.*
2. *L has a strong delegator.*
3. *L has a weak delegator.*

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a trim NFA and $q \in Q$, $a_1 \dots a_k \in \Sigma^k$ such that $\delta(q, a_1) = \{q_1, \dots, q_t\}$ with $t > 1$ (q has nondeterministic transitions on input a_1). Notation–wise, by L_q we denote the language accepted by M if q is chosen as the start state of M (with no other change to its definition).

Definition 3. *With the above notations, we say that q is $a_1 \dots a_k$-*`blind` *if $\delta(q, a_1) = \{q_1, \dots, q_t\}$, $t > 1$, and for all $i \in \{1, \dots, t\}$ the following inequality holds:*

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

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

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. *We denote the* `blindness` *of q (or, the language of blind words for q) as being the language $B_q = \{w \in \Sigma^* / q \text{ is } w\text{–blind}\}$.*

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

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. Consequently, the "interesting" states are those with infinite blindness.

Proposition 1. *The following properties hold:*

1. *For any state q, B_q is prefix–closed, except for the empty word.*
2. *If a NFA M has all states finitely blind, then it accepts a lookahead delegator.*
3. *If a state q of a NFA M is k–blind, $k \geq 2$, then it is l–blind for all $l \in \{1, \dots, k-1\}$.*
4. *If the initial state of a NFA M is infinitely blind then M has no k–lookahead delegator for any integer k.*

3 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 delegator, "`NO`" otherwise.

In the following we first tackle the special case when the input NFA is unambiguous, after which we deal with the general case of NFA's that may be ambiguous.

Definition 5. *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 x.*

Proposition 2. *The following results hold for unambiguous NFA:*

1. *If M is unambiguous, then for every state q and for every string $w \in pref(L_q)$, the pair (q, w) is crucial.*
2. *Let M be an unambiguous NFA, q be a state of M and $w \in \Sigma^k$ for some $k \geq 1$. If (q, w) is crucial for M and q is w–blind, then M cannot have a k–delegator.*
3. *An unambiguous NFA M has a k–delegator* iff *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. Then, M has a delegator if and only if B_q is finite for every state q of M.*
4. *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.

Remark 1. In the following we use the fact that is decidable in polynomial time whether a given NFA is ambiguous or not. The following nondeterministic algorithm which uses LOGSPACE tests if an NFA is ambiguous. The input tape of the Turing machine (which implements the nondeterministic algorithm) contains the encoding of a NFA M. The machine guesses a string w (over the alphabet of M) one symbol at a time, and executes two different computations of M on the string w. If both computations reach accepting states, then M is ambiguous. Since NLOGSPACE is contained in P, the conclusion follows shortly.

Theorem 3. *When the input NFA is unambiguous, Problem 1 is in P, Problem 2 is in co–NP, and Problem 3 is in PSPACE.*

Proof. (sketch) The input to the problem 1 is a (trim) unambiguous NFA $M = (Q, \Sigma, \delta, q_0, F)$, and k is a fixed constant that is not part of the input. By Proposition 2, 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 $\delta(q, a) = \{q_1, q_2, ..., q_t\}$. Recall that $w = av_2...v_k$ is in B_q if and only if for each i, the following condition holds:

$$\left(\bigcup_{j\in\{1,2,\ldots,t\},\ j\neq i} (v_2v_3...v_k)^{-1}L_{q_j} \right) \ \setminus \ (v_2v_3...v_k)^{-1}L_{q_i} \neq \emptyset \ .$$

Let the language on the left–side of the above expression be denoted $B_{q,a,i}$. For each pair (q,w) where $w = v_1v_2...v_k$, we check whether $w \notin B_{q,v_1,i}$ as follows. We compute the sets of states $R_1 = \{p/\ p$ is reachable from q_i on $v_2v_3...v_k\}$, and $R_2 = \{p/\ p$ is reachable from q_j for some $j \neq i$ on $v_2...v_k\}$. Note that for a given pair (q,w), all these sets can be constructed in time polynomial in $|M|$, and use (4) of Proposition 2 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 $O(2^kP(|M|))$ for some polynomial P and hence for a fixed k, the algorithm runs in polynomial time. Next, we consider Problem 2. Now, k is part of the input (in unary). The algorithm guesses a pair $(q, v_1 \ldots v_k)$ for some $q \in Q$ and some string $w = v_1 \ldots v_k \in \Sigma^k$ 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 $O(k|M|)$. The rest of the details are the same as for Problem 1. To show that the Problem 3 can be solved in PSPACE, we use the ideas described above together with the upper–bound established in Theorem 2. □

In the following we deal with the general case, namely the case where M can be ambiguous.

Theorem 4. *Problem 1 for the general case is PSPACE–complete (the hardness holds for every fixed $k = 1, 2, 3, \ldots$). Consequently, Problems 2 and 3 are PSPACE–hard.*

Next, we describe an algorithm for Problem 1 in the general case, significantly better than "brute force" approach (i.e., exhaustive search by generating all imaginable k–lookahead delegators for a NFA M, and for each checking the equivalence with M) mentioned in [2] . To improve algorithm's formalism, we give the following definition.

