Exchanging Intensional XML Data

Summer Internship Report

by

Shobhit Raj Mathur

Department of Computer Science and Engineering
Indian Institute of Technology
Bombay

under the guidance of

Tova Milo and Omar Benjelloun

INRIA-FUTURS
Project Gemo
Orsay CEDEX
France

July 23, 2003

Abstract

This report summarizes my discussion with Tova Milo and Omar Benjelloun on the modifications which can be made to the algorithm for rewriting intensional XML Data as described in [1]. The primary motivation is to distribute the execution of the algorithm, and as a result balance the computation and network load among the peers. This report also contains an algorithm to build A_X as described in [1] incrementally, which can be used to find the minimal depth rewriting.

In Chapter 1 we discuss the proposed algorithm for distributed rewriting. An idea to implement distributed logging is described in Chapter 2. Chapter 3 contains an algorithm to find the minimum depth of rewriting required. The problem of identifying independent substrings in the given input word is discussed in Chapter 4. The algorithms proposed have been followed by an example illustration in their respective chapters.

Acknowledgement

July 23, 2003

I would like to thank Tova Milo and Omar Benjelloun for the consistent directions they fed into my work. Their constant motivation and encouragement proved to be extremely useful.

Shobhit Raj Mathur

Contents

Chapter 1

Distributed Rewriting

1.1 Algorithm

Notation:

- Word representing the input document : w
- Target language : R
- function output types : $R_{f_1} \dots R_{f_n}$ where $R_{f_i} = \tau_{out}(f_i)$
- Alphabet set : α

1. A_X **Construction :**

   (a) **Preprocessing**

      i. **Rename duplicate functions :** Traverse the word and for each function f_i which is repeated in the string w rename the repeated occurrence to f_k (where f_k not in α) and add the letter f_k to α. Similarly rename the functions in each R_{f_i}.

      ii. **Convert w to a string of functions:** Replace each substring $f_i.a_1.a_2.f \dots .a_n.f_j$ (where $a_1.\dots.a_n$ are non function letters in α) by $f_i.f_j$ and append the string $a_1.a_2.\dots.a_n$ to $\tau_{out}(f_i)$. Also replace any string of non function letters followed by a function (like $b_1.\dots.b_n.f_1$) in the beginning of w by $f_0.f_1$ (where f_0 not in α) and $R_{f_n} := \tau_{out}(f_0) := b_1.\dots.b_n$. Add f_0 to α.
      At the end of this step w is converted to a string of functions $f_0.\dots.f_n$
      example illustrating the previous 2 steps:

$$w = a.b.f_1.c.d.f_2.e.f_2.r.f_1.s$$

$$\tau_{out}(f_1) = x.y$$

$$\tau_{out}(f_2) = p.q$$

      We modify the word w into word

$$w\prime = f_0.f_1.f_2.f_3.f_4$$

$$\tau_{out}(f_0) = a.b$$
$$\tau_{out}(f_1) = x.y.c.d$$
$$\tau_{out}(f_2) = p.q$$
$$\tau_{out}(f_3) = p.q.r$$
$$\tau_{out}(f_4) = x.y.s$$

iii. **Function result delimiters:** For each function f_i in α add ϵ_{f_i} and $\epsilon_{f_i'}$ to α .

(b) **Construct the following automata :**

i. An automaton A_w accepting w as a single word.

ii. Automata A_{f_i}, $i = 1 \ldots n$, each accepting the regular language R_{f_i} $(= \tau_{out}(f_i))$.

iii. An automaton $\bar{A}$ accepting the complement of the regular language R. The automaton should be deterministic and *complete*, namely each state has outgoing edges for all possible letters.

(c) Let $A^k_w := A_w$

(d) For j $= 1, \ldots, k$
Consider all the edges $e = (v, u)$ in A^k_w that are labeled by an (invokable) function name f_i, and were not treated in previous iterations. For each such edge:

i. Extend A^k_w by attaching a copy of the automaton A_{f_i}, with its initial state linked to v by ϵ_{f_i} move and the final state linked to u by $\epsilon_{f_i'}$ move.

ii. Denote v as a fork node (for the edge e).

iii. The *fork options* of v (for e) are e itself and the new outgoing ϵ_{f_i} edge.

(e) Construction the Cartesian product automaton $A_X = A^k_w \times \bar{A}$.

(f) The *fork nodes* and *fork options* in A_X reflect those of A^k_w:

i. The *fork nodes* $[q, p] \in A_X$ are those where q was a fork node in A^K_w.

ii. Similarly, a *fork option* in A_X consists of all edges originating from one fork option edge in A^k_w.

(g) Mark nodes in A_X as follows.

i. First mark all accepting states, (namely nodes $[q, p]$) where q and p are accepting states in A^k_w and $\bar{A}$ respectively.

ii. Then iteratively: mark regular (non fork) nodes if one of their outgoing edges points to a marked node; mark fork nodes if both their fork options (for some f_i) contain an edge that points to a marked node.

(h) **A left to right safe rewriting exists iff the initial state is not marked**
To obtain such a rewriting go to the next step. Else if the initial state is marked exit the algorithm.

2. **dependency analysis:**

For the rest of the analysis
$w := f_0. \ldots .f_n$
A_X refers to the marked cross product automaton

(a) **Identifying totally independent functions (level 1):** Since left to right safe rewriting is possible we attempt to find the set of functions I_i ($i = 1, \ldots$) which can be invoked in parallel. I_i is the set of functions which can be called independently at the i^{th} level. Initially $I_i = \phi$ for all i. Below we show how to build I_1. Same analysis is done in the next ietration for I_2 and so on.

i. The start node of A_X is the *fork node* corresponding to the function f_0. By looking at the marking of A_X we know whether we have to invoke f_0 or not. If we have to invoke f_0 we add it to I.

ii. For each i in $1 \ldots n$ we look at the *fork nodes* of f_{i+1} (they are pointed to by ϵ_{f_i}/s) and see whether the decision about f_{i+1} (whether to invoke it or not, by looking at the markings) is the same for all these *fork nodes.* In case it is the same, we know that f_{i+1} is independent of the output of all the other functions. Add f_{i+1} to I_1.

iii. Let I_1 be $\{f_{p_1}, f_{p_2}, \ldots, f_{p_m}\}$ where f_{p_i} is function to be called on peer i. Note that I_1 does not contain functions which we know will not be invoked.

3. **Calling functions in I:** To invoke the function f_{p_i} in I_1 we call a particular webservice *ws* on p_i and send a part of A_X (let it be called A_{p_i}) as a parameter to *ws*. A_{p_i} is constructed as follows:

   (a) Join all the fork node(s) corresponding to f_{p_i} to a new node n by an ϵ edge. n is the start node of A_{p_i}.

   (b) The terminal node(s) are those which contain an outward edge $\epsilon_{f_{p_i}}$/.

4. **Automaton pruning and renaming:** At p_i the following steps take place:

   (a) *ws* invokes f_{p_i} and based on the output it gets, it identifies the branch(es) b (emerging from the many *fork nodes of* f_{p_i}) in A_{p_i} which match the output.

   (b) In the selected branch(es) b if for each function g which occurs more than once, each instance of g is given a distinct name with the same signature as g.

   (c) Also, if in the selected branches there is a cycle containing a function, this cycle is *unfolded* into as many copies as would be required and the function is given a unique name in each opened cycle.

5. **base case:** When a peer p_i receives A_{p_i} and the selected branch(es) b do not contain any service calls, no more functions have to be called by this peer.

6. **recursion:** The same analysis which was done for the 1st level is repeated with A_X replaced by b.

7. **identifying the path:** As each peer p_i moves on the automaton, it *colors* the nodes traversed (i.e b).

