XML Schema Containment Checking Based on Semi-implicit Techniques

Akihiko Tozawa[1] and Masami Hagiya[2]

[1] IBM Research, Tokyo Research Laboratory, IBM Japan ltd., Japan
atozawa@trl.ibm.com
[2] Graduate School of Information Science and Technology
University of Tokyo, Japan
hagiya@is.s.u-tokyo.ac.jp

Abstract. XML schemas are computer languages defining grammars for XML (Extensible Markup Languages) documents. Containment checking for XML schemas has many applications, and is thus important. Since XML schemas are related to the class of tree regular languages, their containment checking is reduced to the language containment problem for non-deterministic tree automata (NTAs). However, an NTA for a practical XML schema has 10^2-10^3 states for which the textbook algorithm based on naive determinization is expensive. Thus we in this paper consider techniques based on BDDs (binary decision diagrams). We used semi-implicit encoding which encodes a set of subsets of states as a BDD, rather than encoding a set of states by it. The experiment on several real-world XML schemas proves that our containment checker can answer problems that cannot be solved by previously known algorithms.

1 Introduction

This paper discusses the containment checking for XML schemas, which is essentially the problem of the containment checking between two non-deterministic tree automata (NTAs). We use reduced and ordered BDDs (ROBDDs) to solve this problem.

In this paper, we refer to a standard bottom-up automaton on binary trees as *NTA*. In the problems of our interest, each NTA involves 10^2-10^3 states. Although this is not a large number for word automata, the situation is different with NTA. For instance, the determinization of NTAs, which we may use to compute the complement automaton, is often very expensive. Thus the textbook algorithm for automata containment does not work as it is. On the other hand, our containment algorithm uses semi-implicit techniques. That is, we do not perform determinization explicitly but rather we do so by encoding a set of subsets of the state set of the NTA (= a set of states of the determinized automaton) with a BDD.

In *symbolic* (or *implicit*) *verification techniques* [CGP99], usually a set of states is encoded by a single BDD, that is, we use a bit vector encoding for each state. In contrast, our technique is called *semi-implicit*, since we do not encode

O.H. Ibarra and Z. Dang (Eds.): CIAA 2003, LNCS 2759, pp. 213–225, 2003.

each state, but rather we encode each subset of the state set, and thus each BDD represents a set of such *subset*s. Semi-implicit techniques are not used in previous work on the language containment, since they are considered not to be efficient. However, our algorithm is efficient. The reason for this efficiency lies in the use of *positive* BDDs. A positive BDD represents an upward closures of a set of subsets. It is safe to use positive BDDs because each subset introduced in the upward closure is always a weaker condition with respect to whether or not the containment holds. By restricting BDDs to positive ones, we further reduce the number and size of BDDs appearing in the analysis.

Background

Our interests in NTAs come from their relation to XML schemas, i.e., grammar specification languages for XML (Extensible Markup Language). For example, a grammar for XHTML (whose complete version is defined by W3C [Wor00]) is approximately as follows:

$$
\begin{aligned}
&S ::= \texttt{html}\langle \mathit{Head}, \mathit{Body}\rangle \\
&\mathit{Head} ::= \texttt{head}\langle \mathit{Meta}^*, \mathit{Title}, \mathit{Meta}^*\rangle \\
&\mathit{Body} ::= \texttt{body}\langle (\mathit{H1}\,|\,\mathit{H2}\,|\,\ldots.)^*\rangle \\
&\ldots
\end{aligned}
$$

The grammar means that we at top have an `html`-node, in which, i.e., between <`html`> and </`html`> tags, we have `head` and `body` nodes, ... and so on. The class of languages that grammars as above define is identical to the class of regular tree languages [HM02, HVP00, MLM01]. The software that checks if XML documents comply with a schema is called a *validator*. A validation corresponds to the execution of a tree automata in correspondence.

The ultimate goal of our study is to develop efficient and innovative tools and softwares for XML and its schemas. Currently only established technologies for XML schemas are validators, but there are other applications. For some applications, it is important to investigate problems with higher complexity. For example, type checkers for XML processing languages [CMS02, HVP00] often use set operations on types which are essentially boolean operations on automata. In particular, they usually define subtyping relation between two types by means of the language containment testing for NTAs, which in the worst case requires EXPTIME to the size of states [Sei90].

Outline

In the next section, we overview the related work both from verification technologies and XML technologies. Sections 3 describes preliminary definitions and concepts. In Section 4, we discuss a containment algorithm for NTAs. We also show some experimental results on examples including those from real-world XML schemas. Section 5 discusses the future work.