Definition 6. *Let q be a state in $M, w = a_1 \ldots a_k$ and $\delta(q, a_1, \ldots a_k) = \{q_1, \ldots q_t\}$, $t \geq 1$. A state q_i is* potential *for (q,w) if it verifies:*

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

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

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 that $P(q, w)$ is obviously computable for any q and w.

Algorithm 1, detailed at page 495, 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 p of M, a set of words $w \in pref_k(L_p)$ 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*".

Definition 7. *For $w \in pref(L_q)$, 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–blind;*
2. *for every state $p \in P(q, w)$ there exists $b_p \in \Sigma$ such that $a_2 \dots a_k b_p \in pref(L_p)$ 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.

Proposition 3. *The following results hold and support the correctness of Algorithm 1:*

1. *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.*
2. *The start state q_0 of M verifies $F_{q_0} \cap pref_k(L) = \emptyset$ iff M has a k–delegator. Under these conditions, Algorithm 1 terminates with a "`YES`" answer.*
3. *If M has no k–delegator then Algorithm 1 terminates with a "`NO`" answer.*
4. *If the algorithm responds "`YES`", then it returns a `trim` k–delegator, that is, a delegator whose all "predictions" (or, "delegations") are used in some successful computations.*

Consequently, we give the following compact characterization of delegation:

Theorem 5. *There exists a delegator for M if and only if F_{q_0} is finite.*

Remark 2. It is not hard to see that there is an exponential space algorithm to determine the membership in F_q (for any q). Indeed, from the foregoing discussion, it is clear that there exists 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 $w_1 w_2 ... w_k$ in F_q. $w_1 w_2 ... w_k$ is in F_q if and only if condition (1) or (2) of Definition 7 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

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 $w \in \Sigma^{\leq k}$ **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. $a_2 \dots a_k \Sigma \cap pref_k(L_p) \cap V[p] = \emptyset$

    $T[q, a_1 \dots a_k] \leftarrow p$, $W' \leftarrow \{a2 \dots a_k b / a_2 \dots a_k b \in pref_k(L_p)\}$ // (**)

    $construct(p, W')$

□

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.

□

corresponds to various instances of the form $(p, x_1 x_2 ... x_k)$. 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.

4 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 algorithms to determine when such a simulation is possible in restricted cases.

The problem that remains unsolved is the decidability of Problem 3 for general NFA's. We believe that this problem is decidable, and Theorem 5 may provide a direction to establish such a result. Indeed, 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; however, presently there are promising efforts to solve the general problem, and we believe that a solution will be given sooner rather than later.

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

1. Berardi, D., Calvanese, D., Giacomo, G.D., Lenzerini, M., and Mecella, M.: Automatic Composition of e-Services that Export their Behavior. In E. Orlowska, M. Papzoglou, S. Weerawarana, and J. Yang (eds), ICSOC 2003,Springer, Lecture Notes in Computer Science **2910** (2003) 43–58.
2. Dang, Z., Ibarra, O.H., and Su, J.:. Composability of Infinite-State Activity Automata. In Rudolf Fleischer and Gerhard Trippen (eds), ISAAC 2004, Springer, Lecture Notes in Computer Science **3341** (2004) 377–388
3. Gerede, C.E., Hull, R., Ibarra, O.H., and Su, J.: Automated Composition of e-Services: Lookaheads. In Traverso and Weerawarana [11], 252–262
4. Gerede, C.E., Ibarra, O.H., Ravikumar, B., and Su, J.: On-Line and Ad-Hoc Minimum Cost Delegation in e-Service Composition. In C. K. Chang and L.–J. Zhang (eds), IEEE SCC, IEEE Computer Society (2005) 103–112
5. Hopcroft, J.E., Motwani, R., and Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation - 3rd edition. Addison-Wesley Longman Publishing Co. Inc., Boston, MA (2006)
6. Hull, R. and Su, J.: Tools for Design of Composite Web Services. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch (eds), SIGMOD 2004, ACM Press (2004) 958–961
7. Ibarra, O.H., Ravikumar, B., and Gerede, C.E.: Quality-Aware Service Delegation in Automated Web Service Composition. To appear in Journal of Automata, Languages and Combinatorics (2006)
8. Mecella, M. and Giacomo, G.D.: Service Composition: Technologies, Methods and Tools for Synthesis and Orchestration of Composite Services and Processes (tutorial). In Traverso and Weerawarana [11].

9. Ravikumar, B. and Santean, N.: Deterministic Simulation of a NFA with k-Symbol Lookahead. Technical Report CS-2006-28, University of Waterloo (August 2006) Also available as
`http://www.cs.uwaterloo.ca/research/tr/2006/CS-2006-28.pdf`
10. Rozenberg, G. and Salomaa A.: Handbook of Formal Languages. Springer-Verlag, Berlin Heidelberg New York (1997)
11. Traverso, P. and Weerawarana, S. (eds): Service Oriented Computing. 2nd International Conference, Proceedings, ICSOC 2004, ACM Press, New York City, NY, USA, (November 15-18, 2004)
12. Yu, S.: Regular Languages. In [10], **1** (1997) 41–110