Non-backtracking Top-down Algorithm for Checking Tree Automata Containment

Tadahiro Suda\(^1\) and Haruo Hosoya\(^2\)

Graduate School of Information Science and Technology, University of Tokyo, Japan
\{tada,hahosoya\}@is.s.u-tokyo.ac.jp

Abstract. Checking tree automata containment is a fundamental operation in static verification of XML processing programs. However, tree automata containment problem is known to be EXP TIME-complete and a standard algorithm with determination of automata easily blows up even in practical cases. Hosoya, Vouillon, and Pierce have proposed a top-down algorithm that efficiently works for a large class of typical instances. However, there still remains a considerable inefficiency because of repeated calculation incurred by backtracking. In this paper, we propose a non-backtracking top-down algorithm which improves this inefficiency. In the algorithm, we introduce “dependencies” among performed computations and, by exploiting these, we can recover certain kinds of information lost by backtracking. One difficulty in constructing such algorithm is, however, that, since some dependency information can be useless, we may be misled to needless computation by using such information. To alleviate this problem, we carefully check the usefulness of each dependency whenever we use it. Since these checks introduce a subtlety to our algorithm, we rigorously formalize it with a correctness proof. Our preliminary experiments show that our algorithm works more efficiently compared to the previous algorithm.

1 Introduction

Tree automata are a finite-state machine model for accepting trees. This paper aims at studying an efficient algorithm for the containment problem of tree automata.

The primary motivation of this study is the application to static typechecking for XML processing. XML \cite{2} is a world-wide emerging standard for tree-structured documents that allows user-defined schemas for imposing structural constraints on those data. The purpose of static typechecking is to analyze a program for processing such XML documents and guarantee, before the execution, that generated documents conform to the schema given by the user. Though various methods for static typechecking have been proposed \cite{8,1,6,10,11,16}, the containment check between schemas is often used as the most important core (in particular, in XDuCe \cite{8}, CDuCe \cite{1}, XQuery \cite{6}, and several theoretical frameworks \cite{10,16}). As tree automata have been proved to be the most natural
model for schemas (e.g., [12]), we focus on the containment problem of tree automata in this paper. Unfortunately, the containment problem of tree automata is known to have a high complexity: EXPTIME-complete [14]. Since, in XML typechecking, we often consider tree automata with a large number of states (more than 100 states), naive algorithms are often not usable. In particular, a standard algorithm which involves determinization of automata easily blows up since it needs to transform input automata to completely separate automata with the number of states exponential in the size of the input.

Hasegawa, Vuillon, and Pierce have proposed a top-down algorithm [9] for checking tree automata containment efficiently in typical cases. Their algorithm checks a given goal of containment by recursively expanding it to subgoals in a top-down way. During this check, the algorithm keeps track of already-encountered containment goals as a set of "assumptions" in order to avoid repetition of checking the same containment. However, in a later computation, the assumptions may turn out to be false. At that moment, the algorithm performs backtracking, which is not only costly by itself but also may incur repetition of the same computation because some valuable information can be lost.

In this paper, we propose a non-backtracking top-down algorithm which improves this inefficiency. In addition to already-encountered containments as in the original top-down algorithm, our algorithm maintains dependency relations among these containments. By exploiting this dependency information, we can recover the above-mentioned information lost by backtracking and thus avoid the wasteful repetition of computation. However, some dependency information proved during the calculation can be useless (i.e., one containment depends on another containment that actually does not hold), a naive introduction of this mechanism would again cause extra computation. In order to overcome this difficulty, we carefully craft our algorithm in a way that eliminates dependency relations as soon as they are proved to be useless. Since the algorithm becomes quite subtle because of this handling of useless dependencies, we rigorously formalize our algorithm and prove its correctness. We have also implemented the algorithm and compared it with the original top-down algorithm using several non-trivial examples. The result indicates that our algorithm runs approximately 3 times to 8 times more efficiently than the previous one in these examples.

The rest of this paper is organized as follows. We first give preliminary notations in Sec. 2. In Sec. 3, we first informally explain the existing top-down algorithm, where we point out the inefficiency caused by backtracking, and then describe the problem arising from naively introducing dependency information. In Sec. 4, we formalize our non-backtracking algorithm that desirably handles dependencies and prove its correctness. We discuss related work in Sec. 5. Finally, we verify the efficiency of our algorithm in practical cases by preliminary experiments in Sec. 6. Due to space constraints, we omit the proofs of theorems from this paper. These can be found in [15].

