# Typechecking for XML Transformers*

Tova Milo
Tel Aviv University

Dan Suciu†
A.T.&T. Labs

Victor Vianu‡
U.C. San Diego

## Abstract

We study the typechecking problem for XML transformers: given an XML transformation program and a DTD for the input XML documents, check whether every result of the program conforms to a specified output DTD. We model XML transformers using a novel device called $k$-pebble transducer, that can express all queries without data-value joins in XML-QL, XSL, and other XML query languages. Types are modeled by regular tree languages, a robust extension of DTDs. The main result of the paper is that typechecking for $k$-pebble transducers is decidable. Consequently, typechecking can be performed for a broad range of XML transformation languages, including XML-QL and XSL, when reduced to their tree manipulation core.

## 1 Introduction

Traditionally, database query languages have focused on data retrieval, with complex data transformations left to applications. The new XML data exchange standard for the Web, and emerging applications requiring data wrapping and integration, have shifted the focus towards data transformations. A typical scenario is provided by a mediator system, where the integrated view is constructed from the data sources [GMPQ+97]. Several languages have been proposed to express XML transformations [DFF+99, Cla99b, Cla99a, Rob99].

XML data can have complex structure and flexible typing. Elements may be nested to arbitrary depth and new tags (element names) can be created at will. The structure of an XML document may be constrained by a Document Type Definition (DTD). An XML document which specifies a DTD and which also conforms to that DTD is called *valid*. The benefits of the information provided by DTDs are numerous. Some are analogous to those derived from schema information in relational query processing. Perhaps most importantly to the context of the Web, DTD can be used to validate data exchange. In a typical scenario, a user community would agree on a common DTD and on producing only XML documents which are valid with respect to the specified DTD. This raises the issue of *typechecking*, which comes in two flavors: dynamic, and static. Dynamic typechecking validates at runtime each XML document produced, and can be performed by a validating parser (several such parsers are publicly available). Its obvious drawback is that errors are only detected at runtime. In contrast, this paper studies the *static typechecking* problem (simply called *typechecking* from here on). Typechecking consists of verifying at compile time that *every* XML document which is the result of a specified transformation satisfies the output DTD. DTDs restricting the input documents may also be available as input to the typechecker. As we shall see, typechecking of XML transformations requires novel query analysis techniques.

Studying formally the typechecking problem is complicated by the fact that XML transformation languages and the notion of DTD are constantly evolving. More specifically:

- There is no generally accepted XML transformation language. Several transformation languages exist for XML, including XML-QL and XSL [DFF+99, Cla99b, Cla99a]. Other languages result from restricting query languages for semistructured data to trees (e.g. Lorel, StruQL, UnQL [MW99, FFK+98, BDHS96]). Yet more languages can be defined using algebras, motivated mainly by systems considerations (see [BT99]). These languages define incomparable classes of transformations, and it is not a priori clear how to define a single query language subsuming them, that could provide a relevant, broad framework for studying the typechecking problem.

- There is no generally accepted notion of schema for XML. DTDs are the most widely accepted

*Work supported in part by the U.S.-Israel Binational Science Foundation under grant number 97-00128

†Contact author address: AT&T Labs, 180 Park Avenue, Rm. E233, Florham Park, NJ 07932-0971, Phone: +1-973-360-8671 Fax: +1-973-360-8187, suciu@research.att.com.

‡This author supported in part by the National Science Foundation under grant number IIS-9802288.

because they are part of the existing XML standard. However, several variations and extensions of DTDs are being considered by the XML community. A recent effort tries to unify these extensions into a single framework called XML-Schema [BLM+99, BM99b]

We address these difficulties as follows:

- We define an abstract and rather general model of XML transformation languages called *k-pebble tree transducer*, expressing transformations on ordered, labeled trees. Up to $k$ pebbles can be placed on the input tree, moved up and down the tree, and removed, subject to a *stack discipline* where only the highest numbered pebble currently placed on the tree can be moved. Traditional top-down regular tree transducers [RS97] correspond to the particular case when $k = 1$ and the pebble is allowed to move only downwards. As we shall show, all transformations expressible in XML query languages (XML-QL, XSL, LOREL, UnQL, StruQL), without joins on data values, can also be expressed by $k$-pebble transducers. Therefore, the results on typechecking described in this paper also apply to these languages, and are likely to remain relevant as the languages evolve. In our view, the notion of $k$-pebble tree transducers is one of the main contributions of this work. Beyond their role in the study of typechecking, we believe they are of interest on their own as a simple, robust model for tree transformations. In particular, the transformations defined by $k$-pebble transducers have PTIME data complexity.
- As type formalism we adopt *regular tree languages*. A regular tree language is the set of trees accepted by a regular tree automaton (of which there are many equivalent variations, see [RS97] for the ranked case and [BKMW98] for the unranked case). It is easy to see that DTDs define a set of valid XML documents which (when viewed as labeled trees) form a regular tree language. Moreover, various proposed extensions of DTDs define regular tree languages as well; e.g., several such proposals decouple element names from tag names [BM99a, CDSS98, PV99b, LPVV99, PV99a]. It is easily seen that DTDs with decoupling define *precisely* the regular tree languages [BM99a, PV99b]. Note that regular tree languages cannot capture semantic constraints or describe unordered data, hence our results do not carry over to such schemas.

Formally, the typechecking problem studied in the paper is the following: given two types (regular tree languages) $\tau_1, \tau_2$, and a $k$-pebble transducer $T$, verify that $T(t) \subseteq \tau_2$ for every $t \in \tau_1$. [1] The main result of the paper is that typechecking is decidable. To understand our approach, consider first a closely related problem: type inference. In our framework, the type inference problem is the following: given a $k$-pebble transducer $T$ and an input type $\tau_1$, construct a type $\tau_1'$ that describes precisely the image of $T$ on inputs satisfying $\tau_1$. If type inference were possible, it would solve the typechecking problem: to check conformance of $T$ to $\tau_2$ on inputs satisfying $\tau_1$, first infer $\tau_1'$, then check that $\tau_1' \subseteq \tau_2$ (inclusion of regular tree languages is decidable). Unfortunately, as discussed in the paper, type inference in the above sense is not possible because, even for simple XML queries, their image is not a regular tree language (and there is no smallest regular tree language containing the image). However, the failure of type inference for transformation languages has an interesting twist, which lies at the core of our approach: the *inverse* of any $k$-pebble transformation admits type inference. That is, given a $k$-pebble transducer $T$ and an *output* type $\tau$, the set of trees $t$ such that $T(t) \subseteq \tau$ is a regular tree language $\tau^{-1}$ which can be inferred[2] from $T$ and $\tau$. We refer to the problem of finding $\tau^{-1}$ given $T$ and $\tau$ as the *inverse type inference* problem. Its solution is the main result in our paper, and it implies immediately a type checking algorithm. The technical development leading to the solution of the inverse type inference problem has three main stages:

(i) define an acceptor variant of the $k$-pebble transducer, called *k-pebble tree automaton*;

(ii) show that for each $k$-pebble transducer $T$ and type $\tau$, the set $\{t \mid T(t) \subseteq \tau\}$ is recognized by some $k$-pebble automaton; and,

(iii) prove that every $k$-pebble automaton recognizes a regular tree language.

**Limitations** There are two limitations of the typechecking method described here. The first is that it does not apply to transformations expressed with joins on data values. All XML query languages mentioned here, except XSL, can express such joins with conditions of the form `x = y`, where `x, y` are variables bound to atomic values in the input XML data (`#PCDATA`, in XML terminology). The domain of such values is infinite (e.g. strings), while $k$-pebble transducers are defined only for trees with labels from a

[1]The transducer $T$ may be non-deterministic; $T(t)$ denotes the set of possible outputs of $T$ on input $t$.

[2]For simplicity, we sometimes blur the distinction between a regular tree language and its specification as a regular tree automaton.

finite alphabet (corresponding to the tags in XML). To model data values, we can extend $k$-pebble transducers to trees with leaves labeled from an infinite domain. If we allow the transducers to test only unary predicates on the infinite domain (such as `(x > 5)` or `(x like ''Smith'')`), then the same typechecking techniques still work. However, if we extend the $k$-pebble transducers with joins on these data values (i.e. the predicate $\mathtt{x} = \mathtt{y}$), then typechecking becomes undecidable[3]. Thus, in the case of XML-QL, Lorel, StruQL, and UnQL, the type checking technique described here applies only to queries without joins on data values. We briefly describe in Sec. 5 a simple idea by which one can apply these techniques to an important class of queries with data joins, but we leave a full treatment for future work. We should clarify however that some languages (Lorel, StruQL) frequently use joins on node oids to search for patterns in the input data: such joins are also of the form $\mathtt{x} = \mathtt{y}$, but with `x, y` bound now to nodes in the tree: these conditions *can* be expressed by $k$-pebble transducers, and, hence, typechecking works for these queries too. The case of XSL is easier since it cannot express joins of data values, hence the typechecking method described here applies to the full language.[4]

The second limitation is the complexity. A direct application of our method for typechecking $k$-pebble transducers results in complexity hyperexponential in $k$ (so non-elementary), due to a translation of a Monadic Second Order formula on trees into a tree automaton. We leave for future work the question of whether this upper bound can be improved. As discussed in the paper, there are many restricted cases of practical interest for which the complexity is reasonable.

**Related Work** Two kinds of tree transducers are described in [RS97]: top-down and bottom-up (called there *root-to-frontier* and *frontier-to-root*). Any top-down transducer can be simulated by a 1-pebble transducer with moves restricted to go downwards only. Interestingly, it is an open problem whether $k$-pebble transducers can simulate all bottom-up transducers (more on that below). Both top-down and bottom-up transducers are known to have the inverse type inference property: $K$-pebble transducers however can express transformations which are not expressible by either top-down or bottom-up transducers.

The $k$-pebble transducers generalize several formalisms considered in the literature. Aho and Ullman [AU71] introduce *tree-walk* automata. These devices have a single head which can move up and down the tree, starting from the root. The set of tree languages accepted by a tree-walk automata is included in the set of regular tree languages, but it is a long standing open problem whether the inclusion is strict [EHB99]. The question whether $k$-pebble transducers can simulate all bottom-up transducers can be reduced to this open problem (in fact, as we explain in the sequel, the two problems become equivalent, when $k = 1$). Note that for the case of strings, the analogous of tree-walk automata are precisely the two-way automata, which are known to express all regular languages.