2 Related Work

Automata and BDDs

There have been several analyses of automata based on BDDs in the context of verification technology. In this context, automata are on infinite objects. Existing symbolic algorithms for ω-automata containment either restrict automata to deterministic ones as in Touati, et al. [TBK95] (so that there is a linear-size complement automaton), or use intricate BDD encoding as in Finkbeiner [Fin01] and Tasiran, et al. [THB95]. None of these algorithms use semi-implicit encoding similar to ours. This may be because their automaton usually represents a Cartesian product of concurrent processes, where the number of states grows exponential to the number of processes. The large number of states makes the semi-implicit encoding almost impossible. This problem does not apply to our automata modeling XML schemas.

Mona's approach [HJJ+95] is also based on BDDs. In Mona, transition functions are efficiently expressed by a set of multi-terminal BDDs each corresponding to a source state. The nodes of each BDD represent boolean vector encoding of labels of transitions and its leaves represent the target states. We have not tested whether this encoding is also valid with problems of our interests. They also extended their representation to deal with NTAs [BKR97].

NTA Containment

Hosoya, et al. proposed another NTA containment algorithm [HVP00] which is one of the best algorithms that can be used in a real-world software. Hosoya, et al's algorithm is based on the search of proof trees of the co-inductively defined containment relation. The algorithm is explicit, i.e., it does not use BDDs. It proceeds from the final states of bottom-up tree automata to the initial states. We later compare our algorithm to theirs.

Kuper, et al. [KS01] proposed the notion of *subsumption* for XML schemas. Subsumption, based on the similarity relation between two grammar specifications, is a strictly stronger relation than the containment relation between two languages. We do not deal with subsumption in this paper.

Shuffle and Other Topics of XML Schemas

The word *XML schema* is not a proper noun. Indeed, we have a variety of XML schema languages including DTD, W3C XML Schema, RELAX NG [Ora01], DSD [KSM02], etc. The result of this paper can directly be applied to schemas that can easily be transformed into NTAs. Such schemas include regular expression types [HVP00], RELAX [rel] and DTDs.

Hosoya, et al. are working on containment of shuffle regular expressions and they have some preliminary unpublished results. Hosoya and Murata [HM02] also proposed a containment algorithm for schemas with attribute-element constraints. We do not discuss shuffle regular expressions and attribute-element

constraints in this paper, but they are also important in XML schema languages such as RELAX NG [Ora01]. We just mention the result of Mayer and Stockmeyer [MS94] stating that the containment of shuffle regular expressions is EXPSPACE-complete. This means the problem of containment for RELAX NG is essentially harder than the problem dealt with in this paper.

3 Preparation

3.1 Binary Trees

As noted in the introduction, we model XML schemas defining a set of XML documents by automata defining a set of binary trees. Indeed, we can view an XML document instance as a binary tree. To clarify the relationship between binary trees and XML trees, we here introduce a special notation for binary trees[1].

Throughout the paper, we use $u, v, w \ldots$ to range over a set of binary trees. In this paper, a binary tree v on the alphabet Σ (we use a to range over Σ) is represented by a sequence

$$a_0\langle v_0\rangle a_1\langle v_1\rangle \cdots a_{n-1}\langle v_{n-1}\rangle$$

of *nodes* $a_i\langle v_i\rangle$, where each a_i is a *label* and each v_i is a *subtree*. We have no restriction on the number n of nodes in a sequence. We use ϵ to denote a null tree ($n = 0$). If $n > 0$, we split a sequence v into the first node $a\langle u\rangle$ ($= a_0\langle v_0\rangle$) and the remainder w ($= a_1\langle v_1\rangle \cdots a_{n-1}\langle v_{n-1}\rangle$). Thus a sequence v is either in the form $a\langle u\rangle w$ or ϵ, i.e., it is a binary tree.

3.2 ROBDD

We use *reduced* and *ordered* BDDs (ROBDDs) by Bryant [Bry86]. Each BDD over a variable set X represents a boolean function over X. A boolean function takes an assignment of a truth value (**0** or **1**) to each variable, and returns again a truth value. Note that each assignment also represents a subset of X such that the subset consists of variables to which **1** is assigned. Thus, a boolean function also represents a set of subsets, i.e., an element of 2^{2^X}, such that each subset represents an assignment which gives the return value **1**.

Let $(X, <)$ be an arbitrary linearly ordered set, and x be an element of X. A BDD α is defined as follows:

$$\alpha ::= (x, \alpha, \alpha) \mid \mathbf{0} \mid \mathbf{1}$$

If α is a *node* (x, β, γ), we can use the notation $\alpha.var = x$, $\alpha.l = \beta$ and $\alpha.h = \gamma$ to obtain the content of α [2]. Otherwise α is called a *leaf*, i.e., **0**-*leaf* or **1**-*leaf*.

In this paper, we interpret the semantics of a BDD as a set of subsets.

