# Tree-Walking Automata and Monadic Second Order Logic

Jan-Pascal van Best

July 9, 1999

# Preface

This report describes the results of a study in the field of theoretical computer science. It was written as my "doctoraalscriptie" (Master's thesis) for my study of Computer Science at Leiden University. The graduation tutor from the Department of Computer Science was J. Engelfriet, PhD.

This report was written using the LaTeX typesetting software on a computer running Linux and the KDE desktop environment.

I would like to thank Joost Engelfriet for his constant advice and his help in conducting this study. My parents have always supported me in many ways. And, last but certainly not least, thanks to Ellen for the title page she designed and for her love and support.

Jan–Pascal

Delft, July 1998

# Contents

# Introduction

Monadic second order logic (MSO logic, [Büc60, Don70, TW68]) is a way to describe properties of graphs. In this paper we will focus on strings and trees as special cases. Monadic second order logic combines great strength with great ease of expression. Closed MSO formulas define sets of trees, while MSO formulas with one or more unbound variables define node relations on trees. One perhaps less desirable property of MSO logic is that its formulas are global in a way that they define properties of trees as a whole. Another way to define node relations is by tree-walking automata. A tree-walking automaton can compute a binary node relation by starting on one node of the tree and finishing on another node. It has already been shown that tree-walking automata that can test unary MSO formulas at each node they visit, compute the same node relations that binary MSO formulas define [BE97]. However, these unary MSO formulas define in a way also global properties of the tree, in that they can test, e.g., the labels of several nodes of the tree they walk on in one formula. It can also be shown that ordinary tree-walking automata do not compute the same node relations that binary MSO formulas recognize. Therefore, in order to compute the node relations defined by binary MSO formulas, we need to extend tree-walking automata with features that make them more powerful. We need to make sure that these features result in only local operations, i.e., the resulting automata can, during their walk on a tree, only test properties of the current node. Also, we would like these operations to be more operational and less descriptive than unary MSO formulas. These considerations lead us to the central question of this paper.

> Is it possible to define a type of tree-walking automaton, with only local operations, that computes the same binary node relations that binary MSO formulas recognize?

In this paper, we try to find an answer to this question. We start by attempting to find an answer for strings in Chapter 1. In this chapter, we introduce the concept of *pebbles*. During the walk of a string- or tree-walking automaton, the automaton can place a pebble on the current node. Later during its walk, the automaton can check for the presence of a pebble and pick it up. Variations are possible in the allowed number of pebbles, coloured pebbles and restriction in the use of pebbles. It is shown that string-walking pebble automata (with just one pebble) compute exactly the binary relations that binary MSO formulas define on strings. In the next chapter, we define some more powerful string-walking automata and show some of their properties. In Chapter 3, we move on to trees. The task of finding a proper extension for tree-walking automata proves to be a bit harder than for string-walking automata. Because it is already known [KS81] that push-down tree-walking automata recognize exactly the regular tree languages (which are the tree languages defined by closed MSO formulas), we develop tree-walking marble automata, similar in concept to push-down tree-walking automata, that also recognize exactly the regular tree languages but, unlike push-down tree-walking automata, allow for a natural definition of the computation of node relations. By adding one pebble, we obtain tree-walking marble/pebble automata. We show that these compute exactly the binary node relations that binary MSO formulas define on trees. In the last chapter, we conclude this paper and give some recommendations for future research.

# Chapter 1

# Binary MSO Formulas and String-Walking Pebble Automata

## 1.1 Preliminaries

In this section we recall some well-known concepts from formal language theory [HU79, RHE91], more specifically concerning strings, finite automata and monadic second order logic (on strings). Before doing so, we recall some standard terminology from set theory.