Engelfriet et al. [EHB99] extend the tree-walk automata with marbles and one pebble, and prove that the new machines capture precisely the regular tree languages. Unlike pebbles, marbles are in infinite supply, but the input head is restricted to stay below the last marble placed. Our $k$-pebble automata (the acceptor version of the $k$-pebble transducer) can also capture all regular tree languages, but use *branching*, a different extension of the tree-walk automata.

String automata with a rather restricted form of $k$-pebbles are considered by Goberman and Harel [GH94]. They prove certain lower bounds in the gap of succinctness of the expressibility of such automata.

*Query automata* are described by Neven and Schwentick in [NS99]. This work was the first to use Monadic Second Order logic (MSO) to study query languages for XML. Query automata however differ significantly from $k$-pebble transducers: they take an XML input tree and return a set of nodes in the tree. By contrast a $k$-pebble transducer returns a new output tree. Several transformation languages which are abstractions of XML are studied in [MN99], and connections to extended tree-walking tree transducers with look-ahead are established. Various static analysis problems are considered, such as termination, emptiness, and usefulness of rules. It is also shown that ranges of the transducers are closed under intersection with *generalized DTDs* (defined by tree regular grammars).

In previous work [MS99] we studied type inference for XML-QL selection queries, focusing on the inference of all possible type assignments for the individual variables in the query; this is different from typechecking XML transformations considered here. In other work [PV99b] we solve the type inference for

[3] This follows from a straightforward reduction from the finite satisfiability problem for first order logic.

[4] In fact in the latest version XSL can express all computable functions on data values (`#PCDATA`) and, through this back door, can express more complex tree transformations than $k$-pebble transducers. In this paper we will use the name XSL to refer only to the fragment of the language without any operations on data values.

XML transformations under a relaxed output type and a restricted class of transformations: without these changes type inference does not admit a solution in general (Section 4).

The paper is organized as follows. Section 2 provides background material, including DTDs, regular tree languages, and regular tree automata. Section 3 introduces and illustrates the $k$-pebble transducer, and shows its PTIME data complexity. Our results on typechecking of $k$-pebble transducers are presented in Section 4, and some extensions are discussed in Sec. 5. We conclude in Section 6. An appendix contains proof sketches and some of the formal definitions.

# 2 Background

## 2.1 Trees

We begin with some basic definitions.

**Unranked trees** Assume a fixed finite alphabet $\Sigma$. Unranked trees over $\Sigma$ are trees with node labels from $\Sigma$, with no predefined bound on the number of children of each node. The trees we consider are ordered. We denote unranked trees ($T$) and forests ($F$) as expressions defined recursively as follows:

$$T ::= a(F), \quad a \in \Sigma; \qquad F ::= T, F$$
$$T ::= a(), \quad a \in \Sigma; \qquad F ::= T$$

We denote $U_\Sigma$ the set of unranked trees.

Given a tree $t$, a subexpression occurrence of type $T$ is called a *node*. The set of all nodes in $t$ is denoted *nodes*$(t)$; the root node is *root*$(t)$. Given a node $T = a(F)$, with $F$ a forest or $\varepsilon$, its *symbol* is $symbol(T) \stackrel{\text{def}}{=} a$, and its *children* are $children(T) \stackrel{\text{def}}{=} F$. We abbreviated a leaf node $a()$ with $a$ when no confusion arises.

A *path expression* is a string $w \in \Sigma^*$. The result of evaluating a path expression on a tree $T$ is a set of nodes in $T$, defined as follows:

$$\begin{aligned}
eval(\varepsilon, T) &\stackrel{\text{def}}{=} \emptyset \\
eval(a, T) &\stackrel{\text{def}}{=} \text{if } a = symbol(T) \text{ then } \{T\} \text{ else } \emptyset \\
eval(a.w, T) &\stackrel{\text{def}}{=} \text{if } a = symbol(T) \\
&\qquad \text{then } \textstyle\bigcup_{T' \in children(T)} eval(w, T') \text{ else } \emptyset
\end{aligned}$$

A *regular path expression* is a regular expression $r$ over $\Sigma$. The result of evaluating a regular path expression on a tree $T$ is $eval(r, T) \stackrel{\text{def}}{=} \bigcup\{eval(w, T) \mid w \in lang(r)\}$. Regular path expressions are used in all XML query languages to query the input tree.

**Ranked trees** Complete binary trees are ranked trees defined as follows. The alphabet $\Sigma$ is partitioned into $\Sigma = \Sigma_0 \cup \Sigma_2$. A node labeled with a symbol $a$ is a leaf, when $a \in \Sigma_0$, and has two children, when $a \in \Sigma_2$. We denote by $T_\Sigma$ the set of complete binary trees over $\Sigma$. Similarly, $T_\Sigma(X)$ denotes the set of ranked trees with "variables in $X$". Formally, $T_\Sigma(X) \stackrel{\text{def}}{=} T_\Omega$, where the alphabet $\Omega$ defined by $\Omega_0 \stackrel{\text{def}}{=} \Sigma_0 \cup X$, and $\Omega_2 \stackrel{\text{def}}{=} \Sigma_2$.

We will denote leaf symbols with $a_0, b_0, \ldots \in \Sigma_0$, while symbols for internal nodes with $a_2, b_2, \ldots \in \Sigma_2$. We omit the index when it is clear from the context, i.e. by writing $a, b, c \ldots$.

We encode unranked trees into complete binary trees over alphabet $\Sigma' \stackrel{\text{def}}{=} \Sigma \cup \{-, |\}$, with $-, |$ two new symbols. Define $\Sigma_0 \stackrel{\text{def}}{=} \{|\}$, and $\Sigma_2 \stackrel{\text{def}}{=} \Sigma \cup \{-\}$. The encoding is given by $encode : U_\Sigma \to T_{\Sigma'}$ (defined recursively with $encode_f$, the encoding of a forest):

$$\begin{aligned}
encode(a(F)) &\stackrel{\text{def}}{=} a(encode_f(F), |) \\
encode(a()) &\stackrel{\text{def}}{=} a(|, |) \\
encode_f(T, F) &\stackrel{\text{def}}{=} -(encode(T), encode_f(F)) \\
encode_f(T) &\stackrel{\text{def}}{=} encode(T)
\end{aligned}$$

For example $encode(a(b, b, c(d), e)) = a(-(b, -(b, -(c(-(d, |), |), -(e, |)))), |)$, (see Figure 1 in the Appendix).

There is a one-to-one label-preserving mapping between the nodes in $T$ and nodes in $encode(T)$. It is easy to translate path expressions, and regular path expressions, while preserving semantics under this mapping (we have chosen the binary encoding such as to make the translation possible). A (regular) path expression $r$ over $\Sigma$ is translated into a (regular) path expression $translate(r)$ over $\Sigma \cup \{-\}$ (the symbol $|$ is never used): we omit its (rather obvious) formal definition, but give two examples.

$$\begin{aligned}
&translate(a.c.d) = a.(-)^*.c.(-)^*.d \\
&translate(a.(b|(c.d))^*.e) = \\
&\qquad a.(-)^*.(b.(-)^*|(c.(-)^*.d.(-)^*))^*.e
\end{aligned}$$

When the translated regular expression is evaluated on the encoded binary tree we get an encoded version of the original result; more precisely:

$$eval(translate(R), encode(t)) = \{encode(t') \mid t' \in eval(R, t)\}.$$

## 2.2 XML (eXtensible Markup Language)

In this paper we take the simplifying assumption that XML is a syntax for unranked trees. Symbols in the alphabet $\Sigma$ correspond to XML tags. For example, the unranked tree in Fig. 1 is written in XML as follows:

```
<a> <b></b> <b></b> <c><d></d></c> <e></e>
</a>
```

In XML terminology, `a`, `b`, `c`, ... are *tags*. Each tag has a start- and end-tag (e.g. `<c>` and `</c>`): the text between them is called an *element*.

Under this assumption, query languages for XML are formalisms for expressing transformations on $U_\Sigma$. Some care is needed here, however, since some of these languages (XML-QL, Lorel, StruQL) were initially defined on a graph data model (OEM [PGMW95]). Restricting these languages to trees gives us a formalism for transforming *unordered* trees, while our ranked trees are ordered. It has been shown both for XML-QL and Lorel how to cope with order, and the same can be done for UnQL and StruQL. XSL was designed for an ordered model.

We will not describe these query languages here, but only mention a feature present in all of them: pattern matching. A *pattern* is a tree in $U_{Reg(\Sigma)}$, where $Reg(\Sigma)$ is the set of regular expressions over $\Sigma$. To illustrate things, consider a pattern with three nodes, $p = [a.b]([c.(a \mid b)], [c^*.a])$, containing three regular expressions. Given an input tree $t \in U_\Sigma$, a *matching* of the pattern in the tree is given by three nodes $x_1, x_2, x_3 \in nodes(t)$, satisfying the following three conditions: $x_1 \in eval(a.b, t)$, $x_2 \in eval(c.(a \mid b), x_1)$, and $x_3 \in eval(c^*.a, x_1)$. As for the case of regular expressions, patterns over unranked trees can be translated into equivalent patterns over ranked trees. This observation is important, and justifies our choice to consider ranked trees only.

## 2.3 Types and DTDs

We next present DTDs and tree automata as type systems for unranked and ranked trees, and explain how the latter can be used to model the former.

**DTDs** In XML one can define a *Document Type Definition* (DTD) to be used for validation purposes. Formally, a DTD is an extended context-free grammar[5] with non-terminals $\Sigma$. An XML document (or, the unranked tree representing it) is valid with respect to the DTD if it encodes a derivation tree for the grammar. For a DTD $D$, we denote by $inst(D)$ the set of unranked trees representing valid XML instances of $D$. For example, the XML document represented by the tree of Fig. 1 is valid w.r.t to the DTD: $a := b^*.c.e$; $b := \varepsilon$; $c := d^*$; $d := \varepsilon$; $e := \varepsilon$

[5] In an extended CFG, the right-hand sides of productions are regular expressions over the terminals and non-terminals.