[1] This notation defines what is often called *hedges* [MLM01].

[2] According to the BDD vocabulary, they are the *variable*, the node on a *low edge*, and the node on a *high edge* of α, respectively.

Definition 1. *A BDD α represents a set $[\![\alpha]\!]$ of subsets of X:*

$$[\![\mathbf{0}]\!] = \emptyset,$$
$$[\![\mathbf{1}]\!] = 2^X,$$
$$[\![(x, \alpha, \beta)]\!] = \{S \mid S \in [\![\alpha]\!],\ x \notin S\} \cup \{S \mid S \in [\![\beta]\!],\ x \in S\}$$

For example, $(x, \mathbf{0}, \mathbf{1})$ denotes a set of S such that $x \in S$ and $(x, \mathbf{1}, \mathbf{0})$, a set of S such that $x \notin S$. These two correspond to boolean functions $f(x_1, .., x_n) = x_k$ and $f(x_1, .., x_n) = \neg x_k$, respectively, where $X = \{x_1, .. x_k, .. x_n\}$ and $x = x_k$. We say a BDD is *ordered* if all variables appear to the given linear order $<$, i.e., for any subnode (x, α, β), each variable y appearing in α or β satisfies $x < y$. We say a BDD is *reduced* if there are no subnodes in the form (x, α, α) [3]. A BDD is an ROBDD if it is reduced and ordered. Each ROBDD is canonical, meaning that for each element of 2^{2^X}, there is one and only one ROBDD that represents it. In this paper, we denote a set of ROBDDs over X by $\mathsf{ROBDD}(X)$.

We use two standard ROBDD operations, $\alpha \wedge \beta$ such that $[\![\alpha \wedge \beta]\!] = [\![\alpha]\!] \cap [\![\beta]\!]$, and $\alpha \vee \beta$ such that $[\![\alpha \vee \beta]\!] = [\![\alpha]\!] \cup [\![\beta]\!]$. Further definitions of these operations of BDDs can be found in Appendix A.

We denote by the node size of a BDD, the number of syntactically distinct subnodes in the BDD. The node size of a BDD is often substantially smaller than the cardinality of the underlying set that this BDD represents.

4 Containment of Non-deterministic Tree Automata

4.1 Non-deterministic Tree Automata

A non-deterministic tree automaton A is a tuple $\langle \Sigma_A, Q_A, \delta_A, I_A, F_A \rangle$, where Σ_A is an alphabet, Q_A is a state set, δ_A is a transition function that maps $\Sigma_A \times Q_A \times Q_A$ to 2^{Q_A}, I_A is an initial state set and F_A is a final state set. Σ_A will not change throughout the paper and is denoted by Σ.

Definition 2. *The language $[\![q]\!]_A$ corresponding to each state q of A is a set of binary trees defined by the following inductive definition:*

- *For $q \in I_A$, we have $\epsilon \in [\![q]\!]_A$.*
- *If $v_1 \in [\![q_1]\!]_A$, $v_2 \in [\![q_2]\!]_A$ and $q_0 \in \delta_A(a, q_1, q_2)$, we have $a\langle v_1 \rangle v_2 \in [\![q_0]\!]_A$.*

An NTA A accepts a binary tree v, if v is included in $[\![q]\!]_A$ for a certain final state q. The set of binary trees accepted by A is called the language of A and denoted by $[\![A]\!]$.

$$[\![A]\!] = \bigcup_{q \in F_A} [\![q]\!]_A$$

[3] Usually reducedness also requires node-sharing in a graph representing BDD structures. In our definition, we distinguish BDDs up to their syntactical equalities, and thus subnodes are already shared.

NTAs define an identical class of languages as regular expression types by Hosoya et al [HVP00]. This class of languages is called *regular*. In implementation, we used Hosoya et al's algorithm that converts XML schemas into tree automata.

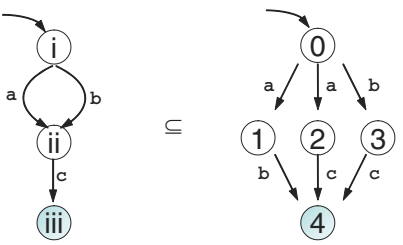
Example 1. An automaton A, such that $Q_A = \{0,1,2\}$, $\delta_A(\mathtt{r},1,0) = \{2\}$, $\delta_A(\mathtt{a},0,1) = \{1\}$, $I_A = \{0,1\}$ and $F_A = \{2\}$, accepts any binary tree of the form $\mathtt{r}\langle\mathtt{a}\langle\rangle^n\rangle$ $(n \geq 0)$.

4.2 Highlight

We propose an algorithm that given two tree automata A and B, decides if $[\![A]\!] \subseteq [\![B]\!]$ holds or not.