The set of natural numbers is $\mathbb{N} = \{0, 1, 2, \ldots\}$ and, for $m, n \in \mathbb{N}$, $[m, n] = \{i \in \mathbb{N} \mid m \leq i \leq n\}$. For a set $S$, $2^S$ is its powerset, i.e., the set of all subsets of $S$. Let $R \subseteq A \times B$ be a binary relation. The transitive reflexive closure of $R$ is denoted $R^*$. The inverse of $R$ is $R^{-1} = \{(y, x) \mid (x, y) \in R\}$. For two binary relations $R_1$ and $R_2$, their composition is $R_1 \circ R_2 = \{(x, z) \mid \exists y : (x, y) \in R_1 \text{ and } (y, z) \in R_2\}$. Note that the order of $R_1$ and $R_2$ is nonstandard. For each $a \in A$, $R(a) = \{b \in B \mid (a, b) \in R\}$ and for each $S \subseteq A$, $R(S) = \bigcup\{R(a) \mid a \in S\}$. The domain of $R$ is $\mathrm{dom}(R) = R^{-1}(B)$. A binary relation $R$ is *functional* if it is a partial function, i.e., $(x, y), (x, z) \in R$ implies $x = z$.

Let $f : A \to B$ be a function; for any $a$ and $b$, $f(a \mapsto b)$ denotes the "perturbed" function $f' : A \cup \{a\} \to B \cup \{b\}$ with $f'(a) = b$ and $f'(a') = f(a')$ for every $a' \in A$ with $a' \neq a$.

### Strings

An *alphabet* is a finite set of symbols. Let $\Sigma$ be an alphabet. A *string over* $\Sigma$ is a finite string of symbols from $\Sigma$. The empty string is denoted $\lambda$. The set of all strings over $\Sigma$ is denoted $\Sigma^*$. A subset of $\Sigma^*$ is called a *language*. If $w$ is a string over $\Sigma$, $|w|$ denotes the length of $w$. The set of positions in $w$ is denoted by $V_w = [1, |w|]$. For a string $w \in \Sigma^*$ and $i \in V_w$, we write $\mathrm{lab}_w(i) = \sigma$ if the $i$th symbol of $w$ is $\sigma$. For $k \in \mathbb{N}$, a $k$-ary *position relation* over $\Sigma$ is a subset of $\{(w, u_1, \ldots, u_k) \mid w \in \Sigma^* \text{ and } u_i \in$

$V_w$ for all $i \in [1,k]\}$. A $k$-ary position relation associates with each string $w$ a $k$-ary relation on the positions of $w$.

We define *marked* strings as follows. Let $\Sigma$ be an alphabet, and $k \geq 1$. We define $B_k = \{0,1\}^k \setminus \{0\}^k$. The alphabet $\Sigma \cup (\Sigma \times B_k)$ contains all symbols $\sigma \in \Sigma$ and the symbols $(\sigma, b_1, \ldots, b_k)$ with $b_i \in \{0,1\}$ for all $i \in [1,k]$, but without $(\sigma, 0, \ldots, 0)$. The role of the latter symbol is played by $\sigma$ itself. This alphabet is used to attach $k$ different marks to the positions in a string. Let $w \in \Sigma^*$ be a string over $\Sigma$, and let $u_1, \ldots, u_k \in V_w$. The *marked string* $w' = \mathrm{mark}(w, u_1, \ldots, u_k) \in (\Sigma \cup (\Sigma \times B_k))^*$ is the string with $\mathrm{lab}_{w'}(u) = \mathrm{lab}_w(u)$ if $u \neq u_i$ for all $i \in [1,k]$, and $\mathrm{lab}_{w'}(u) = (\mathrm{lab}_w(u), (u = u_1), \ldots, (u = u_k))$ otherwise (where $(u = u_i) = 1$ iff $u$ equals $u_i$).

## Finite Automata

Let $\Sigma$ be an alphabet. A *finite automaton* over $\Sigma$ is a quintuple $M = (Q, \Sigma, \delta, I, F)$, where $Q$ is a finite set of states, $\Sigma$ is the input alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states. The elements of $\delta$ are called transitions. For every string $w \in \Sigma^*$, the state transition relation $R_M(w) \subseteq Q \times Q$ is defined as follows. For $\sigma \in \Sigma$, $R_M(\sigma) = \{(p,q) \mid (p, \sigma, q) \in \delta\}$. For $\sigma_1, \ldots, \sigma_n \in \Sigma$, $R_M(\sigma_1 \cdots \sigma_n) = R_M(\sigma_1) \circ \cdots \circ R_M(\sigma_n)$. The language recognized by $M$ is $L(M) = \{w \in \Sigma^* \mid (q_{\mathrm{in}}, q_{\mathrm{fin}}) \in R_M(w)$ for some $q_{\mathrm{in}} \in I, q_{\mathrm{fin}} \in F\}$. A language is *regular* if it is recognized by a finite automaton. The class of all regular languages is named REG.