**Tree automata** A regular tree automaton is a device that accepts or rejects ranked trees. We only consider here binary trees, hence the alphabet $\Sigma$ is $\Sigma_0 \cup \Sigma_2$. The definition below is adapted from [Tho90].

**Definition 2.1** *A* non-deterministic top-down tree automaton *is $A = (\Sigma, Q, q_0, Q_F, P)$ where $Q$ is a finite set of states, $q_0 \in Q$ is the initial state, $Q_F \subseteq \Sigma_0 \times Q$ are the final symbol-state pairs, and $P \subseteq \Sigma_2 \times Q \times Q \times Q$.*

We write $(a, q) \to (q_1, q_2)$ whenever $(a, q, q_1, q_2) \in P$. Informally, the tree automaton starts in state $q_0$, and with a head on the root node of the input tree, labeled with symbol $a$. The automaton chooses two states $q_1, q_2$ such that $(a, q_0) \to (q_1, q_2)$, then continues the computation both on the left child (starting in state $q_1$) and on the right child (starting in state $q_2$). This can be thought of as spawning two computation branches, the first starting at the left child and the second starting at the right child. More computation branches are spawned for the subtrees. The input tree is accepted if all computation branches end in an accepting state, and the set of all trees accepted by $A$ is denoted $inst(A)$. A *type*, $\tau$, is a set of the form $\tau = inst(A)$ for some tree automaton $A$.

An automaton such as the above is referred to in the literature as a top-down or root-to-frontier regular tree automaton. Bottom-up regular tree automata (deterministic or non-deterministic), that start from the leaves and work their way up to the root, are known to be equivalent to the non-deterministic top-down automata.

**DTDs and regular tree automata** What is the relationship of regular tree automata to DTDs? Given a DTD $D$ it is straightforward to construct a regular tree automaton $A$ such that $inst(A) = \{encode(t) \mid t \in inst(D)\}$. The converse is false: regular tree automata are strictly stronger than DTDs. For a simple example consider the tree $t = a(b(c), b(d))$ over the unranked alphabet $\Sigma = \{a, b, c, d\}$. The set $T = \{t\} \subset U_\Sigma$ cannot be described by a DTD, because the "type" of the first $b$ differs from that of the second $b$. However the set $\{encode(t)\} \subseteq T_{\Sigma'}$ is regular, since every singleton set is regular. A very natural extension of DTDs with decoupled tags (also called specialized DTDs) turn out to be precisely equivalent to regular tree automata [BKMW98, PV99b].

**Ranked vs. unranked trees** Since both queries and types on unranked trees can be encoded into complete, binary trees, we shall consider only the latter in remainder of the paper. Hence, from now on, the alphabet $\Sigma$ is assumed to be partitioned into $\Sigma_0 \cup \Sigma_2$. All results carry over to unranked trees via the encoding.

# 3 Tree Transformations

We use $k$-pebble transducers to model tree transformations. They generalize classical top-down tree transducers, which are in turn extensions of the tree automata described above.

## 3.1 The $k$-pebble tree transducer

The $k$-pebble transducer uses up to $k$ pebbles to mark certain nodes in the tree. Transitions are determined by the current node symbol, the current state, and by the existence/absence of the various pebbles on the node. The pebbles are ordered and numbered $1, 2, \ldots, k$. The machine can place pebbles on the root, move them around, and remove them. In order to limit the power of the transducer (so that type-checking can be performed) the use of pebbles is restricted by a stack discipline: pebbles are placed on the tree in order and removed in reverse order, and only the highest-numbered pebble present on the tree can be moved. More precisely, pebble $i$, $(i > 1)$ can only be placed on the tree if pebbles $1, \ldots, i-1$ are present, and pebble $i-1$ cannot be moved as long as pebble $i$ is still on the tree. The stack discipline is essential for the results in the paper.

The transducer works as follows. The computation starts by placing pebble 1 on the root. At each point, pebbles $1, 2, \ldots, i$ are on the tree, for some $i \in \{1, \ldots, k\}$; pebble $i$ is called the *current pebble*, and the node on which it sits is the *current node*. The current pebble serves as the head of the machine. The machine decides which *transition* to make, based on the following information: the current state, the symbol under the current pebble, and the presence/absence of the other $i-1$ pebbles on the current node. There are two kinds of transitions: *move* and *output* transitions. *Move* transitions are of four kinds:

- *down-left, down-right, up-left, up-right:* the current pebble is moved from the current node in the given direction (one edge only). If a move in the specified direction is not possible, the transition does not apply.
- *stay:* the current pebble does not move, only the state changes.[6]
- *place-new-pebble:* pebble $i+1$ is placed on the root (becoming the current pebble).
- *pick-current-pebble:* the current pebble $i > 1$ is removed from the tree and pebble $i-1$ becomes the current pebble.

After each move transition the machine enters a new state, as specified by the transition.

An *output* transition emits some output node with a symbol in $a' \in \Sigma'$ and does not move the input head. There are two kinds of output transitions. In a *binary* output $a' \in \Sigma'_2$ and the machine spawns two computation branches computing the left and right child respectively. Both branches inherit the positions of all pebbles on the input, and do not communicate; each moves the $k$ pebbles independently of the other. In a *nulary* output $a' \in \Sigma'_0$, the node being output is a leaf and that branch of computation halts.

Looking at the global picture, the machine starts with a single computation branch and no output nodes. After a while it has constructed some top fragment of the output tree, and several computation branches continue to compute the remaining output subtrees. The entire computation terminates when all computation branches terminate.

We now define $k$-pebble transducers formally:

**Definition 3.1** *A $k$-pebble tree transducer is $T = (\Sigma, \Sigma', Q, q_0, P)$, where:*

- *$\Sigma, \Sigma'$ are the ranked input and output alphabets.*
- *$Q$ is a finite set of states and is partitioned into:*

$$Q = Q_1 \cup \ldots \cup Q_k$$

  *Each state in $Q_i$ "controls" pebble $i$: we denote elements in $Q_i$ with a superscript, as in $q^{(i)}$*
- *$q_0 \in Q_1$ is the initial state*
- *$P$ is a finite set of transitions of the forms:*

$$\begin{aligned}
&(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{stay})\\
&(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{down-left})\\
&(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{down-right})\\
&(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{up-left})\\
&(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i)}, \textit{up-right})\\
&(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i+1)}, \textit{place-new-pebble})\\
&(a, \bar{b}, q_1^{(i)}) \to (q_2^{(i-1)}, \textit{pick-current-pebble})\\
&(a, \bar{b}, q^{(i)}) \to (a'_0, \textit{output0})\\
&(a, \bar{b}, q^{(i)}) \to (a'_2(q_1^{(i)}, q_2^{(i)}), \textit{output2})
\end{aligned}$$

[6] Observe that this can be simulated by moves of the above form, and is used here only for convenience.

*Here $\bar{b} \in \{0,1\}^{i-1}$ and denotes the presence/absence of the previous $i-1$ pebbles on the current node. (For $i = 1$, $\bar{b} = \varepsilon$ so it provides no information, and is sometimes omitted for brevity.)*

In general $T$ can be nondeterministic and the transformation it defines is a binary relation in $T_\Sigma \times T_{\Sigma'}$. The formal definition is given in the Appendix.

**Relationship to top-down and bottom-up transducers** It is easy to see that every top-down transducer can be expressed as a 1-pebble transducers (details omitted from this abstract). The relationship to bottom-up transducers is open, and there is an interesting connection to an open problem concerning tree-walk automata, which we explain next.

A *tree-walk* automaton [EHB99] is a 1-pebble transducer without output transitions and with an accepting state. A tree-walk automaton starts with the pebble on the root, and walks up and down the tree. If it ever enters the accepting state, then it accepts the tree. (Tree-walk automata are equivalent to 1-pebble automata, Sec.4, without branching instructions.) The tree language accepted by a tree-walk automaton is regular (this also follows from Theorem 4.6 below), but it is an open problem whether tree-walk automata can express all regular tree languages: conjecture that this is not the case.

Given a bottom-up transducer $T$, we may attempt to simulated it with a 1-pebble transducer, by running $T$ in reverse. Starting from the root, at each node $x$ we guess in which state $T$ would end up at $x$, output the corresponding symbol $a'$, then branch to compute the two children of $a'$. The problem is that, during its bottom-up computation, $T$ may choose to ignore the entire output it produced for a subtree of $x$, when it reaches some node $x$: hower, it still uses the terminating state of the computation on that subtree. In the top-down simulation, we need to guess that state, then check on the subtree that the guess was correct. This test is like checking that the subtree is in a certain regular language: when doing so, we cannot branch (because whenever we branch, we must output something), i.e. we are restricted to the behavior of a tree-walk automaton. Thus, 1-pebble transducers can simulate all bottom-up transducers iff tree-walk automata can express all regular tree languages. Similarly, $k$-pebble transducers can simulated all bottom-up transducers iff $k$-pebble automata without branching (Sec.4) can express all regular languages: this is open too.

**XML transformation languages and $k$-pebble transducers** We state, without proof, that all XML transformations expressed in existing XML query languages (XML-QL, XSL, Lorel, StruQL, UnQL) can be expressed as $k$-pebble transducers. This applies to transformations on unranked trees, over a finite alphabet, expressible in these languages. In particular, it does not apply to queries with joins on data values, since these require an infinite alphabet. In lieu of a proof, we illustrate in the following section the power of $k$-pebble transducers by means of examples.