In a traditional algorithm, we complement the automaton at the right hand side, and compute the difference automaton for $A \cap \bar{B}$. In complementation, we determinize B using subsets of Q_B as new states. To get the difference, starting from each pair of an initial state of A and the initial state set of B, we enumerate all reachable state pairs in $Q_A \times 2^{Q_B}$ according to a transition in A, and a corresponding set of transitions in B using the same label. At the final step, if each state pair (q, S) having q in F_A satisfies $S \cap F_B \neq \emptyset$, i.e., S is *not* a final state of $\bar{B}$, the automaton $A \cap \bar{B}$ is empty, and thus the containment holds.

Our algorithm can similarly be applied to word automata. For simplicity, we here use word automata as an example. To check containment between two automata A and B for regular expressions $(\mathtt{a}|\mathtt{b})\mathtt{c}$ and $\mathtt{ab}|\mathtt{ac}|\mathtt{bc}$,



we enumerate these pairs in $Q_A \times 2^{Q_B}$.

$$(\mathrm{i}, \{0\}), (\mathrm{ii}, \{1,2\}), (\mathrm{ii}, \{3\}), (\mathrm{iii}, \{4\}).$$

Note that we have to enumerate *both* $(\mathrm{ii}, \{1,2\})$ *and* $(\mathrm{ii}, \{3\})$. Since the final state iii of A is associated only with $\{4\}$ which is a final state of B, the containment holds.

Enumeration of subsets of states in B may, however, involve an explosion. As we will see later, this explosion is a serious problem when the algorithm is applied to tree automata. We use two techniques in order to suppress an explosion: (1) using ROBDD representation for counting subsets, and (2) recording not exact sets of subsets but their upward closures.

First, we reduce the representation of a set of subsets by using ROBDDs. The algorithm uses the following data structure.

$$D \in Q_A \mapsto \mathsf{ROBDD}(Q_B)$$

Each entry $D(q)$ is a set including subsets of Q_B associated with q. When we check the above containment problem, the contents of D are as follows

$$D(\mathrm{i}) = \hat{0}$$
$$D(\mathrm{ii}) = (\hat{1} \wedge \hat{2}) \vee \hat{3}$$
$$D(\mathrm{iii}) = \hat{4}$$

where $\hat{x}$ denotes $(x, \mathbf{0}, \mathbf{1})$.

Second, $D(q)$ does not record exact subsets that can be reached, but it also represents their arbitrary supersets. For example, $[\![D(\mathrm{i})]\!] = [\![\hat{0}]\!]$ represents all sets that subsume $\{0\}$, and $[\![D(\mathrm{ii})]\!] = [\![(\hat{1} \wedge \hat{2}) \vee \hat{3}]\!]$ represents sets that subsume either $\{1, 2\}$ or $\{3\}$. It is safe to do so because if there is S in $[\![D(q)]\!]$ (e.g., $\{4\} \in [\![D(\mathrm{iii})]\!]$), the presence or absence of S' such that $S \subseteq S'$ (e.g., $S' = \{0, 4\}$, $\{0, 1, 4\}$, etc.) does not affect the result of analysis. Thus, we can freely add S' to $[\![D(q)]\!]$ if there is already S. More specifically, we can observe the following two properties.

- If (r, T') is reached from (q, S'), there is (r, T) reached from (q, S) such that $T \subseteq T'$.
- For $q \in F_A$, if $S \cap F_B \neq \emptyset$, then so with S', i.e., $S' \cap F_B \neq \emptyset$.

The first property states that when we compute $[\![D(r)]\!]$ from $[\![D(q)]\!]$, even if we add or ignore S' in $[\![D(q)]\!]$ (e.g., $\{0, 1\}$ in $[\![D(\mathrm{i})]\!]$), we do neither gain nor lose information in $[\![D(r)]\!]$ up to its upward closures (e.g., using a label `b`, we can compute $\{3, 4\}$ for $[\![D(\mathrm{ii})]\!]$, but this is meaningless where there is $\{3\}$). The second property states that it is also safe to add or ignore S' in the final step of the analysis (recall that in this step we check if all S satisfy $S \cap F_B \neq \emptyset$). Therefore, the result of the containment check depends only on the subset S but not on any upward subset S'.

Fortunately, an ROBDD nicely represents such an upward closure. A *positive* ROBDD corresponds to a boolean formula without negative occurrence of variables. Such an ROBDD is exactly what we want. The use of positive BDDs will further reduce the complexity of the analysis.

4.3 Algorithm

Given the data structure D, it remains to explain how to compute a transition for each entry in D and propagate the result to the entry for the next state. The algorithm is not efficient if such a transition can only be done by explicitly counting elements in each $[\![D(q)]\!]$. In out algorithm, however, we rather compute transitions symbolically on each ROBDD representation.