## Monadic Second Order Logic (on strings)

Monadic second order logic can be used to describe properties of strings [Büc60]. For an alphabet $\Sigma$, we define the language MSOL($\Sigma$) of MSO formulas over $\Sigma$. This language has position variables $x, y, \ldots$ and position-set variables $X, Y, \ldots$. For a given string $w \in \Sigma^*$, position variables range over $V_w = [1, |w|]$ and position-set variables range over the subsets of $V_w$.

There are three types of atomic formulas in MSOL($\Sigma$): $\mathrm{lab}_\sigma(x)$, for every $\sigma \in \Sigma$, signifies that at position $x$ there is a symbol $\sigma$. When MSO logic is expanded to describe properties of trees and graphs, $\mathrm{lab}_\sigma(x)$ means that node $x$ is *labeled* $\sigma$, hence the name $\mathrm{lab}_\sigma$. The atomic formula $\mathrm{pre}(x,y)$ signifies that position $x$ comes directly before $y$, i.e. $y = x + 1$; and $x \in X$ signifies that $x$ is an element of $X$. The connectives used to build formulas from these atomic formulas are $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, with the usual meaning. Both position variables and position-set variables can be quantified with $\exists$ and $\forall$. We will make use of the following abbreviations of MSO formulas:

$$\begin{aligned}
\mathrm{head}(x) &= \forall y\, (\neg \mathrm{pre}(y,x)) \\
\mathrm{tail}(x) &= \forall y\, (\neg \mathrm{pre}(x,y)) \\
\mathrm{true}(x) &= \exists X\, (x \in X) \\
\mathrm{false}(x) &= \neg \mathrm{true}(x)
\end{aligned}$$

If an MSO formula $\phi$ has free variables, say $x, y, X$ and no others, we write $\phi(x,y,X)$ to indicate the free variables of $\phi$. For every $k \in \mathbb{N}$, the set of

MSO formulas over $\Sigma$ with $k$ free position variables and no free position-set variables is denoted $\mathrm{MSOL}_k(\Sigma)$, or the set of $k$-ary MSO formulas. For a closed formula $\phi \in \mathrm{MSOL}_0(\Sigma)$ and a string $w \in \Sigma^*$, we write $w \models \phi$ if $w$ satisfies $\phi$. The language defined by $\phi$ is $L(\phi) = \{w \in \Sigma^* \mid w \models \phi\}$. $L(\phi)$ is called an MSO definable language. The class of all MSO definable languages is named MSO.

Given a string $w$, a valuation function $\nu$ is a function that maps each position variable to a position $u \in V_w$ and each position-set variable to a subset $U$ of $V_w$. Let $\phi$ be an MSO formula. We write $(w, \nu) \models \phi$, if $\phi$ holds in $w$, where the free variables of $\phi$ are assigned values according to the valuation function $\nu$. If $\phi$ has free variables $x, y, X$, we write $(w, u, v, U) \models \phi(x, y, X)$ for $(w, \nu) \models \phi$, where $\nu(x) = u$, $\nu(y) = v$ and $\nu(X) = U$.

Let $\phi(x_1, \ldots, x_k) \in \mathrm{MSOL}_k(\Sigma)$ be an MSO formula with $k$ free positions variables (and no free position-set variables). For each string $w \in \Sigma^*$, the $k$-ary relation that $\phi$ defines on the positions of $w$ is

$$R_w(\phi) = \{(u_1, \ldots, u_k) \in V_w^k \mid (w, u_1, \ldots, u_k) \models \phi(x_1, \ldots, x_k)\}.$$

The MSO formula $\phi(x_1, \ldots, x_k)$ defines the position relation

$$R(\phi) = \{(w, u_1, \ldots, u_k) \mid w \in \Sigma^*, (u_1, \ldots, u_k) \in R_w(\phi)\}.$$

Büchi [Büc60] proved the following classical result.