8. **return value of *ws*** p_i *colors* the path traversed on b. It returns this *colored* A_{p_i} as a answer to the call on *ws*. Note that A_{p_i} may contain fewer nodes, as it was pruned to match the output of f_i.

9. **what to do with the returned automaton:** When the calling peer gets the result of its service call, the following steps are done:

(a) In the returned A_{p_i}, the ϵ edges and the node n which were added in the construction of A_{p_i} before calling *ws* on p_i are removed.

(b) The part of the automaton which was used in the construction of A_{p_i} (the part from the fork nodes of f_i to the nodes having outgoing $\epsilon_{f_{p_i}}\prime$ edges) is cut from the automaton and relaced with the above result.

(c) When all the replacement is done at this level, the modified branches in A_X starting with an uncolored node are pruned.

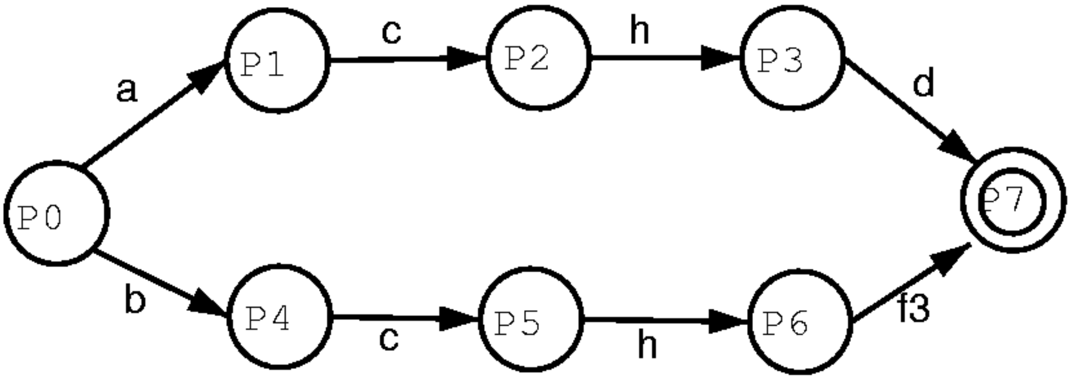
10. **next level:** After the 1st level functions get their results, the original automaton A_X changes due to these function calls. Let this changed automaton be called A_X^1 (automaton after the 1st level functions return their results). In A_X^1 analyse only the fork nodes (if any) at the end of *colored* nodes to identify I_2.

11. **when to stop?:** When the original peer (which started the recusrion process) does not find any more independent functions, we stop. i.e $I_i = \phi$.

1.2 Example

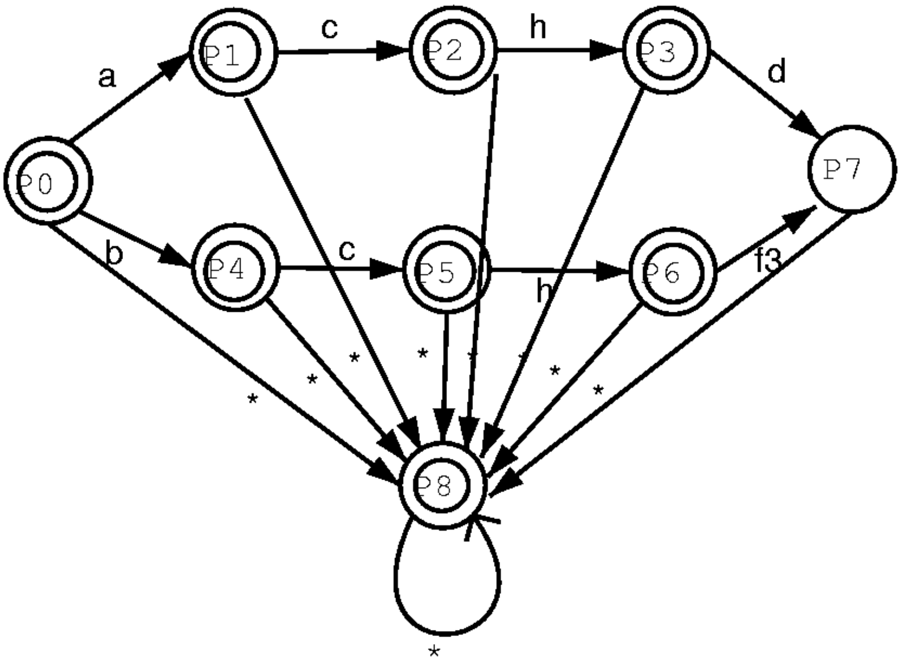
After preprocessing

- Word $w = f_1.f_2.f_3$
- Target Language $R = a.c.h.d|b.c.h.f_3$
- Function Output Types
  - $R_{f_1} = a|b$
  - $R_{f_2} = g.h$
  - $R_{f_3} = d$
  - $R_g = c$
  - $R_h = x$
- $\alpha = \{a, b, c, d, f_1, f_2, f_3, g, h, \epsilon_{f_1}, \epsilon_{f_1}\prime, \epsilon_{f_2}, \epsilon_{f_2}\prime, \epsilon_{f_3}, \epsilon_{f_3}\prime, \epsilon_g, \epsilon_g\prime, \epsilon_h, \epsilon_h\prime\}$