There are numerous schema languages, but most can be expressed more or less by tree automata. (e.g., DTD [2], XML-schema [3], RELAX NG [3]).
2 Preliminaries

Let $\mathcal{L}$ be a finite set of labels, ranged over by $l$. We define a tree $t$ as a labeled binary tree by the following grammar:

\[ t ::= # \mid l(t, t) \]

A tree automaton $M$ is a triplet $(\mathcal{L}, Q, \delta)$ where $\mathcal{L}$ is a finite set of labels, $Q$ is a finite set of states, and $\delta$ is a transition function from states to types. Types are defined as subsets of $\{ # \} \cup \{ l(s, s') : l \in \mathcal{L}, s, s' \in Q \}$. We use the meta-variables $s, S$, and $T$ to range over $Q$, $2^Q$, and types, respectively. Also, we write $\delta(S)$ for $\bigcup_{s \in S} \delta(s)$. For any type $T$ appearing in $M$, we write $[T]_M$ to denote the set of all the trees matched by $T$, formally, the least solution of the following equations: let us write $[a]_M$ for $[[\{a\}]]_M$ and

\[
\begin{align*}
[\phi]_M &= \phi \\
[#]_M &= \{ # \} \\
[[t(s, s')]_M &= \{ l(t, t') : t \in [[\delta(s)]]_M, t' \in [[\delta(s')]_M \} \\
[[a_1, \ldots, a_n]]_M &= \bigcup_{i=1}^n [a_i]_M
\end{align*}
\]

For any set $S$ of states, we define the language $[[S]]_M$ of $S$ as $[[\delta(S)]]_M$. In the rest of paper, we simply write $[[\cdot]]$ for $[[\cdot]]_M$ when $M$ is clear from the context. Also, we write $[\{s\}]$ for $[[\{s\}]]_M$. The tree automata containment problem is formalized as follows: for a tree automaton $M$ and two sets of states $S$ and $S'$, answer “yes” if and only if $[[S]] \subseteq [[S']]$.

3 Problems

Hosoya, Vouillon, and Pierce have proposed a “top-down” algorithm for checking tree automata containment [9]. In their algorithm, containment check of given two sets of states proceeds by recursively unfolding states by the transition function. That is, it starts with checking the given pair of state sets by comparing the types yielded by unfolding the states; this eventually leads to checks of other pairs of state sets, and then we repeat unfoldings and comparisons similarly for these. In order to avoid repetition of checking the same pair, the algorithm keeps an “assumption” set to remember pairs of state sets that have already been seen: later when it encounters a pair that is already in the assumption set, it stops further check of this.

For example, let us consider checking the containment $[[s_1]] \subseteq [[s_2]]$ for the tree automaton in Fig. 1 by their algorithm. In the example, we write an assumption set in the following form:

\[ \{ s_1 \prec s_1', \ldots, s_n \prec s_n' \} \]

Intuitively, each $s_i \prec s_i'$ stands for $[[s_i]] \subseteq [[s_i']]$.

We start with the empty assumption set. To check $[[s_1]] \subseteq [[s_2]]$, we first add $s_1 \prec s_2$ to the assumption set:

\[ \{ s_1 \prec s_2 \} \]
\[ \mathcal{L} = \{a, b, c, d, e, f\} \quad Q = \{s_1, \cdots, s_8\} \]

\[ \delta: s_1 \mapsto \{a(s_1, s_1), b(s_3, s_4)\} \quad s_7 \mapsto \{\#, a(s_7, s_7)\} \]

\[ s_2 \mapsto \{a(s_2, s_3), b(s_5, s_7), b(s_6, s_8)\} \quad s_8 \mapsto \{\#, a(s_2, s_2), b(s_2, s_2)\} \]

\[ s_3 \mapsto \{e(s_3, s_4), d(s_9, s_9), e(s_1, s_4)\} \quad s_9 \mapsto \{f(s_5, s_5)\} \]

\[ s_4 \mapsto \{\#, a(s_1, s_1)\} \quad s_{10} \mapsto \{f(s_5, s_5)\} \]

\[ s_5 \mapsto \{e(s_2, s_8), d(s_{10}, s_{10}), e(s_2, s_2)\} \quad s_{11} \mapsto \{f(s_5, s_5), f(s_5, s_6)\} \]

\[ s_6 \mapsto \{e(s_2, s_2), d(s_{11}, s_{11}), e(s_8, s_8)\} \]

**Fig. 1.** An example of tree automaton

We then unfold \(s_1\) and \(s_2\) by the transition function \(\delta\) and check the following containment of types assuming the validity of containments in the assumption set (i.e., assuming that \([s_1] \subseteq [s_2]\) holds):

\[ \llbracket a(s_1, s_1), b(s_3, s_4) \rrbracket \subseteq \llbracket a(s_2, s_2), b(s_5, s_7), b(s_6, s_8) \rrbracket \]

It is enough to compare the elements on both sides that have the same label. Thus, we check both of the following two containments:

(A) \(\llbracket a(s_1, s_1) \rrbracket \subseteq \llbracket a(s_2, s_2) \rrbracket\)

(B) \(\llbracket b(s_3, s_4) \rrbracket \subseteq \llbracket b(s_5, s_7), b(s_6, s_8) \rrbracket\)

First, for the containment (A), showing \([s_1] \subseteq [s_2]\) suffices. Since \(s_1 \preceq s_2\) is already in the assumption set, we stop checking this containment further and we judge right away that it holds.