**Proposition 1** *A language is MSO definable if and only if it is regular.*

In [BE97], the following (well-known) results are proven for MSO formulas on trees. They also hold for the special case of strings. Corollary 3 is the result of Lemma 2 in the case $j = k$.

**Lemma 2** *Let $\Sigma$ be an alphabet, $k \geq 1$, and $j \in [1, k]$. For every formula $\phi(x_1, \ldots, x_k) \in MSOL_k(\Sigma)$ there is a formula $\psi(x_{j+1}, \ldots, x_k) \in MSOL_{k-j}(\Sigma \cup (\Sigma \times B_j))$ such that, for all $w \in \Sigma^*$ and $u_1, \ldots, u_k \in V_w$,*

$$(w, u_1, \ldots, u_k) \models \phi(x_1, \ldots, x_k) \textit{ iff } (\mathit{mark}(w, u_1, \ldots, u_j), u_{j+1}, \ldots, u_k) \models \psi(x_{j+1}, \ldots, x_k),$$

*and vice versa, i.e., for every formula $\psi(x_{j+1}, \ldots, x_k) \in MSOL_{k-j}(\Sigma \cup (\Sigma \times B_j))$ there is a formula $\phi(x_1, \ldots, x_k) \in MSOL_k(\Sigma)$ such that the above equivalence holds for all $w \in \Sigma^*$ and $u_1, \ldots, u_k \in V_w$.*

**Corollary 3** *Let $\Sigma$ be an alphabet and $k \geq 1$. For every formula $\phi(x_1, \ldots, x_k) \in MSOL_k(\Sigma)$ there is a formula $\psi \in MSOL_0(\Sigma \cup (\Sigma \times B_k))$ such that, for all $w \in \Sigma^*$ and $u_1, \ldots, u_k \in V_w$,*

$$(w, u_1, \ldots, u_k) \models \phi(x_1, \ldots, x_k) \textit{ iff } \mathit{mark}(w, u_1, \ldots, u_k) \models \psi,$$

*and vice versa, i.e., for every MSO formula $\psi \in MSOL_0(\Sigma \cup (\Sigma \times B_k))$ there is a formula $\phi(x_1, \ldots, x_k) \in MSOL_k(\Sigma)$ such that the above equivalence holds for all $w \in \Sigma^*$ and $u_1, \ldots, u_k \in V_w$.*

From Corollary 3 and Proposition 1 follows the following lemma.

**Lemma 4** *Let $\Sigma$ be an alphabet and let $k \geq 1$. For every MSO formula $\phi(x_1, \ldots, x_k) \in MSOL_k(\Sigma)$ there exists a finite automaton $M$ over $\Sigma \cup (\Sigma \times B_k)$ such that, for all $w \in \Sigma^*$ and $u_1, \ldots, u_k \in V_w$,*

$$(w, u_1, \ldots, u_k) \models \phi(x_1, \ldots, x_k) \textit{ iff } \mathit{mark}(w, u_1, \ldots, u_k) \in L(M).$$

## 1.2 String-Walking Automata

String-walking automata are known in the literature as two-way finite automata (see [HU79, Section 2.6]). We will call them string-walking automata to stress both the differences and the similarities with tree-walking automata. String-walking automata are like the well-known finite automata, except that they can walk both forwards and backwards along the string they are examining (hence the name two-way finite automata). At any moment in the execution of a string-walking automaton, the automaton is in one state and at one position of the string. Let $\Sigma$ be an alphabet. Syntactically, a string-walking automaton over $\Sigma$ is a finite automaton with a special set of *directives* as input alphabet. We define the set of directives as

$$D_{\mathrm{SWA}}(\Sigma) = \{\leftarrow, \rightarrow, \mathrm{head}, \neg\mathrm{head}, \mathrm{tail}, \neg\mathrm{tail}\} \cup \{\mathrm{lab}_\sigma \mid \sigma \in \Sigma\}.$$

The meaning of the directives is as follows:

- $\leftarrow$ means go back to the previous position in the input string.
- $\rightarrow$ means go to the next position in the input string.
- head checks whether the current position is the first position in the string.
- tail checks whether the current position is the last position in the string.
- $\neg$head and $\neg$tail are the negations of head and tail respectively.
- $\mathrm{lab}_\sigma$ checks whether the current position is labeled with $\sigma$.