A

Figure 1.1: Automaton A for the target language



A′

Figure 1.2: $\bar{A}$

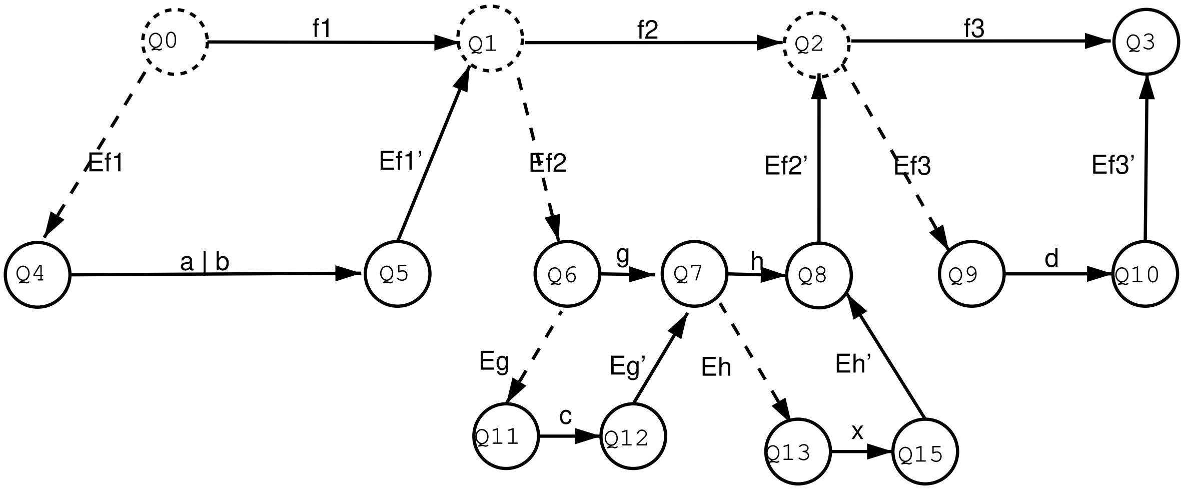


Figure 1.3: A_w^k

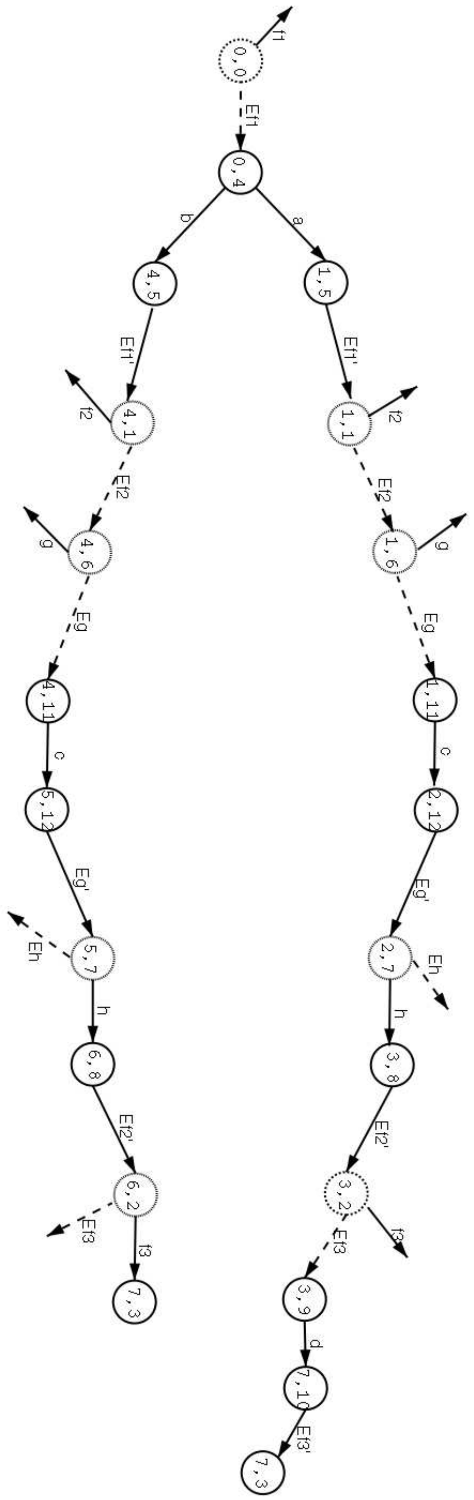


Figure 1.4: the cross product automaton A_X after pruning marked nodes

The rewriting process:
***coloring* is shown by dashed circles**

1. 1_{st} **level invocation :**
   $I_1 = \{f_1, f_2\}$

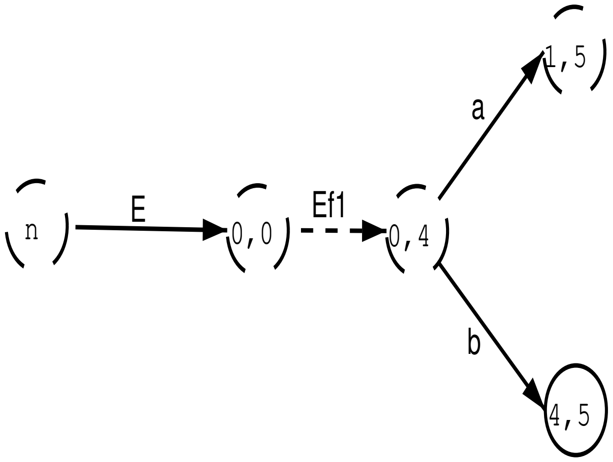


Figure 1.5: A_{p_1} before returning. This is the case when f_1 gives a as output