Next, for the containment (B), it suffices to show:

\[ [s_3] \times [s_4] \subseteq ([s_5] \times [s_7]) \cup ([s_6] \times [s_8]) \]

The right hand side of this containment can be transformed into the conjunctive normal form as follows: let us write \(T\) for the set of all trees and

\[ (([s_5] \times T) \cap (T \times [s_7])) \cup (([s_6] \times T) \cap (T \times [s_8])) \]

\[ = (([s_5] \times T) \cup (T \times [s_8])) \cap (([s_6] \times T) \cup (T \times [s_7])) \]

\[ \cap ([s_7, s_8]) \]

Since \([s_3] \times [s_4] \subseteq ([s_5] \times T) \cup (T \times [s_8])\) is equivalent to \(\llbracket s_3 \rrbracket \subseteq \llbracket s_5 \rrbracket\) or \(\llbracket s_4 \rrbracket \subseteq \llbracket s_8 \rrbracket\) and similarly for the other clauses in the last formula, it is enough to check all of the followings.

(C) \(\llbracket s_3 \rrbracket \subseteq \llbracket s_5 \rrbracket\) or \(\llbracket s_4 \rrbracket \subseteq \llbracket s_8 \rrbracket\)

(D) \(\llbracket s_3 \rrbracket \subseteq \llbracket s_6 \rrbracket\) or \(\llbracket s_4 \rrbracket \subseteq \llbracket s_7 \rrbracket\)

(E) \(\llbracket s_3 \rrbracket \subseteq \llbracket s_5, s_6 \rrbracket\)

(F) \(\llbracket s_4 \rrbracket \subseteq \llbracket s_7, s_8 \rrbracket\)

**Backtracking** Note that the algorithm needs to check disjunctions of containments and this is the cause of backtracking. Let us see this from how the algorithm works for (C).

We begin with checking the first containment \([s_3] \subseteq [s_5]\), for which we first add \(s_3 \prec s_5\) to the assumption set:

\[ \{s_1 \prec s_2, \quad s_3 \prec s_5\} \]
We then unfold $s_3$ and $s_5$ by $\delta$ and check the following containment:

$$[[c(s_4, s_4), d(s_9, s_9), e(s_4, s_4)]] \subseteq [[c(s_8, s_8), d(s_{10}, s_{10}), e(s_2, s_2)]]$$

In order for this containment to hold, we need to show each of the following three checks:

\[\text{(G)} \quad [e(s_4, s_4)] \subseteq [e(s_8, s_8)] \quad \text{(H)} \quad [d(s_9, s_9)] \subseteq [d(s_{10}, s_{10})] \quad \text{(I)} \quad [e(s_4, s_4)] \subseteq [e(s_2, s_2)]\]

For the check of (G), since we only need to check $[s_4] \subseteq [s_8]$, we add $s_4 \prec s_8$ to the assumption set as

$$\{ s_1 \prec s_2, \ s_3 \prec s_5, \ s_4 \prec s_8 \}$$

and check $[[\#, a(s_1, s_1)]] \subseteq [[\#, a(s_2, s_2)]]$ as given by the unfolding of the states $s_4$ and $s_8$. This succeeds by the trivial relation $[[\#]] \subseteq [[\#]]$ and the fact that the assumption set already contains $s_1 \prec s_2$. In order to show (H), all we need is to check $[s_9] \subseteq [s_{10}]$. This holds since the unfolding of the states reduces this relation to checking $[s_9] \subseteq [s_5]$ and $s_3 \prec s_5$ is already in the assumption set. As a result of this check, the assumption set becomes

$$\{ s_1 \prec s_2, \ s_3 \prec s_5, \ s_4 \prec s_8, \ s_9 \prec s_{10} \}.$$

Finally, the containment (I) requires checking $[s_4] \subseteq [s_2]$, but it does not hold since $\# \notin \delta(s_4)$ and $\# \notin \delta(s_2)$. This makes the containment (I) false, leading $[s_3] \subseteq [s_5]$ to fail.

Here, since we have assumed $[s_3] \subseteq [s_5]$ but it turns out to be false, we roll back to the point (C). Since this is just before $s_3 \prec s_5$ was added to the assumption set, the set is now reverted to

$$\{ \ s_1 \prec s_2 \}.$$

This backtracking is a source of inefficiency for the following reasons.