## 3.2 Examples

**Example 3.2** The following 1-pebble transducer simply copies the input tree to the output tree. $T = (\Sigma, \Sigma, \{q, q_1, q_2\}, q, P)$, where $P$ is the following:

$$
\begin{array}{lll}
(a_2, q) & \rightarrow (a_2(q_1, q_2), \textit{output2}) & \textit{for all } a_2 \in \Sigma_2 \\
(a_2, q_1) & \rightarrow (q, \textit{down-left}) & \\
(a_2, q_2) & \rightarrow (q, \textit{down-right}) & \\
(a_0, q) & \rightarrow (a_0, \textit{output0}) & \textit{for all } a_0 \in \Sigma_0
\end{array}
$$

**Example 3.3** We show here a useful "subroutine" that moves pebble 1 from a given node to the next node in pre-order. Start in state $q_1$, stop in state $q_2$; if the tree is exhausted, then enter state $q_f$. Assume that $r \in \Sigma_2$ is the root symbol:

$$
\begin{array}{lll}
(a_2, q_1) & \rightarrow (q_2, \textit{down-left}) & \text{// done} \\
(a_0, q_1) & \rightarrow (q_3, \textit{stay}) & \text{// we are on a leaf:} \\
 & & \quad \text{prepare to move up} \\
(a, q_3) & \rightarrow (q_3, \textit{up-right}) & \text{// move up as long as} \\
 & & \quad \text{coming from the right} \\
(a, q_3) & \rightarrow (q_4, \textit{up-left}) & \text{// one move up left} \\
(a, q_4) & \rightarrow (q_2, \textit{down-right}) & \text{// one move down right} \\
 & & \quad \text{and done} \\
(r, q_3) & \rightarrow (q_f, \textit{stay}) & \text{// tree exhausted}
\end{array}
$$

Here $a_0 \in \Sigma_0$, $a_2 \in \Sigma_2$ and $a \in \Sigma_0 \cup \Sigma_2 - \{r\}$.

**Example 3.4** This example illustrates why $k$-pebble transducers can express pattern matching. As argued in Sec.2.2, patterns are the most essential common denominator of existing XML query languages. Recall that a pattern is a tree labeled with regular expressions. We illustrate with $p = r_1(r_2, r_3(r_4, r_5))$, where $r_1, \ldots, r_5$ are five regular expressions. We can find all matches of this pattern in a input tree, using $k = 6$ pebbles (in general, $k$ is the number of nodes in the pattern, plus one). The transducer enumerates, in lexicographic order, all five-tuples of nodes $(x_1, x_2, x_3, x_4, x_5)$ in the input tree, using five pebbles advanced with the pre-order traversal "subroutine" from the previous example. Then, for each 5-tuple, it uses the sixth pebble to check the following five conditions: $x_1 \in eval(r_1, t)$, $x_2 \in eval(r_2, x_1)$, $x_3 \in eval(r_3, x_1)$, $x_4 \in eval(r_4, x_3)$, $x_5 \in eval(r_5, x_3)$. Each condition has the form $x_i \in eval(r_j, x_k)$ (the first condition involves the root node $t$, which we denote with $x_0$), and we can check it by locating first the node $x_i$ with the sixth pebble (using a pre-order tree traversal), then moving up the tree until we reach $x_k$, and checking the regular expression $r_j$, in reverse, along the way.

### 3.3 Complexity and Expressiveness

Besides their role in typechecking (Sec.4), $k$-pebbles transducers are of independent interest for expressing tree transformations. We show here that their data complexity is PTIME. In the case of a deterministic transducer $T$ this means that there exists a PTIME algorithm that "computes" $T(t)$ for an input tree $t$. Note that the generated output $T(t)$ can have a size exponential in $t$ (proof omitted). Still, the algorithm will produce a polynomial-size encoding of $T(t)$, as a DAG. In the case of non-deterministic $k$-pebble transducers we need to be more careful what the PTIME data complexity means. In particular $T$ can produce an infinite set of outputs for a given $t$: $T(t) = \{t_1, t_2, \ldots\}$. We show:

**Proposition 3.5** *Let $T$ be a fixed $k$-pebble transducer. Then for each input tree $t$, (1) the set $T(t)$ is a regular tree language, and (2) one can construct in* PTIME *(in the size of $t$) a regular tree automaton $A_t$ that accepts the language $T(t)$.*

The proof is outlined in the Appendix.

Proposition 3.5 can be used to effectively answer several questions about $T(t)$. Consider the *construction problem*: assuming a deterministic transducer $T$, given a tree $t$ construct $T(t)$. This can be answered in PTIME in the size of $t$: indeed, $A_t$ serves as a DAG encoding of the result $T(t)$. If required, $T(t)$ can be actually computed from $A_t$ in time polynomial in the size of both $t$ and $T(t)$.

Next consider the *decision problem*: given $t, t'$, is $t' \in T(t)$ ? This obviously can be answered in PTIME in the size of $t$ and $t'$, by "running" the automaton $A_t$ on $t'$. Finally, the *enumeration problem* enumerates $T(t) = \{t_1, t_2, \ldots\}$: this corresponds to enumerating all trees generated by a regular tree grammar, and can be done in amortized PTIME cost in the size of the input and the output.

It easily follows from Proposition 4.5 and Theorem 4.6 below that $k$-pebble transducers cannot express all PTIME transformations. For example, transformations that involve cardinality checking (e.g. "output 'a' if the input tree is a right linear tree labeled with $b^n.c^n$) are not expressible by $k$-pebble transducers. The exact characterization of their expressive power remains open.

## 4 Typechecking $k$-pebble Transducers

Now that we have presented our types and transformation language, we are ready to present the solution for the typechecking problem. We first recall the typechecking and type inference problems, and the connection between them.

Recall that a type is a set of the form $\tau = inst(A)$ for some tree automaton $A$. We denote $T(\tau) \stackrel{\text{def}}{=} \bigcup_{t \in \tau} T(t)$.

**Definition 4.1** *(Typechecking) A $k$-pebble tree transducer $T$* typechecks *with respect to an input type $\tau_1$ and an output type $\tau_2$, if $T(\tau_1) \subseteq \tau_2$.*

The problem of verifying if $T$ typechecks with respect to types $\tau_1, \tau_2$ is called the *typechecking problem*.

The *type inference problem* is closely related to typechecking. Given a $k$-pebble transducer $T$ and an input type $\tau_1$, the type inference problem is to construct, if possible, a type $\tau_2'$ such that $T(\tau_1) = \tau_2'$. Note that a solution to the type inference problem immediately yields a solution to the typechecking problem. To verify that $T$ typechecks with respect to input $\tau_1$ and output $\tau_2$, first infer $\tau_2' = T(\tau_1)$, then check that $\tau_2' \subseteq \tau_2$, using the fact that inclusion of regular tree languages is decidable. Unfortunately, as illustrated by Example 4.2, type inference is not possible even for very simple types and transformations.

**Example 4.2** Let $Q1$ be the following XML-QL query:

```
<result> WHERE <root> <a> $X </a> <a> $Y </a>
               </root>
         CONSTRUCT <b/>
</result>
```

Consider the input DTD $\tau_1$ given by $root := a^*$. Thus, the input XML documents are of the form:

```
<root><a/> . . . <a/> </root>
```

with $n$ occurrences of `<a/>`, for some $n \geq 0$ (in XML `<a/>` is an abbreviation for `<a> </a>`). On such a document, query $Q1$ binds each of the two variables `$X` and `$Y` to some `<a/>` element, and for each binding emits a `<b/>` output element. Hence, the query's outputs have the form:

```
<result> <b/> <b/> . . . <b/> </result>
```

with $n^2$ occurrences of <b>. In short, the query maps an input $a^n$ to the output $b^{n^2}$. Clearly the set $\tau$ of all results is not described by a DTD, since DTDs can describe only regular languages. In practice one often settles for a DTD approximating $\tau$, like *result* $:= b^*$. This is definitely not the best approximation: in fact no best approximation for non-regular tree languages $\tau$ exists, because for any regular set $\tau' \supset \tau$ and for any tree $t \in \tau' - \tau$, the set $\tau' - \{t\}$ is still regular and a better approximation than $\tau'$.

Given the failure of type inference, one might consider several alternatives, such as:

- Use an upper approximation of the output $T(\tau_1)$, like $b^*$ above. This may work in many cases, but will result in some correct transformations being rejected by the typechecker.
- Use a stronger type system to express more sets of the form $T(\tau_1)$: we need to exercise care, because for too powerful type systems the test $T(\tau_1) \subseteq \tau_2$ may become undecidable. This approach is pursued in [PV99b] which extends DTDs from regular to context free languages, thus expressing more sets of the form $T(\tau_1)$ (but still not all), while keeping the inclusion test decidable (when $\tau_2$ is regular).

The failure of type inference has a surprising twist, which is crucial to our solution to the typechecking problem: *inverse* type inference turns out to be possible. Returning to Example 4.2, consider query $Q1$ and assume the output DTD to be $(b.b)^*$: i.e., we require to have an even number of $b$'s. It is easy to see that $(a.a)^*$ is the DTD describing precisely the input documents mapped by $Q1$ into output documents with an even number of $b$'s.

More precisely, the *inverse type inference problem* is the following: given a $k$-pebble transducer $T$ and an output type $\tau$, construct a type $\tau^{-1}$ such that $\tau^{-1} = \{t \mid T(t) \subseteq \tau\}$: the type $\tau^{-1}$ is called the *inverse inferred type* for $T$ and $\tau$. The solution to the inverse type inference problem immediately yields a solution to the typechecking problem: to typecheck $T$ with respect to input $\tau_1$ and output $\tau_2$, first compute the inverse type $\tau_2^{-1}$ for $T$ and $\tau_2$, then check that $\tau_1 \subseteq \tau_2^{-1}$. This establishes the main result of the paper:

**Theorem 4.3** *It is decidable, given a $k$-pebble transducer $T$, an input type $\tau_1$, and an output type $\tau_2$, whether $T$ typechecks with respect to $\tau_1$ and $\tau_2$.*

The remainder of the section is dedicated to solving the inverse type inference problem, thus establishing Theorem 4.3. We do this in three stages:

**Step 1** define an acceptor variant of the transducer, called *$k$-pebble automaton*;

**Step 2** show that for each $k$-pebble transducer $T$ and type $\tau$, the complement of $\{t \mid T(t) \subseteq \tau\}$ is recognized by some $k$-pebble automaton; and,

**Step 3** prove that every $k$-pebble automaton recognizes a regular tree language.

We elaborate on the three steps next.