2. I_1 at $p_2 = \{g\}$.

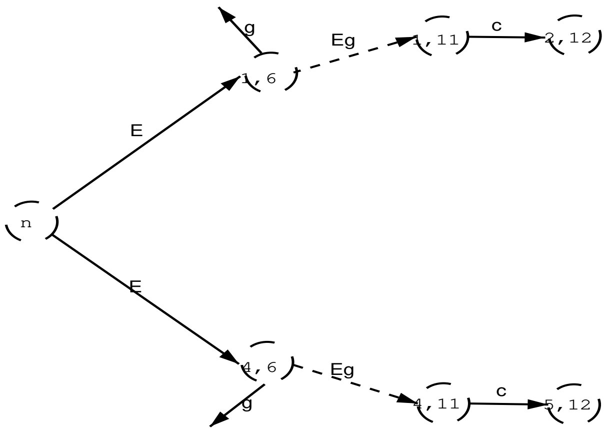


Figure 1.6: The automaton which the host on which g is called returns to p_2

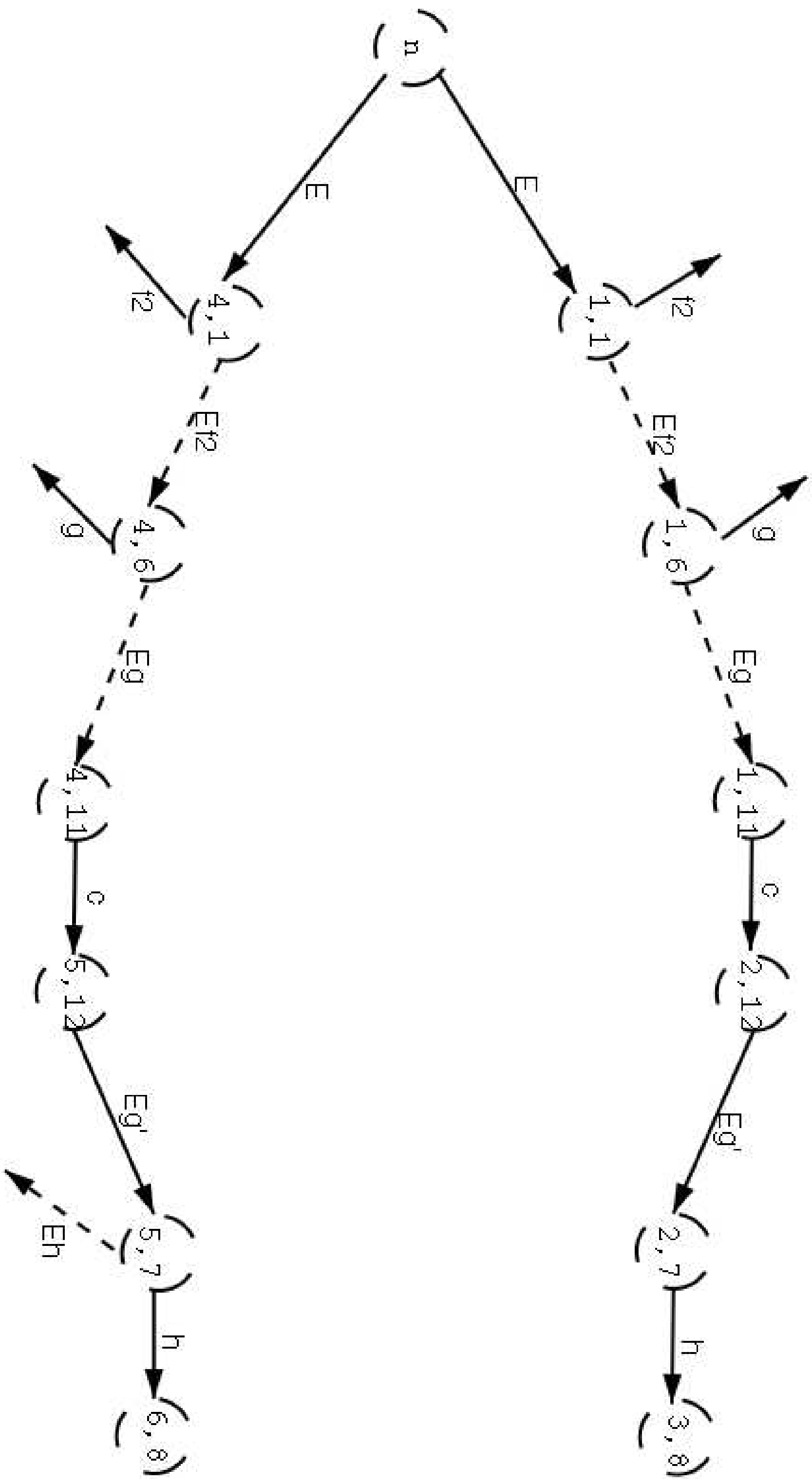


Figure 1.7: A_{p_2} before returning

3. **after receiving results of service calls and pruning:**

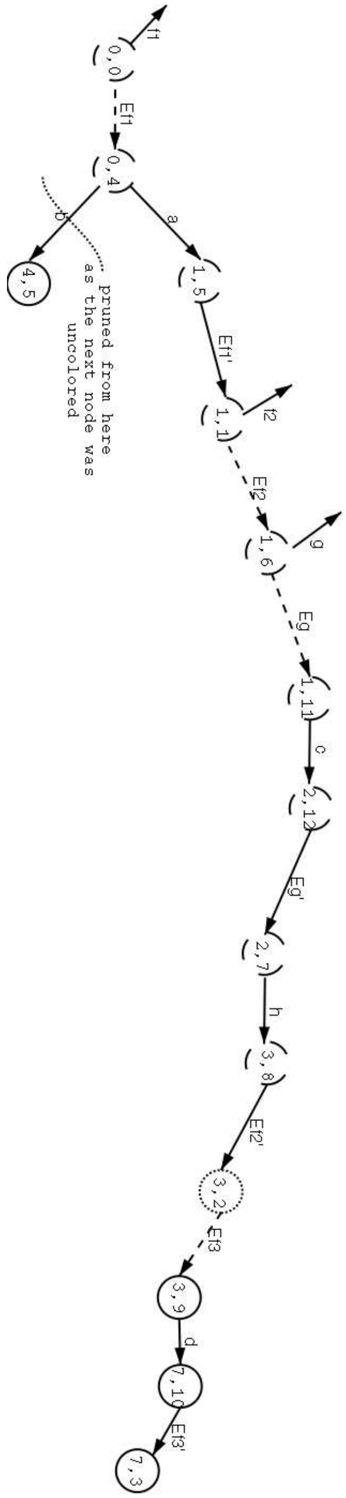


Figure 1.8: A_X^1

4. **at the end of the process:**

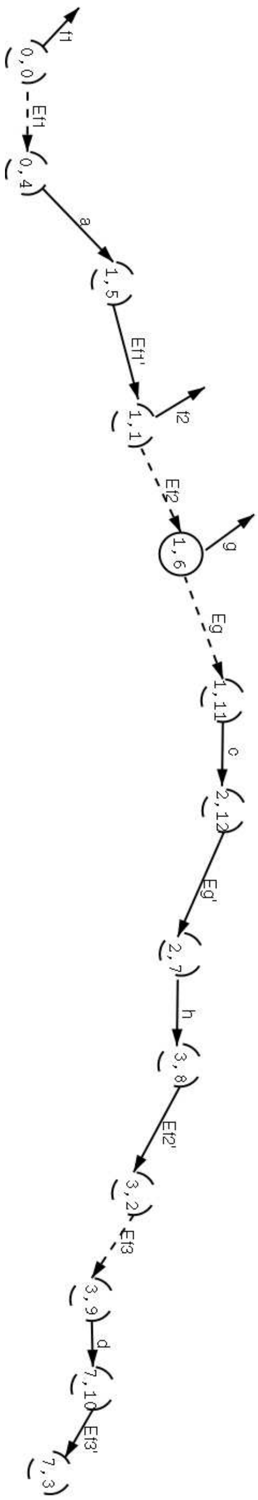


Figure 1.9: A_X^2

Chapter 2

Distributed Logging

In the preceding chapter we discussed the algorithm to distribute the rewriting process. When finally the clients request has been satisfied, the client/administrator may wish to know the rewriting done by each peer in the network to satisfy the request. This demands a mechanism to make sense out of the individual logs at each peer. In this chapter we propose a simple method based on *Request IDs* to do the above. This chapter is divided into 2 sections. The 1^{st} section describes how a service call is executed and the 2^{nd} section analyses the result returned.

2.1 Service Call Execution

2.1.1 Request ID

Suppose a client C calls a web service running on peer A, A generates an ID called the *Request ID* which will henceforth be mentioned as *RID*. *RID* is a combination of the time-stamp (which could be got by making a service call to a particular peer) and the peer on which the call was made, here A. In our discussion we assume it is $A : T$ where T is the time-stamp when the client's request begins to be processed. Note that *RID* is unique for each client request in the network.

2.1.2 Request Forwarding

Now A may need to call other peers $A_1, A_2, \ldots, A_n$ as demanded by the distributed rewriting algorithm. So A sends the *RID*, $A : T$ as computed above, to $A_1, A_2, \ldots, A_n$ as a parameter to the service call it makes on these peers. A_1 may need to call $A_{11}, A_{12}, \ldots, A_{1n}$ and so on. Each peer A_i, when it calls A_{ij} sends the *RID* as a parameter to the service call. The *RID* generated by A, i.e $A : T$ is used by all the peers. *RID* is generated only by the peer (here A) on which the client calls a webservice.

2.1.3 Local Log maintainance

Log has 2 parts:

1. **Rewriting log:** Each peer stores the log of the rewriting etc. it did corresponding the *RID* it received.
2. *sc* **log:** A_{ij} $j = 1, 2, \ldots, m$ returns to A_i along with the result a service call $sc_{A_{ij}}$ which can be used to query the log on A_{ij}. When A_i gets the results from all A_{ij} $j = 1, 2, \ldots, m$ it

makes an entry in its log-file as follows:

Request ID	Service calls
$A:T$	$sc_{A_{i1}}\ sc_{A_{i2}}\ sc_{A_{i3}}\ \ldots\ sc_{A_{im}}$

That is, service calls are stored with respect to their corresponding *RID*.

2.1.4 Working of sc_P

sc_P takes a *RID* and a schema s as parameters. It returns the *information* contained in the log at P corresponding to that *RID* appropriately rewritten to match the supplied schema s. s can specify whether it needs the rewriting log or the sc log or a combination of both, in which case some of the sc may have to be *rewritten*.

2.1.5 Return Value

Each peer A_i returns the result with a sc_{A_i} to the calling peer, after getting all the results from A_{ij} $j = 1, 2, \ldots, m$.
A returns the final result with sc_A and $A:T$ to C.

2.2 Result Analysis

If the client/administrator wishes to analyze the result. It invokes the service call sc_A that was returned by A with the *RID* $A:T$ and a schema s as parameters. A then queries its log to search for the service calls it made corresponding to the *RID* $A:T$. It then does the rewriting according to the schema s supplied.

It is interesting to note that s can specify the peers from which it needs the log and the schema which the result should satisfy. The same rewriting algorithm is then used to satisfy C's request.

Chapter 3

Finding the minimum k

3.1 Algorithm

This algorithm can be used to build the cross product automaton A_X incrementally from depth 0 to depth k, where k is minimum depth of rewriting required.
Assumptions: input word $w = f_0.f_1 \ldots f_n$
Notation used is same as before, but the A_X for i depth rewriting is denoted by A_X^i. At each iteration A_w^{i+1} and A_X^{i+1} $(i = 0, \ldots, k)$ will be constructed from A_w^i and A_X^i respectively.

Initially $i = 0$
Construct A_w^0 and A_X^0 $(= A_w^0 \times \bar{A})$ and do the marking on A_X^0

1. Check if the initial node of A_X^i is marked. If yes go the next step, else exit with success.

2. Construct A_w^{i+1} from A_w^i by doing the following:

   Consider all the edges $e = (v, u)$ in A_w^i that are labeled by an (invokable) function name f_m. For each such edge:

   (a) Extend A_w^i by attaching a copy of the automaton A_{f_m}, with its initial state linked to v by ϵ_{f_m} move and the final state linked to u by $\epsilon_{f_m}\prime$ move.