1. After the backtracking, we check the second check $[s_4] \subseteq [s_8]$ of the disjunction in (C). However, the algorithm repeats the same calculation that has already been done in the check of (G).
2. After finishing the check of (C), we continue with (D), which eventually leads to the check of $[s_4] \subseteq [s_2]$. Even though this relation has already been refuted when checking (I), the algorithm needs the same check once again.

**Introducing dependency information** We propose a technique to improve the above inefficiency of the previous top-down algorithm. Our algorithm works similarly to the previous one but it additionally maintains more refined information including containment dependencies (expressing that “a containment depends on other containments”) and refuted containments and uses these for
avoiding the above-mentioned needless calculation. Since our new algorithm
never reverts dependency information to a previous point in a blind way, we
call it a non-backtracking algorithm as oppose to the previous backtracking one.
For example, note that, in checking (G) above, when we judged \([s_4] \subseteq [s_8]\), we
assumed the relation \([s_1] \subseteq [s_2]\). For this, our algorithm maintains the dependency
"([s_4] \subseteq [s_8]\) depends on \([s_1] \subseteq [s_2]\)." This information is useful since,
later when we need to check \([s_4] \subseteq [s_8]\), we can stop going further but instead
immediately say "succeed, provided \([s_1] \subseteq [s_2]\) holds."

However, in order to obtain an enough efficient algorithm, we need a care
in constructing our new algorithm. Among dependencies that have been proved
during the course of computation, there are useless ones expressing "a contain-
ment \(A\) depends on a false containment \(B\)." It would make the algorithm slower
if we naively return as described in the last paragraph whenever we encounter \(A\)
for the next time. This is because the "conditional success" will later be canceled
by \(B\)'s falsity any way and the computation from now to then will be wasted.
One might think that such a dependency is stupid from the first place, but
the issue is that \(B\)'s falsity cannot be known at the moment where the depen-
dency is generated. As an example, the dependency "([s_9] \subseteq [s_{10}] \) depends on
([s_3] \subseteq [s_5])" was proved in the check of (H). However, \([s_3] \subseteq [s_5]\) was refuted later;
the dependency turns out to be useless at the refutation point.

Our approach to this issue is to check uselessness of each dependency (i.e.,
validity of the depended containments) whenever we use it. However, such treat-
ment is not straightforward since blindly performing such a check will make
algorithm infinitely loop. Our observation is that the depended containments
that are unsafe to check can be characterized as those "still under checking,"
and a formalization of an algorithm involving a careful classification of contain-
ment relations is our main technical contribution in this paper.

4 Algorithm

We first introduce the following notation to express the containments that are
handled in our algorithm: a containment \(cont\) is either of the form \(s \prec S\) (standing
for \("[s] \subseteq [S]\"), or of the form \(T \prec T'\) (standing for \("[T] \subseteq [T']\"), or of the form
\(s \prec S \mid s' \prec S'\) (standing for \("[s] \subseteq [S]\) or \([s'] \subseteq [S']\"). We also use \(\pi\) to range
over sets of containments of the form \(s \prec S\).

Our algorithm maintains an (extended) dependency set \(A = (U, D, F)\) where
\(U\) and \(F\) are sets of pairs of the form \(s \prec S\) and \(D\) is a set of triplets of the form
\(\pi \vdash s \prec S\). We store under-checking containments in \(U\), dependencies ("\(s \prec S\)
depends on the containments in \(\pi'\)"") in \(D\), and failed containments in \(F\). For a
set \(D\) of dependencies, \(D_{pair}\) denotes \([s \prec S \mid \pi \vdash s \prec S \in D]\). A dependency set
\((U, D, F)\) is well-defined if it satisfies \(U \cap F = \phi, U \cap D_{pair} = \phi, F \cap D_{pair} = \phi,\)
and, moreover, for any \((s, S)\), there is at most one triplet of the form \(\pi \vdash s \prec S\) in
\(D\). In the rest of this paper, we assume that any dependency set is well-defined.
A containment set \(\pi\) is consistent if \([s] \subseteq [S]\) holds for all \(s \prec S\) in \(\pi\) and a
dependency set \((U, D, F)\) is consistent if it satisfies the following conditions:
- If $\pi \vdash s \prec S \in D$ and $\pi$ is consistent, then $[s] \subseteq [S]$.
- If $s \prec S$ $\in F$, then $[s] \not\subseteq [S]$.

(Note that the set $U$ does not constrain the consistency of the dependency set.)

Our algorithm is defined by a set of rules that derive the following relation

$$A, \text{cont} \sim A', \rho$$

where $\rho := \pi | \bot$ and $\pi$ is a subset of the under-checking pairs $U$ of $A$. This relation reads “the algorithm checks a given containment $\text{cont}$ under a given dependency set $A$ and, as the result, it transforms $A$ to $A'$ and returns $\rho$”. The meaning of the last part “returns $\rho$” depends on the form of $\rho$.