We write $\delta(a, S, T)$ to denote a union of images $\bigcup\{\delta(a, q_1, q_2) \mid q_1 \in S,\ q_2 \in T\}$. Formally, we need a function that computes the set of unions of images $\{S \mid \delta(a, T, U) \subseteq S,\ T \in [\![\alpha]\!],\ U \in [\![\beta]\!]\}$ of δ taking a triple of a and two *sets of sets*, α and β, as arguments. This function is encoded by a simple function tr in Figure 1 (which uses two ROBDD operations $\wedge$ and $\vee$ inside).

Using function tr, we compute D as follows:

$$
\begin{array}{l}
tr \in \Sigma \times \mathsf{ROBDD}(Q_B) \times \mathsf{ROBDD}(Q_B) \to \mathsf{ROBDD}(Q_B) \\
tr' \in \Sigma \times Q_B \times \mathsf{ROBDD}(Q_B) \to \mathsf{ROBDD}(Q_B) \\
tr'' \in \Sigma \times Q_B \times Q_B \to \mathsf{ROBDD}(Q_B)
\end{array}
$$

$$
\begin{array}{ll}
tr(a,\alpha,\beta) = \mathbf{0} & (\text{if } \alpha = \mathbf{0}) \\
tr(a,\alpha,\beta) = \mathbf{1} & (\text{if } \alpha = \mathbf{1}) \\
tr(a,\alpha,\beta) = tr(a,\alpha.l,\beta) & \\
\quad \vee (tr(a,\alpha.h,\beta) \wedge tr'(a,\alpha.var,\beta)) & \\
 & (\text{otherwise}) \\
tr'(a,x,\beta) = \mathbf{0} & (\text{if } \beta = \mathbf{0}) \\
tr'(a,x,\beta) = \mathbf{1} & (\text{if } \beta = \mathbf{1}) \\
tr'(a,x,\beta) = tr'(a,x,\beta.l) & \\
\quad \vee (tr'(a,x,\beta.h) \wedge tr''(a,x,\beta.var)) & \\
 & (\text{otherwise}) \\
tr''(a,x,y) = \bigwedge_{z \in \delta_B(a,x,y)} (z, \mathbf{0}, \mathbf{1}) &
\end{array}
$$

Fig. 1. Function tr

– Initialize D_0:

$$
\begin{array}{ll}
D_0(q) = \bigwedge_{q' \in I_B} (q', \mathbf{0}, \mathbf{1}) & (q \in I_A) \\
D_0(q) = \mathbf{0} & (\text{otherwise})
\end{array}
\tag{1}
$$

– At i-th step, update D_i for each $q_0, q_1, q_2 \in Q_A$ and $a \in \Sigma$ such that $q_0 \in \delta_A(a, q_1, q_2)$:

$$
D_{i+1}(q_0) = D_i(q_0) \vee tr(a, D_i(q_1), D_i(q_2)) \tag{2}
$$

Repeat until $D_i = D_{i+1}$ $(= D)$ holds (fixpoint computation).

After all, we have

Theorem 1.

$$
[\![A]\!] \subseteq [\![B]\!] \Leftrightarrow \bigvee_{q \in F_A} D(q) \wedge \bigwedge_{q \in F_B} (q, \mathbf{1}, \mathbf{0}) = \mathbf{0}.
$$

We check the right hand side of $\Leftrightarrow$ using ROBDD operations. If it succeeds, the containment holds. The proof of the theorem can be found in Appendix B.

4.4 Experiments

We implemented the algorithm described so far. For comparison, we also implemented other algorithms in the literature.

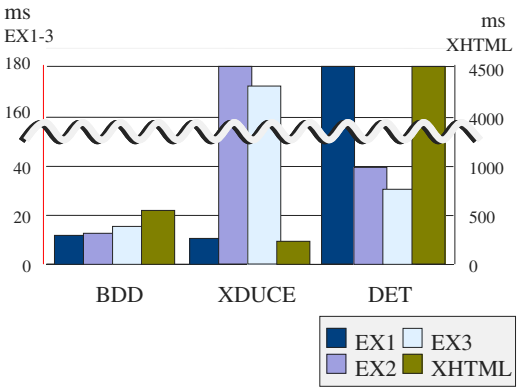


Fig. 2. Experimental Results (J2RE IBM JIT-enabled on Windows, Pentium III, 1GHz, 384MB memory)

BDD An ROBDD-based algorithm.
DET A textbook algorithm that involves determinization [4].
XDUCE An XDuce's algorithm by Hosoya, et al. [HVP00], which was originally implemented in Ocaml. We have re-implemented it in Java.