   (b) Denote v as a fork node (for the edge e).

   (c) The *fork options* of v (for e) are e itself and the new outgoing ϵ_{f_m} edge.

3. For each A_{f_m} added to A_w^i above do the following:
   Let the nodes contained in A_{f_m} be $q_{m_1} \ldots q_{m_n}$

   (a) Look at the node(s) in A_X^i which has an outgoing edge f_m. Let this node be $[p_{f_m}, q_{f_m}]$. Convert this node as a fork node for the function f_m. Add an ϵ_{f_m} transition from $[p_{f_m}, q_{f_m}]$ to $[p_{f_m}, q_{m_1}]$. From this node start adding new nodes to A_X^i with trasitions defined by the edges of A_{f_m}.

   (b) Check whether any new transitions can be made on the last of these newly added nodes. If so continue to do these transitions (and keep adding new nodes) till no more transitions can be made.

   (c) Do the marking on these new nodes using the marking of A_X^i.

4. Check the marking of the fork node $[p_{f_m}, q_{f_m}]$ of each function f_m. If it needs to be unmarked go to the next step, else go back to the step 2.

5. Consider all the nodes which have an outgoing edge to $[p_{f_m}, q_{f_m}]$. Unmark them (if they are not fork nodes, if they are fork nodes unmark them only if all the outgoing edges from this fork node go to unmarked nodes). Keep going backward till no more nodes can be unmarked.

6. This is A_X^{i+1}. Increment i.

7. If the initial node is unmarked then exit with success, else go back to the step 2.

3.2 example

- $w = f_1.f_2$
- $R = a.b.c.f_2$
- $\tau_{out}(f_1) = a.f_3$
- $\tau_{out}(f_3) = b.f_4$
- $\tau_{out}(f_4) = c$
- $\tau_{out}(f_2) = d$