$\rho = \pi$: “if both $A$ and $\pi$ are consistent, then $\text{cont}$ holds.”

$\rho = \bot$: “if $A$ is consistent, then $\text{cont}$ does not hold.”

Since we are actually interested in simply checking $[s_0] \subseteq [S_0]$ for given $s_0$ and $S_0$, we start the algorithm by giving $A = (\phi, \phi, \phi)$ and $\text{cont} = s_0 \prec S_0$. (Note that, when it succeeds, $\rho = \pi = \phi$ since we have $U = \phi$ and ensure $\pi \subseteq U$.)

**Derivation Rules** First, let us see the derivation rules for the case $\text{cont} = s \prec S$ (where $A = (U, D, F)$, $A' = (U', D', F')$ and $A_i = (U_i, D_i, F_i)$).

\[
\begin{align*}
A, s \prec S & \sim A, \phi \quad \text{(Mem)} \\
A, s \prec S & \sim A, \{s \prec S\} \quad \text{(InU)} \\
A, s \prec S & \sim A, \bot \\
A, s \prec S & \sim A', \pi \quad \text{(Unf)}
\end{align*}
\]

\[
\begin{align*}
s \not\in S & \quad s \prec S \not\in U \cup F \cup D_{par} \\
A, s \prec S & \sim (U \cup \{s \prec S\}, D, F), \delta(s) \prec \delta(S) \sim A', \pi
\end{align*}
\]

\[
\begin{align*}
s \not\in S & \quad s \prec S \not\in U \cup F \cup D_{par} \\
A, s \prec S & \sim (U \cup \{s \prec S\}, D', F'), \delta(s) \prec \delta(S) \sim A', \bot
\end{align*}
\]

where $\pi = \{s_1 \prec S_1, \ldots, s_n \prec S_n\}$ and $A_0 = (U \cup \{s \prec S\}, D \setminus \{\pi \vdash s \prec S\}, F)$$A, s \prec S \sim (U_m \setminus \{s \prec S\}, D_m \cup \{\pi'' \vdash s \prec S\}, F_m), \pi''$ where $\pi'' = \bigcup_{i=1}^{n} \pi \setminus \{s \prec S\}$

$$\pi \vdash s \prec S \in D \quad \text{for all } 1 \leq i \leq m,$$ $A_{i-1}, s_i \prec S_i \sim A_i, \pi_i$

\[
\begin{align*}
A_{j-1}, s_j \prec S_j & \sim A_j, \bot \\
A_j, \delta(s) \prec \delta(S) & \sim A', \pi'
\end{align*}
\]

where $\pi = \{s_1 \prec S_1, \ldots, s_m \prec S_m\}$ and $A_0 = (U \cup \{s \prec S\}, D \setminus \{\pi \vdash s \prec S\}, F)$$A, s \prec S \sim (U' \setminus \{s \prec S\}, D' \cup \{\pi'' \vdash s \prec S\}, F'), \pi''$ where $\pi'' = \pi \setminus \{s \prec S\}$

$$\pi \vdash s \prec S \in D \quad \text{for all } 1 \leq i \leq j-1,$$ $A_{i-1}, s_i \prec S_i \sim A_i, \pi_i$

\[
\begin{align*}
A_{j-1}, s_j \prec S_j & \sim A_j, \bot \\
A_j, \delta(s) \prec \delta(S) & \sim A', \bot
\end{align*}
\]

where $\pi = \{s_1 \prec S_1, \ldots, s_m \prec S_m\}$ and $A_0 = (U \cup \{s \prec S\}, D \setminus \{\pi \vdash s \prec S\}, F)$$A, s \prec S \sim (U' \setminus \{s \prec S\}, D' \cup \{\pi'' \vdash s \prec S\}, F'), \pi''$ where $\pi'' = \pi \setminus \{s \prec S\}$