*Step 1.* The $k$-pebble automaton works similarly to the $k$-pebble transducer, but produces no output: its function is simply to accept or reject its input. More formally:

**Definition 4.4** *A $k$-pebble tree automaton is a 4-tuple $(\Sigma, Q, q_0, P)$, where:*

- *$\Sigma, Q, q_0$ are as in a $k$-pebble transducer.*
- *$P$ is a set of transitions. Move transitions are as in a $k$-pebble transducer. Output transitions are replaced by the following:*

$$(a_0, \bar{b}, q^{(i)}) \rightarrow (\mathit{branch0})$$
$$(a_2, \bar{b}, q^{(i)}) \rightarrow ((q_1^{(i)}, q_2^{(i)}), \mathit{branch2})$$

The *branch0* and *branch2* transitions are analogous to the transitions of regular tree automata. The *branch0* transition stops the current computation branch and accepts. The *branch2* transition spawns two independent computations, in states $q_1^{(i)}$ and $q_2^{(i)}$ respectively; the input head is not moved. The set of trees *inst*$(A)$ accepted by $A$ is formally defined in the Appendix.

*Step 2.* Inverse type inference can be achieved using $k$-pebble automata.

**Proposition 4.5** *For each $k$-pebble transducer $T$ and type $\tau$ there exists a $k$-pebble automaton $A$ such that $\mathit{inst}(A) = \overline{\{t \mid T(t) \subseteq \tau\}}$*

A bar denotes the complement set. Note that $\overline{\{t \mid T(t) \subseteq \tau\}} = T^{-1}(\bar{\tau})$: hence, intuitively $A$ is obtained by composing $T$ with the top-down automaton $B$ describing the type $\bar{\tau}$. This is possible since $B$ consumes the tree top-down, exactly in the order in which $T$ produces it. A formal proof is provided in the Appendix.

*Step 3.* Finally, we can show the following result, which is the cornerstone of our approach to inverse type inference, and technically the most interesting.

**Theorem 4.6** *$k$-pebble tree automata accept precisely the regular tree languages.*

The proof is provided in the Appendix. The difficult part is showing that $k$-pebble automata do not accept *more* than the regular tree languages. This is done as follows: for each $k$-pebble automaton $A$, construct a Monadic Second Order (MSO) sentence $\varphi$ that defines acceptance of a binary tree by $A$. The result then follows from the known fact that MSO defines precisely the regular tree languages [RS97].

# 5 Extensions

The main limitations of our typechecking approach are the high complexity and the lack of data values. Both can be dealt with in restricted cases: of special interest are transformations expressed in the usual XML query languages. A full treatment is beyond the scope of this paper; here we mention some initial results indicating that both limitations can be overcome in significant practical cases.

**Complexity of typechecking** The algorithm resulting from our typechecking technique for a $k$-pebble transducer is hyperexponential in $k$ (so non-elementary). The number of pebbles in the pebble automaton is the main source of complexity in the typechecking procedure. This is not unexpected. Indeed, as noted by [GH94] for the case of strings, automata with $k$ pebbles (more restricted than ours) are *extremely concise*: their equivalent automata without pebbles are larger by a tower of exponentials of height $k$. The good news is that automata with pebbles are convenient and very economical. The bad news is that the complexity of usual decision problems involving $k$-pebble automata is likely to be harder in proportion to their conciseness. For example, the straightforward test of emptiness of pebble automata is non-elementary (tower of exponential with height depending on the number of pebbles). The question of lower bounds remains however open.

Thus, typechecking in its full generality appears to be prohibitively expensive. However, our approach can be adapted to yield better complexity in restricted cases of practical interest. Fortunately, even one or two pebbles can be quite powerful. For example, XML-QL queries with a single pattern and returning a single variable can be simulated by 1-pebble transducers (with exponentially many states), and as a consequence can be typechecked in 2-EXPTIME. This can be extended to the case when the query constructs a complex result, but groups only on one variable at a time: typechecking can still be done in 2-EXPTIME.

**Data Values** To model `#PCDATA` in XML we assume an infinite alphabet $D$ of data values in addition to $\Sigma$. Trees are now in $T_\Sigma(D)$, while types are regular tree languages over $T_\Sigma(\{d\})$, i.e. where all data values are treated as a distinguished leaf symbol $d$. The $k$-pebble transducers are extended with three new kinds of transitions (see also Sec. 1): a comparison predicate $x = y$ that takes two data values $x, y$ and enters either some state $q_1$ (when $x = y$) or $q_2$ (when $x \neq y$), unary comparisons on data values, and an output transition which copies a data value from the input tree to the output tree. All XML-QL queries, including those with selections and joins on data values, can be expressed with the extended transducers. The last two kinds of transitions do not affect typechecking. For example, to handle unary predicates we apply the technique in [AV97]: if the machine uses $m$ predicates, then replace the infinite set of data values with $2^m$ new constants. The comparison predicate $x = y$ however renders typechecking undecidable in general, because of a reduction from the finite satisfiability problem for first-order logic: the equality predicate allows to evaluate any first-order formula on standard encodings of first-order structures as labeled trees with data values.

However, the typechecking techniques in this paper can be extended to an important class of queries with data value joins. Intuitively, these are queries where all equality tests performed are independent of each other. Consequently, all truth assignments to the equality tests are consistent. As far as typechecking is concerned, the actual equality tests can therefore be replaced by nondeterministic guesses of their truth value, without the risk of inconsistent guesses. In practice, many queries expressed in languages such as XML-QL can be implemented in this manner. To illustrate, assume a relational schema consisting of three relations `Person(pid, name)`, `Worksin(pid, did)`, `Dept(did, name)`, with `pid, did` keys in `Person, Dept` respectively, and consider a three way join $Q$ = `Person` $\bowtie$ `Worksin` $\bowtie$ `Dept`. Such joins are typical in XML-QL queries exporting a relational database to an XML view. The following extended $k$-pebble transducer $T$ implements $Q$ and performs only independent comparisons. In an outermost loop $T$ iterates `w` over `Worksin`. At each step it iterates `p` over `Person` checking `w.pid = p.pid`: *$T$ stops the inner loop when an equality is found*. Next it iterates over `Dept` and stops when an equality is found. Each comparison done by $T$ is independent on all previous comparisons, i.e. both outcomes $x = y$ or $x \neq y$ are consistent with all previous comparisons. Notice that other implementations of the three way join may not have the independence property. Thus, $T$ can be simulated by a non-deterministic transducer $T'$ with inputs in $T_\Sigma(\{d\})$ by replacing every predicate $x = y$ with output states $q_1, q_2$ with a non-deterministic choice between $q_1$ and $q_2$. Every nondeterministic choice of $T'$ corresponds to a possible run of $T$ (on certain data values) and we can apply the typechecking techniques from this paper to $T'$.

## 6 Conclusion

We have developed a general, robust framework for typechecking XML transformers. The $k$-pebble transducer provides an abstraction that subsumes the tree transformation core of the main XML transformation languages such as XML-QL and XSL. The regular tree languages capture current DTDs and several proposed extensions. Thus, our framework for typechecking is likely to remain relevant as the XML standards evolve.

Future work has to address the complexity of typechecking, and the treatment of data values, in the context of restricted classes of XML transformations. Restrictions are often acceptable in practice, and may lead to efficient typechecking algorithms. We believe that $k$-pebble transducers provide a good framework for such a study, as suggested by our preliminary results in this direction.

## References


[AU71] A.V. Aho and J.D. Ullman. Translations on a context free grammar. *Information and Control*, 19(19):439–475, 1971.

[AV97] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 122–133, 1997.

[BDHS96] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 505–516, 1996.

[BKMW98] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree languages over non-ranked alphabets, 1998. Available at `ftp://ftp11.informatik.tu-muenchen.de/pub/misc/caterpillars/`.

[BLM+99] D. Beech, S. Lawrence, M. Maloney, N. Mendelsohn, and H. Thompson. Xml schema part 1: Structures, May 1999. `http://www.w3.org/TR/xmlschema-1/`.

[BM99a] Catriel Beeri and Tova Milo. Schemas for integration and translation of structured and semi-structured data. In *Proceedings of the International Conference on Database Theory*, 1999. to appear.

[BM99b] P. Biron and A. Malhotra. Xml schema part 2: Datatypes, May 1999. `http://www.w3.org/TR/xmlschema-2/`.

[BT99] C. Beeri and Y. Tzaban. SAL: an algebra for semistructured data and XML. In *Proceedings of WebDB*, Philadelphia, PA, June 1999.

[CDSS98] S. Cluet, C. Delobel, J. Simeon, and K. Smaga. Your mediators need data conversion ! In *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 177–188, 1998.

[Cla99a] James Clark. Xml path language (xpath), 1999. `http://www.w3.org/TR/xpath`.

[Cla99b] James Clark. Xsl transformations (xslt) specification, 1999. `http://www.w3.org/TR/WD-xslt`.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for xml. In *Proceedings of the Eights International World Wide Web Conference (WWW8)*, Toronto, 1999.

[EHB99] J. Engelfriet, H.J. Hoogenboom, and J.P. Van Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.

[FFK+98] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. Catching the boat with Strudel: experience with a web-site management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1998.

[GH94] Noa Globerman and David Harel. Complexity results for multi-pebble automata and their logics. In *Proceedings of ICALP*, pages 73–82, Jerusalem, Israel, 1994.

[GHR95] Raymond Greenlaw, H.James Hoover, and Walter L. Ruzzo. *Limits to Parallel Computation. P-Completeness Theory.* Oxford University Press, New York, Oxford, 1995.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[LPVV99] B. Ludaescher, Y. Papakonstantinou, P. Velikhov, and V. Vianu. View definition and dtd inference for xml. In *Workshop on Semistructured Data and Nonstandard Data Formats*, January 1999.

[MN99] S. Maneth and F. Neven. Structured document transformations based on xsl. In *In Proc. Eighth Int'l. Workshop on Database Programming Languages*, Scotland, 1999.

[MS99] Tova Milo and Dan Suciu. Type inference for queries on semistructured data. In *Proceedings of the ACM Symposium on Principles of Database Systems*, 1999.