In implementation, we used our own ROBDD package in Java. We do not follow a particular variable ordering heuristics, but we rather used a random ordering. Investigation of variable ordering is left for future work.

Figure 2 shows the performance of each algorithm on the following examples:

EX1 This example has an expression of the form $(a\langle\rangle|b\langle\rangle)^{15}a\langle\rangle(a\langle\rangle|b\langle\rangle)^*$ on the right hand side of containment. This regular expression is famous as its conversion to DFA from right to left leads to exponential blow up.
EX2 The reverse of EX1. Determinizing $(a\langle\rangle|b\langle\rangle)^*a\langle\rangle(a\langle\rangle|b\langle\rangle)^{15}$ from left to right blows up states. The DET algorithm is not affected, which determinizes from right to left, while the XDUCE algorithm is affected.
EX3 Another example that XDUCE cannot handle, including $a\langle b_1\langle\rangle\rangle b_1\langle\rangle|...$ $|a\langle b_{15}\langle\rangle\rangle b_{15}\langle\rangle$ on the right hand side.
XHTML A real-world example. We check that "xhtml1-transitional.dtd" contains "xhtml1-strict.dtd" [Wor00].

XDUCE is one of the best known algorithms but still causes blow-up as in EX2 and EX3. Our algorithm performs good in general.

[4] It is known that if we restrict ourselves to DTDs, there is a more efficient algorithm for containment test. However this algorithm cannot be applied to other schema languages having the same expressive power as NTAs. Thus we here used a naive algorithm for containment of NTAs in general.

In some cases where the containment test fails, XDUCE can detect the problem very early. This is because XDUCE is a top-down algorithm, and problems are likely to be found near the roots of trees, i.e., more accurately, near the final states of tree automata. To simulate this early failure detection, we have to check if $D_i(q) = \mathbf{1}$ or not at each i-th step for each q which is *useful*, i.e., there is a transition from q that reaches a final state. Once there is such q, the test always fails.

5 Concluding Remark

This line of research aims at two major applications. One application is an XML schema version check tool based on the containment algorithm. As we noted earlier, the containment for some XML schemas such as RELAX NG [Ora01] is more difficult than what we have done in this paper. We are seeking the extension of our algorithm to do with these XML schemas. The other application is a type-checker of XML processing languages with types based on XML schemas. We are currently developing a typed XML processing language using the proposed NTA containment algorithm, which is released from IBM alphaWorks.

Acknowledgement

We would like to thank Bernd Finkbeiner for valuable discussions on related topics, and we also thank Makoto Murata for a lot of helpful suggestions.

References

[BKR97] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, WIA '96, London, Ontario, Canada, LNCS 1260*. Springer Verlag, 1997. 215

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions and Computers*, C-35(8):677–691, August 1986. 216

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT press, 1999. 213

[CMS02] Aske Simon Christensen, Anders Muller, and Michael I. Schwartzbach. Static analysis for dynamic XML. In *Proceedings of 1st Workshop on Programming Languages Technology for XML (PLAN-X 2002)*, 2002. 214

[Fin01] Bernd Finkbeiner. Language containment checking with nondeterministic BDDs. In *7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *LNCS*, pages 24–38, 2001. 215

[HJJ+95] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *LNCS*, pages 89–110. Springer, 1995. 215

[HM02] Haruo Hosoya and Makoto Murata. Validation and boolean operations for attribute-element constraints. In *Proceedings of 1st Workshop on Programming Languages Technology for XML (PLAN-X 2002)*, 2002. 214, 215

[HVP00] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, Sep., 2000. 214, 215, 218, 221

[KS01] G. M. Kuper and J. Simeon. Subsumption for XML types. In *Proceedings of International Conference on Database Theory (ICDT)*, Jan., 2001. 215

[KSM02] Nils Klarlund, Michael I. Schwartzbach, and Anders Møller. The DSD schema language. *Automated Software Engineering Journal, to appear*, 2002. 215

[MLM01] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML schema languages using formal language theory. In *Proceedings of Extreme Markup Language 2001, Montreal*, pages 153–166, 2001. 214, 216

[MS94] Alain J. Mayer and Larry J. Stockmeyer. The complexity of word problems–this time with interleaving. *Information and Computation*, 115(2):293–311, 1994. 216

[Ora01] Oraganization for Advancement of Structured Information Standards (OASIS). RELAX NG, 2001. `http://www.oasis-open.org/committees/relax-ng/`. 215, 216, 222

[rel] RELAX (REgular LAnguage description for XML). `http://www.xml.gr.jp/relax/`. 215

[Sei90] Hermut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990. 214

[TBK95] Hervé J. Touati, Robert K. Brayton, and Robert Kurshan. Testing language containment for ω-automata using BDDs. *Information and Computation*, 118(1):101–109, April 1995. 215