$$\pi \vdash s \prec S \in D \quad \text{for all } 1 \leq i \leq j-1,$$ $A_{i-1}, s_i \prec S_i \sim A_i, \pi_i$
If \( s \in S \), we can easily conclude \([s] \subseteq [S]\) by the definition of languages. We thus return \( \rho = \phi \) as the result of this check (Empty). When \( s \prec S \in U \) (where \((U, D, F)\) is the original dependency set), we stop the further check of \( s \prec S \) and immediately return \( \rho = \{ s \prec S \} \) as the result (InU). This rule means the trivial statement "if \([s] \subseteq [S]\) holds, then \([s] \subseteq [S]\) holds." If \( s \prec S \in F \), then this implies that we have already checked \([s] \not\subseteq [S]\) and therefore return \( \rho = \perp \) immediately (InF\(_l\)). When \( s \prec S \) is neither in \( U \), \( F \), nor \( D_{pair} \), then this means that we encounter \( s \prec S \) for the first time and therefore we first add \( s \prec S \) to \( U \) (this ensures the termination of our algorithm since the number of containments in \( U \cup F \cup D_{pair} \) increases monotonously) and then check \( \delta(s) \prec \delta(S) \) under the new dependency set \((U \cup \{s \prec S\}, D, F)\). Suppose that the check succeeds with \( A' = (U', D', F', \pi) \). Then, its direct meaning is that \([s] \subseteq [S]\) depends on \( \pi \). However, this actually implies that \([s] \subseteq [S]\) depends on \( \pi' = \pi \setminus \{s \prec S\} \) (intuitively because we can construct a proof tree such that every application of InU with \( s \prec S \) is recursively replaced by the proof tree showing \([s] \subseteq [S]\) depends on \( \pi \)). Thus we add \( \pi' \vdash s \prec S \) to \( D' \), remove \( s \prec S \) from \( U' \), and then return \( \rho = \pi' \) (Unfold). On the other hand, suppose that \([s] \subseteq [S]\) is refuted as a result of checking \( \delta(s) \prec \delta(S) \). In this case, we remove \( s \prec S \) from \( U' \), add \( s \prec S \) to \( F' \), and return \( \rho = \perp \) (Unfold\(_f\)).

If \( \pi \vdash s \prec S \in D \), then this means that we have already proved that \( s \prec S \) depends on the consistency of \( \pi \). However, since the dependency may be useless, that is, \( \pi \) may be inconsistent, we successively check each pair in \( \pi \). (Before checking these pairs, we add \( s \prec S \) to \( U \) and remove \( \pi \vdash s \prec S \) from \( D \) in order to avoid rechecking \( s \prec S \).) Suppose that all checks of pairs \( s_j \prec S_j \) in \( \pi \) succeed \( (1 \leq i \leq m) \), each proving that it depends on \( \pi_i \) for some \( \pi_i \). From this result and the above dependency \( \pi \vdash s \prec S \), we know that \( s \prec S \) depends on \( \bigcup_{i=1}^{m} \pi_i \). Then, similarly to Unfold, since the obtained dependency implies that \( s \prec S \) depends on \( \pi' = \bigcup_{i=1}^{m} \pi_i \setminus \{s \prec S\} \), we add \( \pi \vdash s \prec S \) to \( D' \), remove \( s \prec S \) from \( U' \), and return \( \rho = \pi' \) (InD). When some pair \( s_j \prec S_j \) in \( \pi \) turns out to be false, this makes \( \pi \) inconsistent and hence \( \pi \vdash s \prec S \) becomes useless. In this case, we recheck \( s \prec S \) by unfolding these states, similarly to Unfold and Unfold\(_f\), under the dependency set resulted from checking \( s_1 \prec S_1 \) through \( s_j \prec S_j \) with \( s \prec S \) in \( U \) (InD\&Unf and InD\&Unf\(_f\)).

Next, we show the rules for cont = \( T \prec T' \).

\[
\begin{align*}
T = \phi & \quad (\text{Empty}) \\
A, T \prec T' & \rightsquigarrow A, \phi \\
T = \{ \# \} & \quad (\text{Leaf}) \\
\# \in T' & \quad (\text{Leaf}) \\
T = \{ \# \} & \quad (\text{Leaf}) \\
\# \notin T' & \quad (\text{Leaf}) \\
T = \{ a_1, \ldots, a_m \} & \quad (\text{Union})
\end{align*}
\]

\[
T = \{ a_1, \ldots, a_m \} (m \geq 2) \quad \text{for all} \ 1 \leq i \leq m, \ A_{i-1} \prec A_i, \ a_i \prec A_i, \pi_i \\
A_0, T \prec T' & \rightsquigarrow A_0, \bigcup_{i=1}^{m} \pi_i
\]

\[
T = \{ a_1, \ldots, a_m \} (m \geq 2) \quad \text{for all} \ 1 \leq i \leq j - 1, \ A_{i-1} \prec A_i, \ a_i \prec A_i, \pi_i \\
A_j, \{ a_i \} \prec T' & \rightsquigarrow A_j, \perp \\
A_0, T \prec T' & \rightsquigarrow A_0, \perp
\]
\[ T = \{ l(s, s') \} \quad \text{for all } 1 \leq i \leq 2^m, \ A_i, s \sim S | s' \sim S' \sim A_i, \pi_i \]

where \[
\begin{align*}
\{ l'(s, s') \in T' \mid l' = l \} &= \{ l(s_1, s'_1), \ldots, l(s_m, s'_m) \}, \\
I_i \subseteq \{1, \ldots, m\}, \ S_i = \{ s_k \mid k \in I_i \} \text{ and } S'_i = \{ s'_k \mid k \not\in I_i \}
\end{align*}
\]