[MW99] J. McHugh and J. Widom. Query optimization for xml. In *Proceedings of VLDB*, Edinburgh, UK, September 1999.

[NS99] Frank Neven and Thomas Schwentick. Query automata. In *Symposium on Principles of Database Systems*, pages 205–214, Philadelphia, PA, 1999.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *IEEE International Conference on Data Engineering*, pages 251–260, March 1995.

[PV99a] Y. Papakonstantinou and P. Velikhov. Enhancing semistructured data mediators with document type definitions. In *Int'l Conf. on Data Engineering*, 1999.

[PV99b] Y. Papakonstantinou and V. Vianu, 1999. DTD inference for views of XML data.

[Rob99] Jonathan Robie. The design of xql, 1999. `http://www.texcel.no/whitepapers/xql-design.html`.

[RS97] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages.* Springer Verlag, 1997.

[Tho90] W. Thomas. Automata on infinite objects. In *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 4, pages 133–192. Elsevier, Amsterdam, 1990.

# Appendix

## A Definitions of $k$-pebble transducers and automata

We formaly define below the transformations/tree language defined by $k$-pebble transducers and automata, respectively.

**$k$-pebble transducers** We define the transformation computed by a $k$-pebble transducer $T$, as a binary relation in $T_\Sigma \times T_{\Sigma'}$.

Given a tree $t \in T_\Sigma$, a *configuration* of $T$ on $t$ is $\gamma = [i, q^{(i)}, \theta]$, where $1 \leq i \leq k$, $q^{(i)} \in Q_i$, and $\theta : \{1, 2, \ldots, i\} \to \mathit{nodes}(t)$. Let $\Gamma_{T,t}$ be the set of all configurations for transducer $T$ and input tree $t$: we will drop the indexes when no confusion arises, writing simply $\Gamma$. The *initial* configuration is: $\gamma_{0,T,t} \stackrel{\text{def}}{=} [1, q_0, \theta]$, where $\theta(1) = \mathit{root}(t)$ (recall that $q_0$ is the initial state in $T$ and belongs to $Q_1$) and we abbreviate it with $\gamma_0$.

A partial computation of the transducer $T$ on input $t$ is captured by a tree in $T_{\Sigma'}(\Gamma)$. Recall that this is the set of trees over $\Sigma'$ with leaves labeled with elements in $\Gamma$. The tree represents the output generated so far; the leaves labeled with $\gamma \in \Gamma$ represent the on-going branches of the computation. The transducer $T$ defines a rewriting $\Rightarrow$ on $T_{\Sigma'}(\Gamma)$, called the $T$-rewriting, describing one step of the computation: intuitively $t'_1 \Rightarrow t'_2$ if $t'_2$ can be obtained from $t'_1$ by applying one transition step to one of the on-going branches of computations in $t'_1$. The $T$-rewriting is defined in terms of a simpler rewriting $\to$, from $\Gamma$ to $T_{\Sigma'}(\Gamma)$, with the meaning that $\gamma \to t'$ if $t'$ can be obtained from $\gamma$ in one transition step: we define $\to$ below. Then $\Rightarrow$ is the congruence closure of $\to$, as

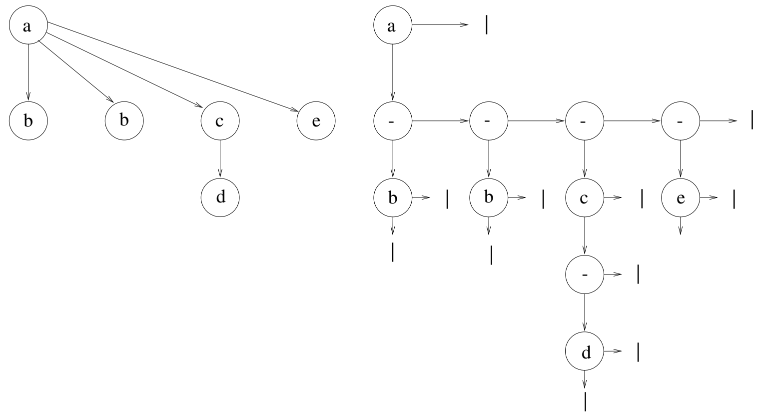


Figure 1: Encoding of an unranked tree into a binary tree.

follows: (1) if $\gamma \to t'$ then $\gamma \Rightarrow t'$, and (2) whenever $t'_1 \Rightarrow t'_2$, then we also have:

$$\begin{aligned} a'_2(t'_1, t') &\Rightarrow a'_2(t'_2, t') \\ a'_2(t', t'_1) &\Rightarrow a'_2(t', t'_2) \end{aligned}$$

for all $a'_2 \in \Sigma'_2$ and $t' \in T_{\Sigma'}(\Gamma)$. Let $\stackrel{*}{\Rightarrow}$ be the transitive closure of $\Rightarrow$.

The *transformation* defined by the $k$-pebble tree transducer $T$ is the binary relation consisting of all pairs $(t, t') \in T_\Sigma \times T_{\Sigma'}$ s.t. $\gamma_0 \stackrel{*}{\Rightarrow} t'$. We denote with $T(t)$ the set of trees $t'$ such that $(t, t')$ is in the transformation defined by $T$.

It remains to define the rewriting $\gamma \to t'$, which applies one transition step to $\gamma$. Recall that $P$ is the set of transitions in the transducer $T$. Given $\gamma = [i, q^{(i)}, \theta]$ and a transition $(a, \bar{b}, q_1^{(i')}) \to \ldots$ in $P$, we say that the transition *applies* to $\gamma$ if the following four conditions hold: (1) $i = i'$, (2) the symbol of the node $\theta(i)$ is $a$, (3) $\beta_1\beta_2 \ldots \beta_{i-1} = \bar{b}$ where $\beta_j = 1$ if $\theta(j) = \theta(i)$, and $\beta_j = 0$ if $\theta(j) \neq \theta(i)$, for $j = 1, i-1$, and (4) $q^{(i)} = q_1^{(i)}$. Now we can finally define $\to$. This is done separately for move transitions and for output transitions. For each move transition

$$(a, \bar{b}, q_1^{(i)}) \quad \to \quad (q_2^{(i)}, d)$$

in $P$, which applies to $\gamma$, we define $\gamma \to \gamma'$, where $\gamma' = (q_2^{(i)}, \theta')$, $\theta'(j) = \theta(j)$ for $j = 1, i-1$, and $\theta'(i)$ is obtained from $\theta(i)$ as follows: if $d =$ *stay*, then $\theta'(i) = \theta(i)$; if $d =$ *down-left*, then $\theta'(i)$ is the left child of $\theta(i)$; if $d =$ *up-left*, then $\theta'(i)$ is the parent of $\theta(i)$ provided that $\theta(i)$ is the left child of $\theta'(i)$ (otherwise the transition $\gamma \to \gamma'$ is not defined); etc.

For each output transition:

$$(a, \bar{b}, q^{(i)}) \quad \to \quad (a'_0, \textit{output0})$$

in $P$, which applies to $\gamma$, we define $\gamma \to a'_0$. For each output transition:

$$(a, \bar{b}, q^{(i)}) \quad \to \quad (a'_2(q_1^{(i)}, q_2^{(i)}), \textit{output2})$$

which applies to $\gamma$, we define $\gamma \to a'_2(\gamma_1, \gamma_2)$, where $\gamma_1 = [i, q_1^{(i)}, \theta]$, $\gamma_2 = [i, q_2^{(i)}, \theta]$.

**$k$-pebble automata** We next describe formally the set of trees accepted by $A$ next.

As in the case of $k$-pebble transducers, given a $k$-pebble automaton $A$ and a tree $t$, we consider configurations $\gamma$ of $A$ on $t$, and denote $\Gamma_{A,t}, \gamma_{0,A,t}$ the set of all configurations, and the initial configuration respectively. We abbreviate them with $\Gamma, \gamma_0$.

The partial computation of an automaton $A$ on a tree $t$ is described by a word in $\Gamma^*$, denoting the configurations reached so far by the different branches. We define now the $A$-rewriting $\to$ on $\Gamma^*$, similar to the $T$-rewriting for transducers. We start with single configurations. If $\gamma'$ is obtained from $\gamma$ in a single move, then $\gamma \to \gamma'$ (the definition here is similar to that for transducers and is omitted). Whenever a *branch0* transition:

$$(a, \bar{b}, q^{(i)}) \to (\textit{branch0})$$

applies to some configuration $\gamma$, then $\gamma \to \varepsilon$. Finally, whenever a *branch2* transition:

$$(a, \bar{b}, q^{(i)}) \to ((q_1^{(i)}, q_2^{(i)}), \textit{branch2})$$

applies to $\gamma = [i, q^{(i)}, \theta]$, then $\gamma \to \gamma_1\gamma_2$, where $\gamma_1 = [i, q_1^{(i)}, \theta], \gamma_2 = [i, q_2^{(i)}, \theta]$.

The $A$-rewriting $\Rightarrow$ is defined to be the congruential closure of $\to$. (1) if $\gamma \to w$, then $\gamma \Rightarrow w$, and (2) if $w_1 \Rightarrow w_2$ then:

$$\begin{aligned} w.w_1 &\Rightarrow w.w_2 \\ w_2.w &\Rightarrow w_2.w \end{aligned}$$

A tree $t$ is *accepted* by $A$, iff $\gamma_0 \stackrel{*}{\Rightarrow} \varepsilon$. The tree language *inst*$(A)$ accepted (or *recognized*) by $A$ consists of all trees accepted by $A$.

# B Some Proofs

We provide below proof sketches of the main results of the paper. First we need to extend top-down automata with silent transitions.