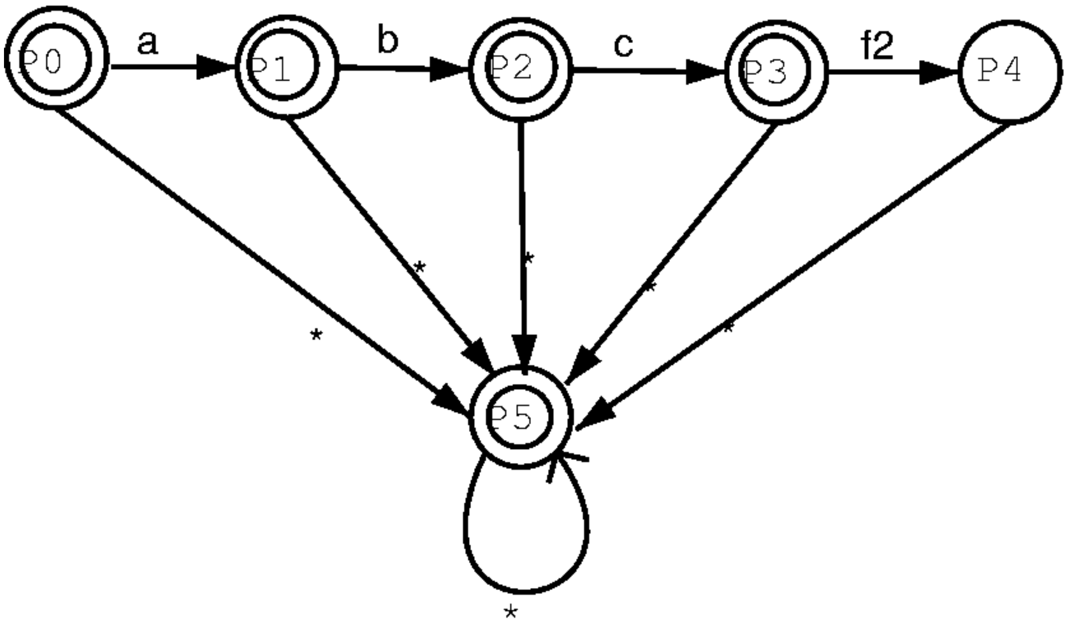


Figure 3.1: $\bar{A}$

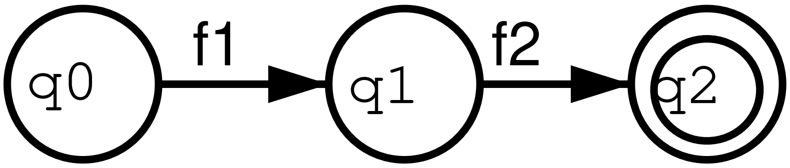


Figure 3.2: A_w^0

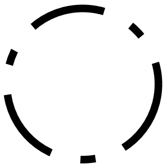


Figure 3.3: This denotes a marked node

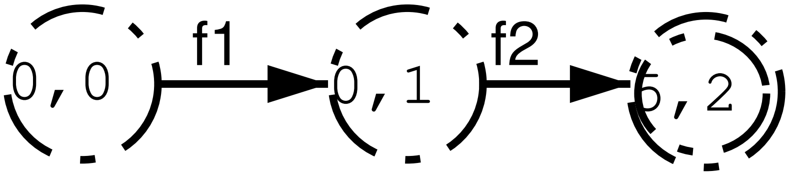


Figure 3.4: A_X^0

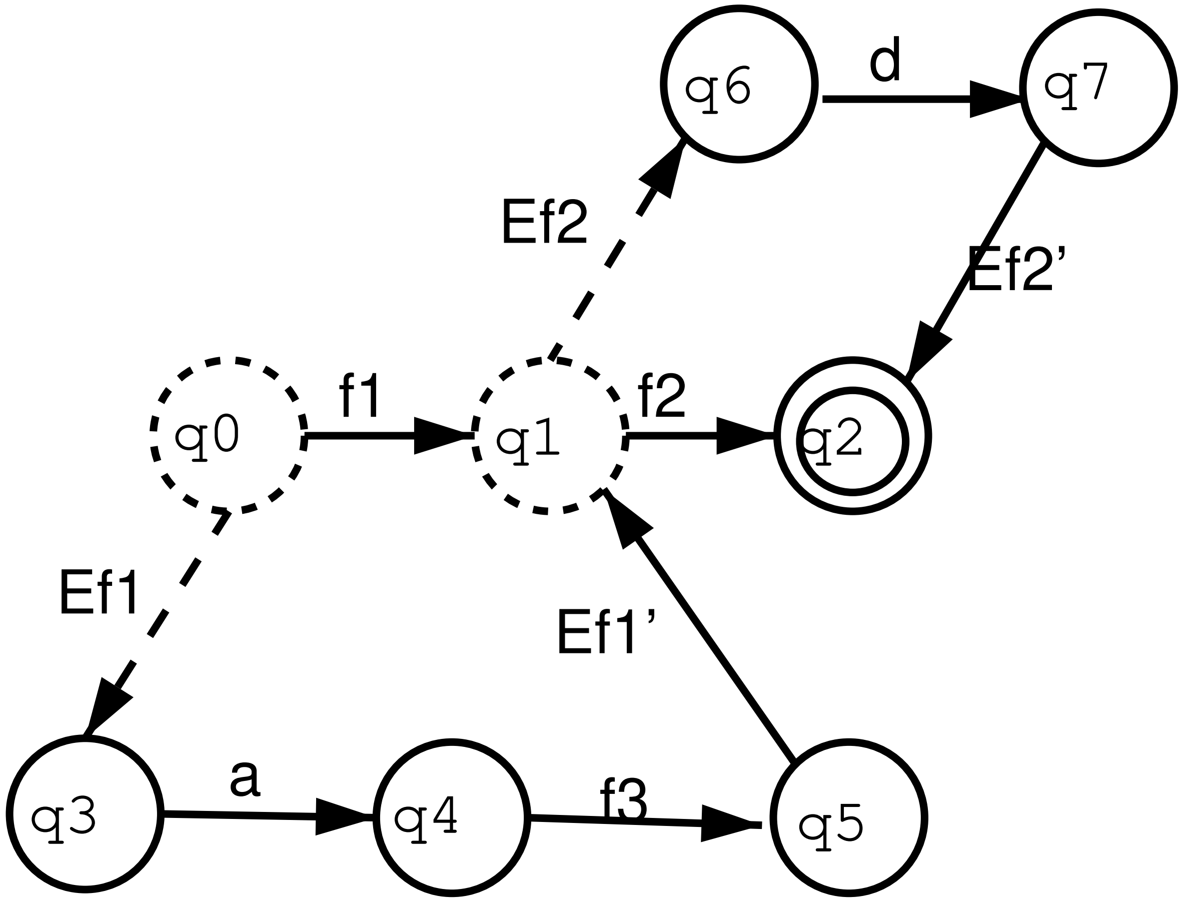


Figure 3.5: A_w^1

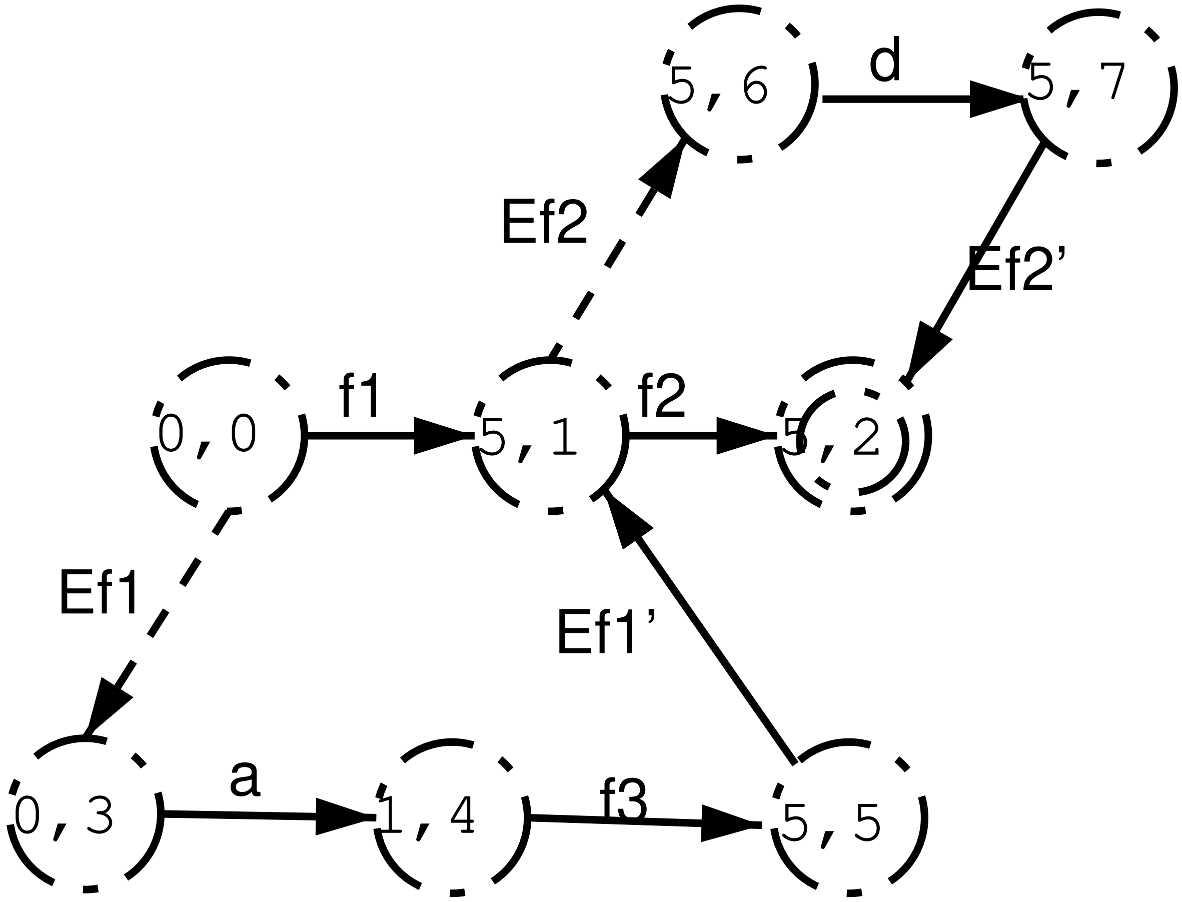


Figure 3.6: A_X^1

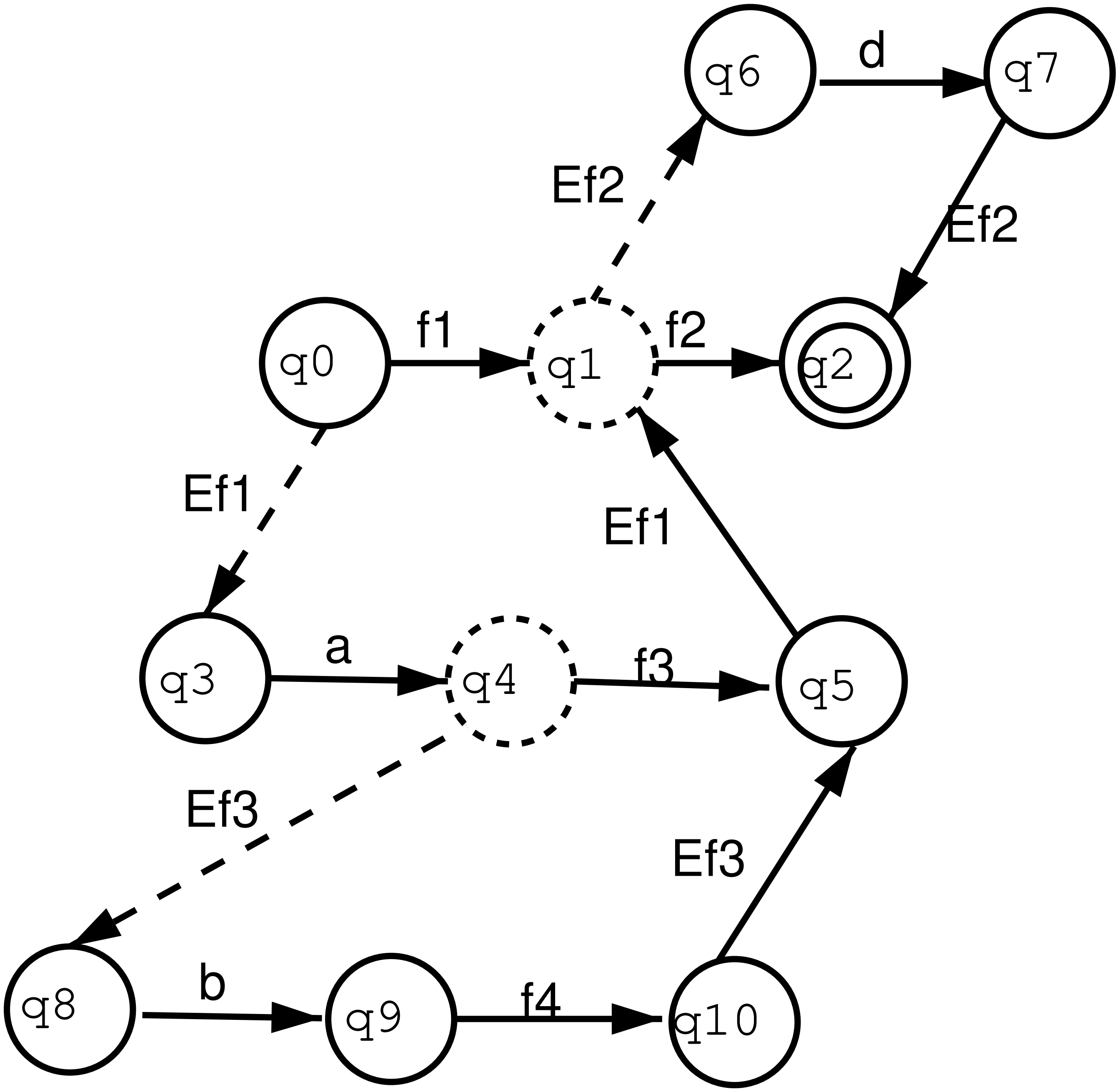


Figure 3.7: A_w^2

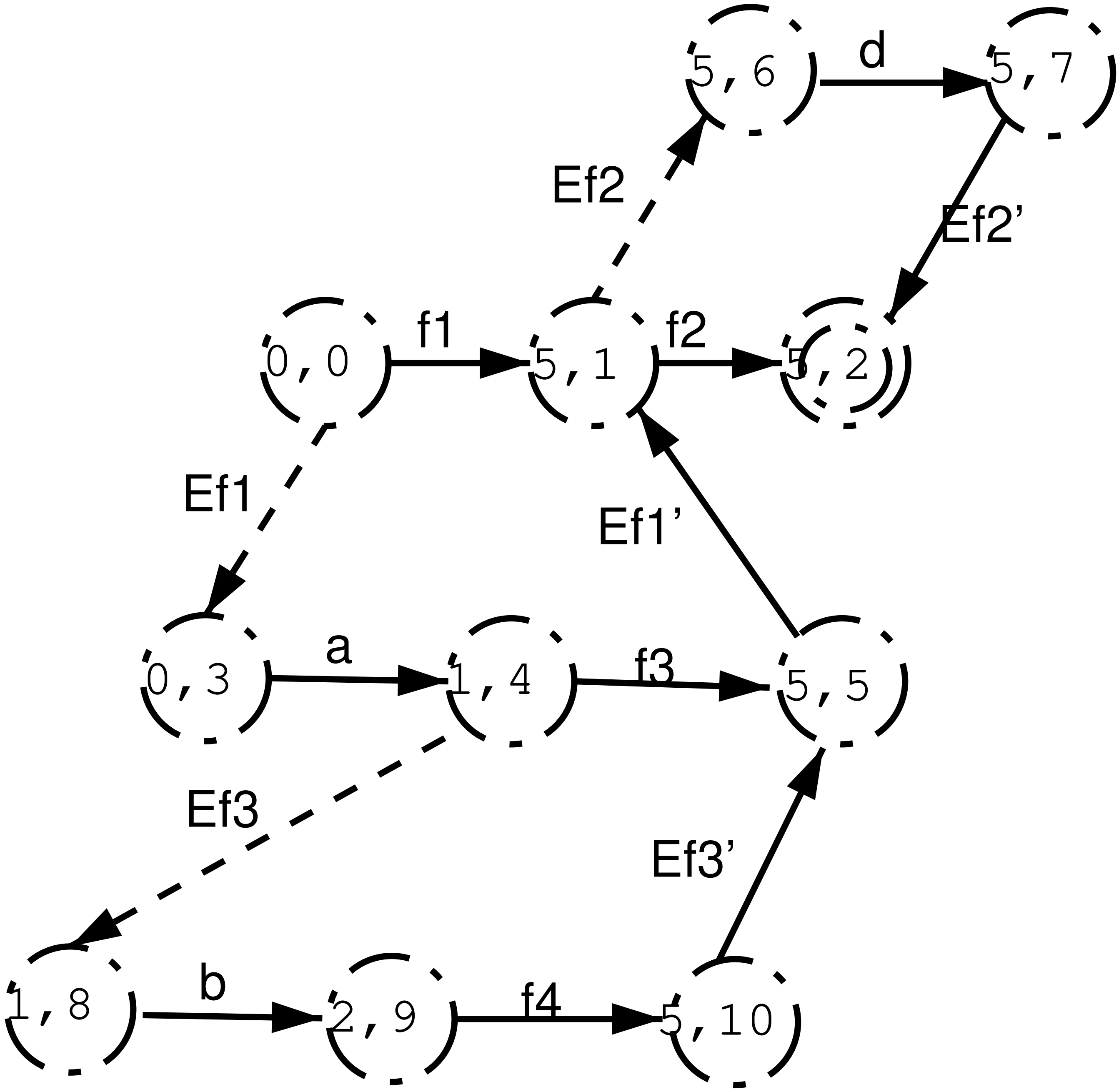


Figure 3.8: A_x^2

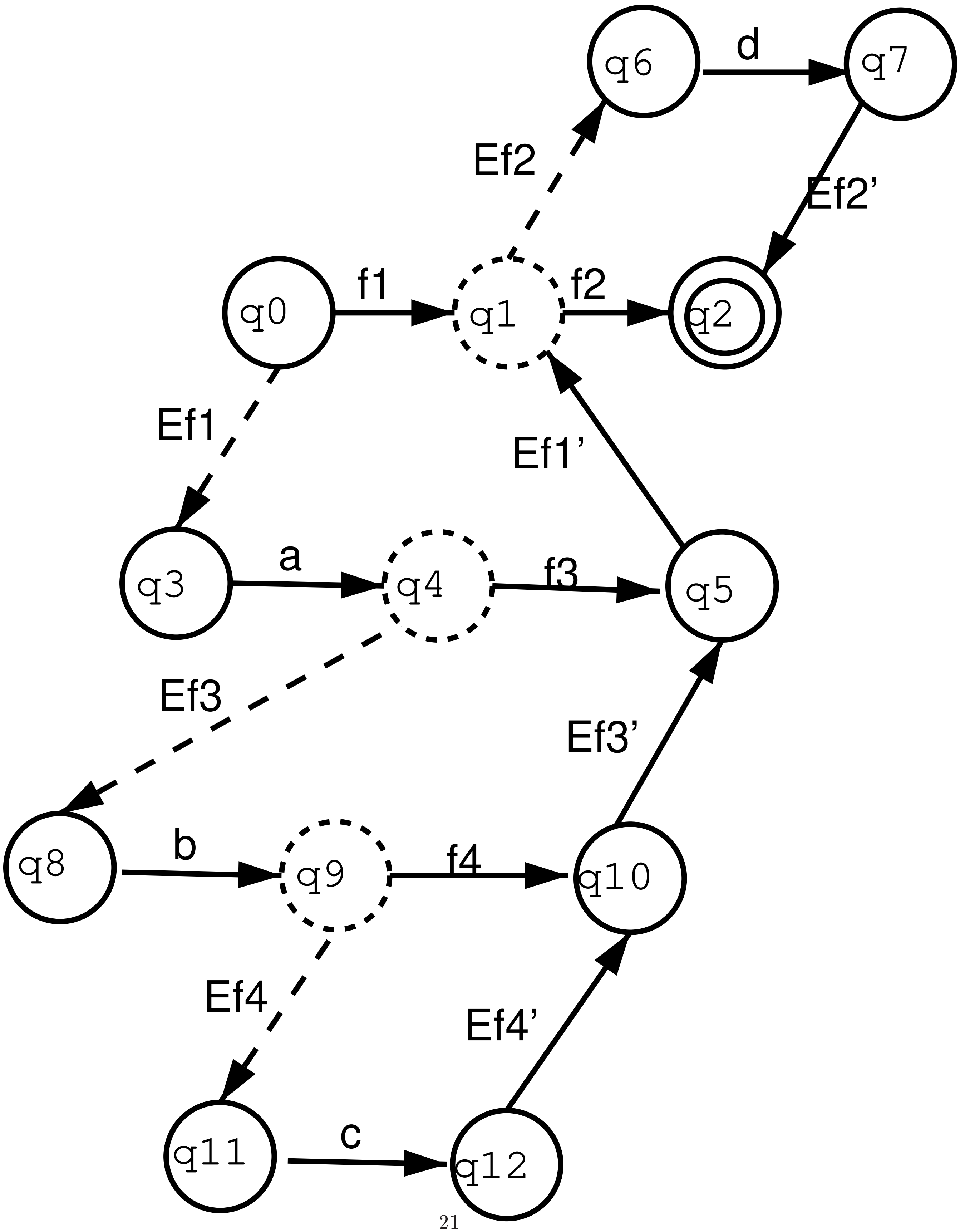
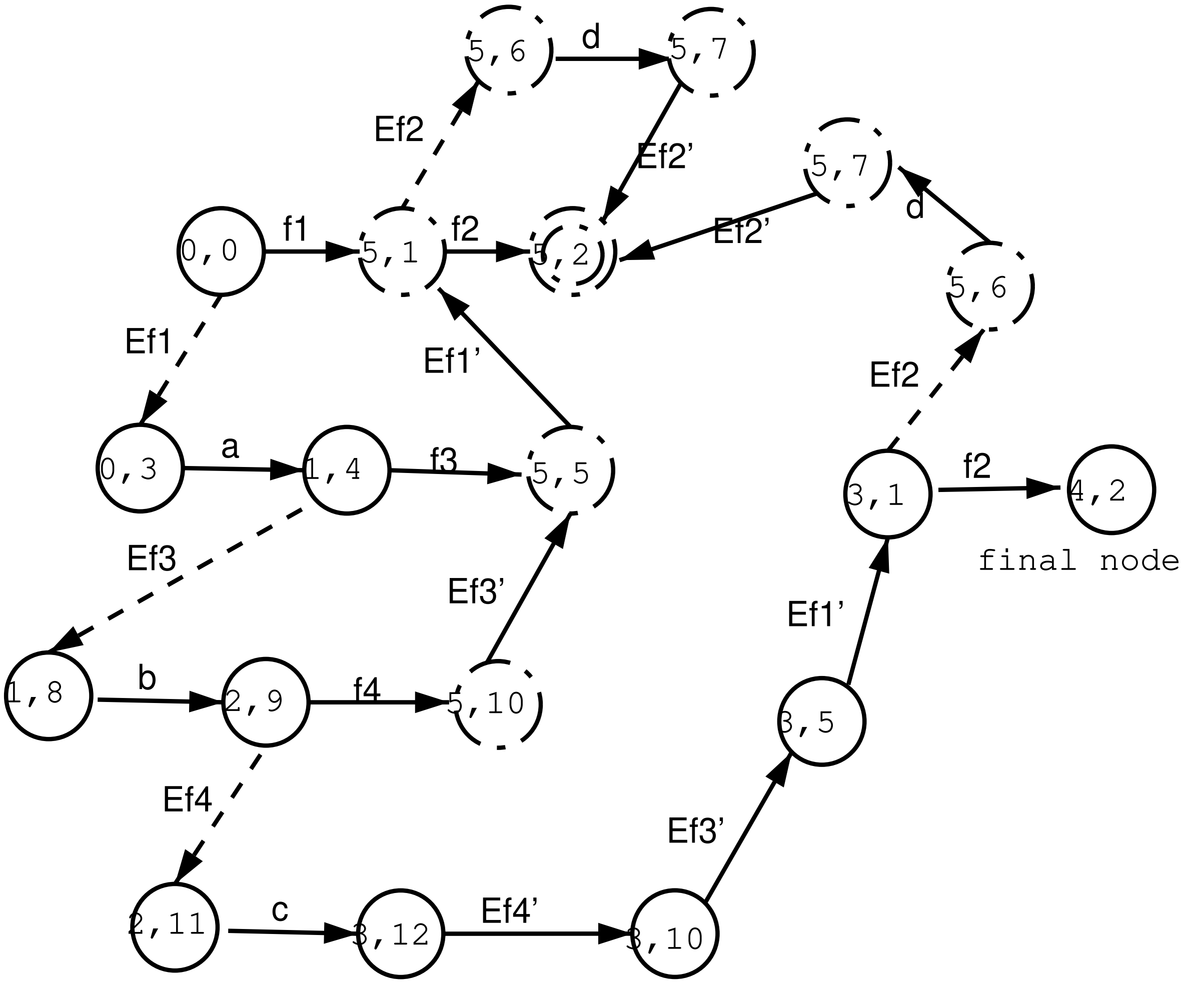


Figure 3.9: A_w^3



the solid circles show the final path

Figure 3.10: A_X^3

Chapter 4

Identifying Independent Substrings

In the discussion to follow when we say that a function f_i *depends* on another function f_j, we mean that the decision whether to invoke f_j or not depends on the output of f_i.

Given a word w (a string of functions), we try to *break* it into $s_1, s_2, \ldots, s_n$ where $w = s_1.s_2 \ldots s_n$ and

- Each $s_i = f_{i_1}.f_{i_2} \ldots f_{i_m}$
- f_{i_k} (in s_i) is dependent only on f_{i_j} (in s_i), $j < k$ and no other function in w.

These f_{i_1} for $i = 1, 2 \ldots n$ are called the *breakpoints*.