[THB95] S. Tasiran, R. Hojati, and R. K. Brayton. Language containment using non-deteministic omega-automata. In *Proc. of CHARME'95*, volume 987 of *LNCS*. Springer-Verlag, 1995. 215

[Wor00] World Wide Web Consortium. XHTML1.0, 2000. `http://www.w3.org/TR/xhtml`. 214, 221

A BDD

In Figure 3, we summarize the standard operations on ROBDDs. Function *rd* is called reducer functions which guarantee that resulting BDDs are reduced. Note that it is not efficient if these operations are implemented as is. In implementation, we use two kinds of hash tables (1) in order to use a unique pointer to refer to each syntactically equivalent BDD, and (2) in order to implement $\wedge$ and $\vee$ as *memoise* functions.

B Proof

In this section, we describe the proof of the theorem in Section 4. The algorithm terminates as there are finitely many possibilities for D_i. Here we show its correctness.

$$
\begin{array}{l}
rd(x,\alpha,\beta) = \alpha \qquad\qquad (\alpha = \beta) \\
rd(x,\alpha,\beta) = (x,\alpha,\beta) \ (\text{otherwise})
\end{array}
$$

$$
\begin{array}{l}
\mathbf{0} \wedge \alpha = \alpha \wedge \mathbf{0} = \mathbf{0} \\
\mathbf{1} \wedge \alpha = \alpha \wedge \mathbf{1} = \alpha \\
\alpha \wedge \beta = \\
\left\{ \begin{array}{ll}
rd(\alpha.var,\ \alpha.l \wedge \beta,\ \alpha.h \wedge \beta) & (\alpha.var < \beta.var) \\
rd(\alpha.var,\ \alpha.l \wedge \beta.l,\ \alpha.h \wedge \beta.h) & (\alpha.var = \beta.var) \\
rd(\beta.var,\ \alpha \wedge \beta.l,\ \alpha \wedge \beta.h) & (\alpha.var > \beta.var)
\end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
\mathbf{0} \vee \alpha = \alpha \vee \mathbf{0} = \alpha \\
\mathbf{1} \vee \alpha = \alpha \vee \mathbf{1} = \mathbf{1} \\
\alpha \vee \beta = \\
\left\{ \begin{array}{ll}
rd(\alpha.var,\ \alpha.l \vee \beta,\ \alpha.h \vee \beta) & (\alpha.var < \beta.var) \\
rd(\alpha.var,\ \alpha.l \vee \beta.l,\ \alpha.h \vee \beta.h) & (\alpha.var = \beta.var) \\
rd(\beta.var,\ \alpha \vee \beta.l,\ \alpha \vee \beta.h) & (\alpha.var > \beta.var)
\end{array} \right.
\end{array}
$$

Fig. 3. BDD operations

In preparation, we define the notion that BDD α is *positive* as follows.

$$\forall S \in [\![\alpha]\!].\ \forall T.\ S \subseteq T \Rightarrow T \in [\![\alpha]\!]$$

Assuming orderedness of α, we have α positive, iff, either $\alpha = \mathbf{0}$, $\alpha = \mathbf{1}$, or $\alpha = (x, \beta, \gamma)$ where β and γ are positive and satisfy $[\![\beta]\!] \subseteq [\![\gamma]\!]$. Positiveness is closed under $\wedge$ and $\vee$, and thus all BDDs appearing in the computation of D are positive.

First, we need the following lemma. For any positive α and β,

$$[\![tr(a,\alpha,\beta)]\!] = \{S \mid T \in [\![\alpha]\!],\ U \in [\![\beta]\!],\ \delta_B(a,T,U) \subseteq S\} \tag{3}$$

where $\delta_B(a,T,U) = \bigcup_{x \in T,\ y \in U} \delta_B(a,x,y)$. It is easy to see $[\![tr''(a,x,y)]\!] = \{S \mid \delta_B(a,x,y) \subseteq S\}$, by using which we inductively show that $[\![tr'(a,x,\beta)]\!] = \{S \mid U \in [\![\beta]\!],\ \delta_B(a,x,U) \subseteq S\}$ for any positive β:

Case ($\beta = \mathbf{0}$)**:** $[\![tr'(a,x,\beta)]\!] = \emptyset$.
Case ($\beta = \mathbf{1}$)**:** $[\![tr'(a,x,\beta)]\!] = 2^{Q_B}$. This is OK, since $\emptyset \in [\![\beta]\!]$.
Case (otherwise)**:** By induction,

$$
\begin{array}{l}
[\![tr'(a,x,\beta)]\!] \\
= [\![tr'(a,x,\beta.l) \vee (tr''(a,x,\beta.var) \wedge tr'(a,x,\beta.h))]\!] \\
= \{S \mid \exists U \in [\![\beta.l]\!].\ \delta_B(a,x,U) \subseteq S \\
\qquad\qquad \vee \exists T \in [\![\beta.h]\!].\ \delta_B(a,x,\beta.var) \cup\ \delta_B(a,x,T) \subseteq S\} \\
= \{S \mid \exists U \in [\![\beta.l]\!] \cup \{T \mid T \in [\![\beta.h]\!]\ \wedge\ \beta.var \in T\}.\ \delta_B(a,x,U) \subseteq S\} \\
= \{S \mid \exists U \in [\![\beta]\!].\ \delta_B(a,x,U) \subseteq S\}.
\end{array}
$$

The last rewrite follows from positiveness of β. Proving (3) from the above result is a similar work.

Second, we prove that for any $q \in Q_A$,

$$[\![D(q)]\!] = \{S \mid \exists v \in [\![q]\!]_A.\ \{q' \mid v \in [\![q']\!]_B\} \subseteq S\} \tag{4}$$

For ($\supseteq$), we show that $D(q)$ contains all possibilities. We prove that $\{q' \mid v \in [\![q']\!]_B\} \in [\![D(q)]\!]$ for any $v \in [\![q]\!]_A$ by induction on the definition of $[\![q]\!]_A$.

Case ($v = \epsilon$)**:** From (1), we have $I_B \in [\![D(q)]\!]$.

Case ($v = a\langle v_1\rangle v_2$)**:** There are q_1 and q_2 such that $q \in \delta(a, q_1, q_2)$ and $v_k \in [\![q_k]\!]_A$. The induction hypothesis guarantees $\{q' \mid v_k \in [\![q']\!]_B\} \in [\![D(q_k)]\!]$ ($k = 1, 2$). We use (3) to show that $[\![tr(a, [\![D(q_1)]\!], [\![D(q_2)]\!])]\!]$ ($\subseteq [\![D(q)]\!]$) contains $\delta_B(a, \{q' \mid v_1 \in [\![q']\!]_B\}, \{q' \mid v_2 \in [\![q']\!]_B\})$ ($= \{q' \mid a\langle v_1\rangle v_2 \in [\![q']\!]_B\}$).

For ($\subseteq$), we show that each $D_i(q)$ contains nothing too restrictive. We prove that for any $S \in [\![D_i(q)]\!]$, there exists $v \in [\![q]\!]_A$ such that $\{q' \mid v \in [\![q']\!]_B\} \subseteq S$.

Case ($i = 0$)**:** From (1).

Case (otherwise)**:** For each q_k and $S_k \in [\![D_{i-1}(q_k)]\!]$, let $v_k \in [\![q_k]\!]_A$ be a witness of the induction hypothesis, i.e., $\{q' \mid v_k \in [\![q']\!]_B\} \subseteq S_k$ ($k = 1, 2$). For any $S \in [\![tr(a, D_{i-1}(q_1), D_{i-1}(q_2))]\!]$, (3) implies that there are S_1 and S_2 such that $\{q' \mid a\langle v_1\rangle v_2 \in [\![q']\!]_B\}$ ($= \delta_B(a, \{q' \mid v_1 \in [\![q']\!]_B\}, \{q' \mid v_2 \in [\![q']\!]_B\})$) $\subseteq \delta_B(a, S_1, S_2) \subseteq S$. Therefore any S added to $D_i(q)$ by step (2) (where $q \in \delta_A(a, q_1, q_2)$) has a new witness $a\langle v_1\rangle v_2$ ($\in [\![q]\!]_A$).

Finally we prove Theorem 1 as follows.

$$\begin{aligned}
&\textstyle\bigvee_{q \in F_A} D(q) \wedge \bigwedge_{q \in F_B} (q, \mathbf{1}, \mathbf{0}) = \mathbf{0} \\
&\Leftrightarrow \{S \mid \exists q \in F_A.\ S \in D(q)\} \cap \{S \mid F_B \cap S = \emptyset\} = \emptyset \\
&\Leftrightarrow \forall q \in F_A.\ \forall S \in D(q).\ F_B \cap S \neq \emptyset
\end{aligned}$$

We use (4) and obtain

$$\begin{aligned}
&\Leftrightarrow \forall q \in F_A.\ \forall v \in [\![q]\!]_A.\ \forall S.\ \{q' \mid v \in [\![q']\!]_B\} \subseteq S \\
&\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow F_B \cap S \neq \emptyset \\
&\Leftrightarrow \forall q \in F_A.\ \forall v \in [\![q]\!]_A.\ F_B \cap \{q' \mid v \in [\![q']\!]_B\} \neq \emptyset \\
&\Leftrightarrow [\![A]\!] \subseteq [\![B]\!].
\end{aligned}$$

□