**Top-down tree automata with silent transitions** According to Definition 2.1 a top-down tree automaton $A = (\Sigma, Q, q_0, Q_F, P)$ has transitions of the form:

$$(a, q) \quad \rightarrow \quad (q_1, q_2) \tag{1}$$

A top-down tree automaton with *silent* transitions has, in addition, transitions of the form:

$$(a, q) \quad \rightarrow \quad q' \tag{2}$$

Such a move causes a state change from $q$ to $q'$ while keeping the head on the current node. Their semantics is defined as follows (we follow the constructions in Sec. 3.1). For a given tree $t$, a *configuration* of $A$ on $t$ is $\delta = [q, x]$, where $q \in Q$, $x \in nodes(t)$; $\Delta$ denotes the set of all configurations. The relation $\rightarrow$ on $\Delta \times \Delta^*$ is defined as follows. Let $\delta = [q, x]$. For every non-silent transition (of type 1) applicable[7] to $\delta$ we define $\delta \rightarrow \delta_1\delta_2$ where $\delta_1 = [q_1, x_1], \delta_2 = [q_2, x_2]$, and $x_1, x_2$ are the two children of $x$. For every silent transition (of type 2) applicable to $\delta$ we define $\delta \rightarrow \delta'$, where $\delta' = [q', x]$. Finally, whenever $(a, q) \in Q_F$, where $a$ is $x$'s label, then we also define $\delta \rightarrow \varepsilon$. Next, define $\Rightarrow$ on $\Delta^* \times \Delta^*$ to be the congruential closure of $\rightarrow$: $\delta \rightarrow w'$ implies $\delta \Rightarrow w'$, and $w_1 \Rightarrow w_2$ implies $w_1 w \Rightarrow w_2 w$ and $w w_1 \Rightarrow w w_2$. Let $\stackrel{*}{\Rightarrow}$ be its transitive and reflexive closure. A tree $t$ is accepted by $A$ if $\delta_0 \stackrel{*}{\Rightarrow} \varepsilon$, where $\delta_0 = [q_0, root(t)]$ is the initial configuration. We state without proof that, for automata without silent transitions, this definition of acceptance coincides with that in Definition 2.1.

For every top-down automaton $A = (\Sigma, Q, q_0, Q_F, P)$ with silent transitions there exists an equivalent top-down automaton $A_0$ without silent transitions. To see that, denote $q \rightarrow_a q'$ whenever a transition $(a, q) \rightarrow q'$ is in $P$, and let $q \stackrel{*}{\rightarrow}_a q'$ be the transitive and reflexive closure. Define $P'$ to consists of all transitions of the form $(a, q) \rightarrow (q_1, q_2)$ whenever $q \stackrel{*}{\rightarrow}_a q'$ and $(a, q') \rightarrow (q_1, q_2)$ is in $P$, and $Q'_F = \{(a, q) \mid q \stackrel{*}{\rightarrow}_a q', (a, q') \in Q_F\}$. Then $A_0 \stackrel{\text{def}}{=} (\Sigma, Q, q_0, Q_F, P_0)$ is equivalent to $A$ (proof omitted) and has no silent transitions.

**Proof of Proposition 3.5:** (Sketch) For the sequel we fix some $k$-pebble tree transducer $T = (\Sigma, \Sigma', Q, q_0, P)$ and some input tree $t$. Recall that $T(t)$ is the set of output trees. We will construct a top-down tree automaton $A_t$, with silent transitions, which accepts precisely the language $T(t)$. Namely $A_t \stackrel{\text{def}}{=} (\Sigma', \Gamma \cup \{q_f\}, \gamma_0, \Sigma' \times \{q_f\}, P')$, where $\Gamma$ is the set of configurations of $T$ on $t$ (which we call $T$-configurations here), $q_f$ is a new symbol denoting the unique final state, and $\gamma_{0,T,t}$ is the initial $T$-configuration (Sec. 3.1). We define the productions $P'$ next. Recall from Sec. 3.1 the rewriting relation $\gamma \rightarrow t'$, where $\gamma \in \Gamma$ and where $t' \in T_{\Sigma'}(\Gamma)$ had one of the following three forms: (1) $t'$ is a $T$-configuration $\gamma'$, (2) $t'$ is an output symbol $a \in \Sigma'_0$, (3) $t'$ is a tree with three nodes $a'(\gamma_1, \gamma_2)$, $a' \in \Sigma', \gamma_1, \gamma_2 \in \Gamma$. Define $P'$ to consists of the following:

$$\begin{array}{ll} (a', \gamma) \rightarrow \gamma' & \text{(for every rewriting } \gamma \rightarrow \gamma'\text{)} \\ (a', \gamma) \rightarrow q_f & \text{(for every rewriting } \gamma \rightarrow a'\text{)} \\ (a', \gamma) \rightarrow (\gamma_1, \gamma_2) & \text{(for every rewriting } \gamma \rightarrow a'(\gamma_1, \gamma_2)\text{)} \end{array}$$

If $t$ has $n$ nodes, then there are $O(n^k)$ configurations, hence $A_t$ can be computed in PTIME in the size of $t$.

It is easy to see that $A_t$ accepts some tree $t'$ iff $t' \in T(t)$.

$\square$

**Proof of Proposition 4.5:** (Sketch) Let $T = (\Sigma, \Sigma', Q^T, q_0^T, P^T)$ be a $k$-pebble tree transducer. Consider the complement of $\tau^{-1} = \{t \mid T(t) \subseteq \tau\}$, i.e. the set of trees $t$ such that $T(t) \cap \bar{\tau} \neq \emptyset$. We construct a $k$-pebble automaton $A$ accepting this set.

Since $\tau$ is a regular tree language, its complement $\bar{\tau}$ is also a regular tree language, accepted by some top-down automaton $B = (\Sigma, Q^B, q_0^B, Q_F^B, P^B)$, hence $\bar{\tau} = inst(B)$. Define $A = (\Sigma, Q^T \times Q^B, (q_0^T, q_0^B), P)$, where the transitions $P$ are defined as follows. Recall that $k$-pebble transducers have *move* and *output* transitions, and $k$-pebble automata have *move* and *branch* transitions.

- For every *move* transition in $P^T$:

  $$(a, \bar{b}, q_1^T) \rightarrow (q_2^T, d)$$

  and for every state $q^B \in Q^B$, $A$ has a move transition

  $$(a, \bar{b}, (q_1^T, q^B)) \quad \rightarrow \quad ((q_2^T, q^B), d) \tag{3}$$

[7] The definition of when a transition is applicable to a configuration is the same as in Sec. 3.1.

- For every output transition in $P^T$:

$$(a, \bar{b}, q^T) \rightarrow (a'_0, \mathit{output0})$$

and for every state $q^B \in Q^B$ for which $(a'_0, q^B) \in Q^B_F$ we have in $A$ a *branch0* transition:

$$(a_0, \bar{b}, (q^T, q^B)) \quad \rightarrow \quad (\mathit{branch0}) \qquad (4)$$

- For every output transition in $P^T$:

$$(a, \bar{b}, q^T) \rightarrow (a'(q^T_1, q^T_2), \mathit{output2})$$

and for every branch transition in $P^B$:

$$(a', q^B) \rightarrow (q^B_1, q^B_2)$$

we have in $A$ a *branch2* transition:

$$(a, \bar{b}, (q^T, q^B)) \quad \rightarrow \quad ((q^T_1, q^B_1), (q^T_2, q^B_2), \mathit{branch2}) \qquad (5)$$

To conclude the proof we need to show that for every tree $t$, $T(t) \cap \mathit{inst}(B) \neq \emptyset$ iff $t \in \mathit{inst}(A)$. This is intuitively obvious, since $A$ simulates $T$ in parallel with the computation of $B$ on $T$'s output. The formal proof is omitted here.

**Proof of Theorem 4.6:** Clearly, $k$-pebble automata can simulate classical top-down tree automata using a single pebble, so each regular tree language is accepted by some $k$-pebble automaton. The converse is considerably harder. To prove it, we show how to construct for each $k$-pebble tree automaton $A$ an MSO (monadic second order logic) formula $\varphi_A$ s.t. for every tree $t$, $A$ accepts $t$ iff $t \models \varphi_A$. Since MSO formulas define precisely the regular tree languages, the theorem follows.

The result that MSO on trees defines precisely the regular tree languages assumes a certain representation of trees as first-order structures. A tree $t$ is represented by a structure $(D, succ1, succ2, (R_a)_{a\in\Sigma})$, where $D$ is the set of nodes, $succ1(x, y)$ and $succ2(x, y)$ denote the fact that $y$ is the left (right) child of $x$, and $R_a(x)$ holds if node $x$ has label $a$. As a warm-up, consider a simple example: the descendant relation among nodes of $t$. This is defined by the MSO formula:

$$\varphi(x, y) = \forall S.\ (\ S(x) \wedge \mathit{closed}(S) \Longrightarrow\ S(y))$$

where

$$\begin{aligned}\mathit{closed}(S) = \quad & (\forall u\ \forall v\ (S(u) \wedge succ_1(u, v) \Longrightarrow S(v))) \\ & \wedge\ (\forall u\ \forall v\ (S(u) \wedge succ_2(u, v) \Longrightarrow S(v)))\end{aligned}$$

Clearly, $\varphi(x, y)$ holds iff $y$ is a descendant of $x$ in $t$. Indeed, $x$'s descendants can be described as the smallest set $S$ containing $x$ and which is closed under $succ_1$ and $succ_2$. Note the universal quantification of $S$.