\[ A_0, T \sim T' \sim A_2^m, \bigcup_{i=1}^{2^m} \pi_i \] (Conj)

\[ T = \{ l(s, s') \} \quad \text{for all } 1 \leq i \leq j - 1, \ A_i, s \sim S | s' \sim S' \sim A_i, \pi_i \]

\[ A_{j-1}, s \sim S | s' \sim S' \sim A_{j-1} \perp \]

where \[
\begin{align*}
\{ l'(s, s') \in T' \mid l' = l \} &= \{ l(s_1, s'_1), \ldots, l(s_m, s'_m) \}, \\
I_i \subseteq \{1, \ldots, m\}, \ S_i = \{ s_k \mid k \in I_i \} \text{ and } S'_i = \{ s'_k \mid k \not\in I_i \}
\end{align*}
\]

\[ A_0, T \sim T' \sim A_j, \perp \] (Conj')

If \( T = \emptyset \), then \([T](= \phi) \subseteq [T']\) trivially holds and therefore we return \( \rho = \phi \) right away (Empty). If \( T = \{ \# \} \), then \([T]\subseteq [T']\) is equivalent to \( \# \in T' \).

Therefore, in this case, we return \( \rho = \phi \) immediately when \( \# \notin T' \) holds (Leaf). If \( T = \{ a_1, \ldots, a_m \} \), then we successively check \( a_i \sim T' \) for each \( i = 1, \ldots, m \). And if all checks of \( a_i \sim T' \) succeed with \( \pi_i \) as a result, this proves that \( T \sim T' \) depends on \( \rho = \bigcup_{i=1}^{m} \pi_i \) and therefore we return \( \rho \) (Union). On the other hand, if some pair \( a_j \sim T' (a_j \in T) \) is refuted, then this concludes \([T] \notin [T']\) and therefore we return \( \rho = \perp \) (Union').

When \( T = \{ l(s, s') \} \), it is enough to compare \( l(s, s') \) with the elements labeled with \( l \) in \( T' \). Suppose that such elements in \( T' \) are \( l(s_i, s'_i) \) for \( 1 \leq i \leq m \). Then the relation \([T] \subseteq [T']\) is equivalent to

\[
[[s]] \times [[s']] \subseteq \bigcup_{i=1}^{m} ([s_i] \times [s'_i]) = \bigcap_{I \subseteq \{1, \ldots, m\}} \left( \bigcup_{i \in I} [[s_i]] \times T \right) \cup \left( T \times \bigcup_{i \not\in I} [[s'_i]] \right)
\]

by transforming the right hand similarly to the process in Sec. 3. We thus check

\[ [s] \subseteq \bigcup_{i \in I} [s_i] \text{ or } [s'] \subseteq \bigcup_{i \not\in I} [s'_i] \]

for each subset \( I \) of \( \{1, \ldots, m\} \) (Conj and Conj').

Finally, the rules for the case \( \text{cont} = s \sim S \mid s' \sim S' \) are:

\[
\begin{align*}
A_i, s \sim S &\sim A', \pi \quad \text{(Front)} & A_i, s \sim S &\sim A', \perp & A_i, s \sim S &\sim A', \pi \quad \text{(Post)} \\
A_i, s \sim S | s' \sim S' &\sim A', \pi & A_i, s \sim S | s' \sim S' &\sim A', \perp & A_i, s \sim S | s' \sim S' &\sim A', \pi
\end{align*}
\]

(Neither)

Here, we need to check whether or not at least one of \([s] \subseteq [S] \) and \([s'] \subseteq [S'] \) holds. Hence, we first check \([s] \subseteq [S] \) and, if it succeeds with \( \pi \), then we return \( \rho = \pi \) without checking \([s'] \subseteq [S'] \) (Front); if \([s] \subseteq [S] \) is refuted, then we check \([s'] \subseteq [S'] \) (Post and Neither).

**Theorem 1.** The complexity of the algorithm is bounded in \( 2^{|Q|} \).

**Theorem 2** (Soundness). Suppose \( A_i, s \sim S \sim A', \pi \). If both \( A \) and \( \pi \) are consistent, then \([s] \subseteq [S] \) holds and \( A' \) is consistent.

**Theorem 3** (Completeness). Suppose \( A_i, s \sim S \sim A', \perp \). If \( A \) is consistent, then \([s'] \subseteq [S'] \) holds and \( A' \) is consistent.
5 Related Work

Since the tree automata containment problem is a key to XML type-checking, several authors have investigated various techniques for practical algorithms. As mentioned in Introduction, Hosoya, Vanillon, and Pierce have proposed a top-down algorithm [9]. Our algorithm in the present paper is constructed on top of theirs and adds an improvement to avoid backtracking by keeping track of dependencies among containments to be checked. Frisch has also pursued for improving the top-down algorithm [7]. He has pointed out that functional data structures used in the original top-down algorithm for performing backtracking can actually be replaced by destructive data structures (a hash table with a stack). Our experiments implement an algorithm incorporating his remark and confirmed that this replacement can indeed contribute to the efficiency (Sec. 6).