Note that in each s_i

- f_{i_1} is an independent function. i.e all *breakpoints* are independent functions and vice versa.
- f_{i_k} depends only on the functions to the left of it in s_i.

It is assumed all the marked nodes in A_X are pruned
If such a *breaking* is possible, the following result holds.
Result: In the marked cross product automaton A_X, a function having only one fork node is a *breakpoint* (but the converse is not true).
Proof: Let the fork node be called n_f and correspond to a function f. We prove by contradiction. Suppose that f is not a *breakpoint*. Then f should depend on the output of some function to the left of it in the string w. After doing the marking on A_X, we know whether we have to invoke f at the node n_f. So the decision whether to invoke f or not does not depend on the output of any other function. Therefore f is independent and hence a *breakpoint*. Contradiction.

The converse is not true. Corresponding to a *breakpoint* we may have many fork nodes in A_X. These nodes are equivalent but can appear as distinct nodes in A_X.

example :

- $w = f.g.h$
- $\tau_{out}(f) = c|d$
- $\tau_{out}(g) = y$

- $\tau_{out}(h) = z$
- $R = (c.(g|a).e)|(d.(g|b).(z|e))$

In A_X we know the nodes which are probable candidates of being equivalent (the fork nodes corresponding to each function). But checking the equivalence of these nodes in NP hard, as A_X is an NFA and to check the equivalence of two nodes we have to first convert the automata that follow these nodes to DFAs (NP hard) and then check whether these DFAs identify the same language.

Even though theoretically the above problem seems to be NP hard, such cases are a rare occurrence in practical examples. Considering simple target schemas and input documents, such equivalent nodes will usually appear as a single node in A_X making it easy to identify them. In cases where these nodes are distinct, checking their equivalence though theoretically NP hard may not take much time as the automata following them may not blow up exponentially when made deterministic, or even if they do, their size may not be difficult to handle.

Another interesting analysis is to have restrictions on the schema level, to avoid cases when there is an exponential blow up of states during the determinization process.

Bibliography

[1] Tova Milo, Serge Abiteboul, Bernd Amann, Omar Benjelloun, and Fred Dang Ngoc. Exchanging Intensional XML Data. *SIGMOD*, 2003.