Also by way of warm-up, consider an *and/or* tree whose internal nodes are labeled $\wedge, \vee$ and whose leaves are labeled $0, 1$. It is easy to see that there is a 1-pebble automaton accepting precisely those trees that evaluate to 1 when viewed as Boolean circuits. An MSO formula $\varphi$ defining the same set of trees is obtained by "traversing" the tree bottom-up, from leaves to root, and defining the set of nodes whose subtrees evaluate to 1 using a unary relation $S$:

$$\begin{aligned}\varphi = \forall S.\ ((\ \forall x.R_1(x) \Longrightarrow S(x)) \wedge \mathit{reverse\text{-}closed}(S)) \\ \Longrightarrow S(\mathit{root})\end{aligned}$$

where

$$\mathit{reverse\text{-}closed}(S) = \forall x \forall y\ .(R_\vee(x)\ \wedge\ (succ_1(x, y) \vee succ_2(x, y))\ \wedge S(y)) \Longrightarrow S(x) \wedge\ \forall x \forall y \forall z\ .(R_\wedge(x) \wedge succ_1(x, y) \wedge succ_2(x, z)\ \wedge\ S(y)\ \wedge\ S(z)) \Longrightarrow S(x)$$

We reduce the simulation problem of a $k$-pebble automaton to the *Alternating Graph Accessibility Problem*, AGAP [GHR95]. An alternating graph (or *and/or*) graph is a graph $G = (V, E)$ whose nodes $V$ are partitioned into *and*-nodes and *or*-nodes: $V = V_\wedge \cup V_\vee$. The problem consists in deciding whether a node $x \in V$ is *accessible*, where accessibility is defined as follows: an *and*-node is accessible if all its successors are accessible; an *or*-node is accessible if at least one of its successors is accessible. The set of accessible nodes $x$ is defined by:

$$\varphi(x) = \forall S.(\mathit{reverse\text{-}closed}(S) \Rightarrow S(x))$$

where

$$\mathit{reverse\text{-}closed}(S) \ = \ \forall y.(D_\vee(y)\ \wedge\ \exists z.(E(y, z)\ \wedge\ S(z)) \ \Longrightarrow\ S(y))\ \wedge\ \forall y.(D_\wedge(y)\ \wedge\ \forall z.(E(y, z)\ \Rightarrow S(z)) \Longrightarrow S(y))$$

Given a $k$-pebble automaton $A$ and a tree $t$, construct the following *and/or* graph $G_{A,t} = (V, E)$. Its *or*-nodes are $A$-configurations on $t$, $V_\vee = \Gamma_{A,t}$ (see Sec. 4); its *and*-nodes are pairs of configurations $(\gamma_1, \gamma_2)$ which have the same $\theta$ component, plus a node called $\varepsilon$, $V_\vee = \{\varepsilon\} \cup \{(\gamma_1, \gamma_2) \mid \gamma_1 = [i, q, \theta], \gamma_2 = [i, q', \theta], i = 1, k\}$. The set of edges $E$ contains all pairs $(\gamma, u)$ for which $\gamma \rightarrow u$ (see Sec. 4 for the definition of $\rightarrow$; recall that $u$ is either a configuration $\gamma'$, a pair of configurations $\gamma_1\gamma_2$, or $\varepsilon$), and all pairs $((\gamma_1, \gamma_2), \gamma_i)$, for $i = 1, 2$. It follows directly from the definitions that a tree $t$ is in $\mathit{inst}(A)$ iff the initial configuration $\gamma_{0,A,t}$ is accessible in the graph $G_{A,t}$.

A node in $G_{A,t}$ can be represented by an $i$-tuples of tree nodes, for $i = 1, k$. Indeed we represent a configuration $\gamma = [i, q, \theta]$ by the $i$-tuple $(\theta(1), \ldots, \theta(i))$ ($q$ does not depend on the tree $t$ and is encoded separately), and a similarly for a pair $(\gamma_1, \gamma_2)$ (since they have the same $\theta$). The difficulty in expressing the AGAP problem for $G_{A,t}$ in MSO lies in the fact that we can no longer quantify over sets of nodes: the set $S$ in the formula above would need to have arity $i$, for $i = 1, k$. The idea is to rely on a special structure of $G_{A,t}$. Namely if two nodes described by $i$-tuples $(x_1, \ldots, x_i)$ and a $j$ tuple $(y_1, \ldots, y_j)$ are connected by an edge, then either $i = j$, or $i = j+1$, or $i = j-1$, and the tuples agree on all but the last position. This follows from the stack discipline on pebbles in $A$ (only the last pebble can be moved) and allows us to quantify independently, on different portions of the graph. The construction of the MSO formula below relies on this observation.

We next outline the construction of the MSO formula. Consider a $k$-pebble tree automaton $A = (\Sigma, Q, q_0, P)$. Let $Q = Q_1 \cup Q_2 \cup \ldots Q_k$, where $Q_i$ is the set of states controlling pebble $i$. We enumerate the states in $Q$ so that $Q = \{q_0, q_1, \ldots, q_n\}$, and:

$$\begin{aligned} Q_1 &= \{q_0, \ldots, q_{n_1}\} \\ Q_2 &= \{q_{n_1+1}, \ldots, q_{n_2}\} \\ &\ldots \\ Q_k &= \{q_{n_{k-1}+1}, \ldots, q_{n_k}\} \end{aligned}$$

We first illustrate the case when $k = 1$, so $A$ uses a single pebble. The MSO formula $\varphi$ defining acceptance by $A$ uses one unary relation $S_j$ for each state $q_j \in Q$. The formula $\varphi$ is of the form:

$$\varphi = \forall S_0. \forall S_1 \ldots \forall S_{n_1} (\textit{reverse-closed} \Longrightarrow S_0(\textit{root})) \quad (6)$$

where *reverse-closed* is a sentence stating that $S_0, S_1, \ldots, S_{n_1}$ are closed under reverse transitions of $A$ according to the *and/or* semantics. Thus, $\varphi$ states that the initial configuration of $A$ (pebble at the root in state $q_0$) is reachable in the *and/or* graph of configurations by reverse transitions. It follows that $\varphi$ holds iff $A$ accepts the tree.

The key element in the construction of $\varphi$ is of course the formula *reverse-closed.* This is straightforward when $k = 1$, since no extra pebbles are involved. For each of the transitions of $A$, *reverse-closed* includes a corresponding conjunct. Some of these are illustrated next:

- for $(a, q_u) \to (q_v, \textit{stay})$, *reverse-closed* contains the conjunct:

$$\forall\, x.(R_a(x) \wedge S_v(x)) \Longrightarrow S_u(x)$$

- for $(a, q_u) \to (q_v, \textit{down-left})$, *reverse-closed* contains the conjunct:

$$\forall x.\ \forall y.(R_a(x) \wedge \textit{succ}_1(x, y) \wedge S_v(y)) \Longrightarrow S_u(x)$$

- for $(a, q_u) \to (q_v, q_w, \textit{branch2})$, *reverse-closed* contains the conjunct:

$$\forall x.(R_a(x) \wedge S_v(x) \wedge S_w(x)) \Longrightarrow S_u(x)$$

- for $(a, q_u) \to (\textit{branch0})$, *reverse-closed* contains the conjunct:

$$\forall x.R_a(x) \Longrightarrow S_u(x) \quad (7)$$

We now extend Equation (6) to the case when $k$ is arbitrary. We will define a predicate *reverse-closed*$^{(i)}$, for each $i = 1, k$, stating that $S_{n_{i-1}+1}, \ldots, S_{n_i}$ are closed under reverse transitions of $A$. Then, the MSO formula equivalent to $A$ will be $\varphi$ in Equation (6), with *reverse-closed* replaced with *reverse-closed*$^{(1)}$. The predicate *reverse-closed*$^{(i)}$ assumes that pebbles $1, 2, \ldots, i-1$ are fixed, and their positions described by the variables $x_1, \ldots, x_{i-1}$; it only considers moves affecting pebbles $i, i+1, \ldots, k$. It also has free variables $S_0, S_1, \ldots, S_{n_{i-1}}$ describing the accessibility relation on all configurations with $i' \leq i$. For each transition $(a, \bar{b}, q^{(i)}) \to \ldots$ in $A$, *reversed-closed*$^{(i)}$ includes a corresponding conjunct. For move transitions these conjuncts are as for *reverse-closed* above, except that now they also inspect the presence/absence of the previous $i-1$ pebbles. For example, for $i = 3$ and a *stay* transition of the form $(a, 01, q_u) \to (q_v, \textit{stay})$ the corresponding conjunct in *reverse-closed*$^{(i)}$ is:

$$\forall\, x.(R_a(x) \wedge (x \neq x_1 \wedge x = x_2) \wedge S_v(x)) \Longrightarrow S_u(x).$$

In general, given $b \in \{0, 1\}$ we write $x =_b y$ to mean $x = y$ when $b = 1$ and $x \neq y$ when $b = 0$; given $\bar{b} \in \{0, 1\}^{i-1}$, we write $\textit{pebbles}_{\bar{b}}(x)$ for the formula $\bigwedge_{j=1,i-1} x =_{b_j} x_j$. Then, for a *stay* transition $(a, \bar{b}, q_u^{(i)}) \to (q_v^{(i)}, \textit{stay})$, the corresponding conjunct is:

$$\forall\, x.(R_a(x) \wedge \textit{pebbles}_{\bar{b}}(x) \wedge S_v(x)) \Longrightarrow S_u(x).$$

The other move and branch transitions are handled similarly.

The new transitions are the *pick* and *place* transitions. These determine the following conjuncts in *reverse-closed*$^{(i)}$:

- for $(a, \bar{b}, q_u^{(i)}) \to (q_v^{(i+1)}, \textit{place})$, *reverse-closed*$^{(i)}$ contains the conjunct:

$$\forall x_i.(R_a(x_i) \wedge \textit{pebbles}_{\bar{b}}(x_i) \wedge \varphi^{(i+1)} \Longrightarrow S_u(x_i))$$

where

$$\varphi^{(i+1)} = \forall S_{n_i+1} \ldots \forall S_{n_{i+1}} (\textit{reverse-closed}^{(i+1)} \\ \Longrightarrow S_v(\textit{root}))$$

Note the resemblance of $\varphi^{(i+1)}$ to Equation (6): here $q_v$ acts like an initial state for pebble $i+1$.

- for $(a, \bar{b}, q_u^{(i)}) \rightarrow (q_v^{(i-1)}, \textit{pick})$, $\textit{reverse-closed}^{(i)}$ contains the conjunct:

  $$\forall x_i.(R_a(x_i) \wedge \textit{pebbles}_{\bar{b}}(x_i) \wedge S_v(x_{i-1}) \Longrightarrow S_u(x_i))$$

  Note the resemblance to Equation (7): here $q_u$ acts like a terminal state (*branch0*) for pebble $i$.

This completes the proof of the translation of $A$ into MSO. The resulting formula $\varphi$ has a size which is exponential in $k$ (due to the replication of the $\varphi^{(i)}$ subformulas), but the quantifier depth is $k$ (for first-order variables $x_i$) and $\mid Q \mid$ (for second order variables $S_j$). Note that the stack discipline imposed on the use of pebbles is essential to the construction in the proof. $\square$