Frisch has also proposed a different approach [7] that, given a containment to check, generates a set of certain forms of constraints and delegates this to a local constraint solver [4]. An empirical comparison with this algorithm is still planned.

Another completely different approach has been investigated by Tozawa and Hagiya [17]. They have used binary decision diagrams (BDDs) for representing sets of sets of states of given tree automata and solving the containment problem by using a series of operations on BDDs. Their algorithm behaves in a bottom-up way and therefore some combinations of states may potentially be examined even when it is not needed. Nevertheless they have given some experimental results that indicate a potential advantage over the top-down approach.

6 Experiments

In our preliminary experiments, we compare the following three algorithms.

NonBack our non-backtracking top-down algorithm

Origin Hosoya-Vanillon-Pierce’s original top-down algorithm [9] using functional data structures, maintaining no dependencies of containments or failed containments

Stack Frisch’s version of top-down algorithm using destructive data structures (mentioned in the previous section), maintaining only failed containments (no dependencies of containments)

We have experimented on the third algorithm in order to see whether the cheaper optimization suggested by Frisch can compete with our rather involved treatment. We have used seven examples (explained below) as inputs to the algorithms and measured the amortized elapsed time that each algorithm takes for each example. We have also counted the number of times that states were unfolded by the transition function during the check. The experiment has been done in the following environment: Intel Mobile PentiumIII 700MHz with 256 megabytes memory under Linux (kernel version 2.4.7). The result is shown in Fig. 2.

In each of the first four examples, we take a small but non-trivial program written in XDave and measure the time spent for checking all the containments
needed in typechecking the program. Although these programs are not "real" applications, these contain typical patterns of XML programming and some of these (bookmarks and html2latex) use a relatively large schema, i.e., XHTML, and hence experimenting on these is a meaningful benchmark. The result shows that our non-backtracking algorithm works about 3 times to 8 times faster than the original algorithm. It also shows that the running time is similar in our algorithm and in Frisch's version except that ours is about twice faster in html2latex. Hence, it indicates that the cheaper optimization can be enough for many cases.

In the next example (complex_pat), we examine the containment between $A$ and $B_1 \ldots B_n$ where $A, B_1, \ldots, B_n$ are defined as follows using the notation of regular expression types [9].

\[
\begin{align*}
A & = a[T_1 \ast], a[T_2 \ast], \ldots, a[T_n \ast] \\
B_i & = a[T_1 \ast], \ldots, a[S_i], a[T_{i+1} \ast], \ldots, a[T_n \ast] \quad (i = 1, \ldots, n - 1) \\
B_n & = a[T_1 \ast], \ldots, a[T_{n-1} \ast], a[S_n \ast] \\
T_i & = b_i[X] \\
S_i & = b_i[Y]
\end{align*}
\]

($B_n$ has an additional $\ast$ in the last label whereas the other $B_i$'s does not. This is for ensuring the containment to hold.) In the above, $X$ and $Y$ are defined as some types (not shown here) where the containment between $X$ and $Y$ holds, but its check needs a large computation. Although this example itself is not taken from a real application, a similar one could appear in checking a pattern match (in the style of XDuce) that extracts, from a given sequence of a fixed length, the first element whose content matches a particular type. The result in Fig. 2 shows that, for the case $n = 6$, our algorithm runs 6 times more efficiently than Frisch's version, and in fact, the ratio is proportional to $n$. Hence, this result implies that, only with his simple optimization techniques, the algorithm can still behave in a catastrophic manner in some cases.

In the final example (docbook), we perform a single containment check between version 2 and version 4 of the DocBook schema [13]. This series of schemas is one of the largest popular schemas for XML and hence is a challenging example for the containment algorithms. The result is that the original top-down algorithm could not finish checking in a reasonable amount of time whereas the other algorithms finish in about 1 second. Frisch's algorithm is about 70% faster than ours. However, note that the numbers of unfoldings are exactly the same.
for these. Since, from the way the algorithm is constructed, the number of unfoldings is always equal or smaller in our algorithm and since the additional overhead is about 70% even in such a large example, we can expect that the relative slowdown can be bounded by 70% in almost any situation. Considering that there are cases where we can save the algorithm from a catastrophic slowdown, we believe that this overhead is acceptable in practice.

Acknowledgment We would like to express our deepest gratitude to Susumu Nishimura for his advice in improving the presentation of this paper. We also thank Alain Frisch for his precious comments and suggestions. This work was partly supported by The Inamori Foundation and Japan Society for the Promotion of Science.

References