Formally, we define for each string $w \in \Sigma^*$ and each directive $d \in D_{\mathrm{SWA}}(\Sigma)$ the following binary relation $R_w(d)$ on $V_w$:

$$\begin{array}{rcl}
R_w(\leftarrow) & = & \{(i, i-1) \mid 1 < i \leq |w|\} \\
R_w(\rightarrow) & = & \{(i, i+1) \mid 1 \leq i < |w|\} \\
R_w(\mathrm{lab}_\sigma) & = & \{(i, i) \mid 1 \leq i \leq |w| \text{ and the } i\text{th symbol of } w \text{ is } \sigma\} \\
R_w(\mathrm{head}) & = & \{(1, 1)\} \\
R_w(\neg\mathrm{head}) & = & \{(i, i) \mid 1 < i \leq |w|\} \\
R_w(\mathrm{tail}) & = & \{(n, n) \mid n = |w|\} \\
R_w(\neg\mathrm{tail}) & = & \{(i, i) \mid 1 \leq i < |w|\}
\end{array}$$

A *string-walking automaton over* $\Sigma$ is a finite automaton $A$ over $D_{\mathrm{SWA}}(\Sigma)$. For a string-walking automaton $A = (Q, D_{\mathrm{SWA}}(\Sigma), \delta, I, F)$ and a string $w \in \Sigma^*$, an element $(q, u)$ of $Q \times V_w$ is a *configuration* of the automaton. The configuration $(q, u) \in Q \times V_w$ signifies that $A$ is in state $q$ at position $u$. We say that the *current position* of the automaton on $w$ is $u$. The set of all configurations of $A$ and $w$ is denoted by $\mathbb{C}_{A,w}$.

Let $w \in \Sigma^*$. One step of $A = (Q, D_{\mathrm{SWA}}(\Sigma), \delta, I, F)$ on $w$ is defined by the binary relation $\twoheadrightarrow_{A,w}$ on the set of configurations, as follows. For every $(q, u), (q', u') \in \mathbb{C}_{A,w}$,

$$(q, u) \twoheadrightarrow_{A,w} (q', u') \text{ iff } \exists d \in D_{\mathrm{SWA}}(\Sigma) : (q, d, q') \in \delta \text{ and } (u, u') \in R_w(d).$$

For each string $w \in \Sigma^*$, $A$ computes the binary relation

$$R_w(A) = \left\{(u, v) \in V_w \times V_w \mid (q, u) \twoheadrightarrow^*_{A,w} (q', v) \text{ for some } q \in I \text{ and } q' \in F\right\}.$$

Thus, $A$ computes the position relation

$$R(A) = \{(w, u, v) \mid w \in \Sigma^*, (u, v) \in R_w(A)\}.$$

We define the *language* that $A$ recognizes as all strings $w$ with the property that there is a walk of $A$ on $w$ starting at the beginning of the string in an initial state and ending in a final state, or

$$L(A) = \{w \in \Sigma^* \mid \text{there exists } v \in V_w \text{ such that } (1, v) \in R_w(A)\}.$$

The class of all languages recognized by a string-walking automaton is named SWA.

In the following we will also allow transitions of the form $(p, s, q)$ with $s \in D_{\mathrm{SWA}}(\Sigma)^*$. If $s = d_1 d_2 \cdots d_n$, we define, for all $w \in \Sigma^*$,

$$R_w(s) = R_w(d_1) \circ \cdots \circ R_w(d_n).$$

These extended directives can easily be implemented by adding extra transitions and states.

A string-walking automaton $A = (Q, D_{\mathrm{SWA}}(\Sigma), \delta, I, F)$ over $\Sigma$ is *deterministic* if the following three conditions hold:

1. $\#I = 1$.
2. If $(p, d, q) \in \delta$, then $p \notin F$.
3. For all distinct transitions $(p, d_1, q_1), (p, d_2, q_2) \in \delta$, $d_1$ and $d_2$ are two mutually exclusive directives in $D_{\mathrm{SWA}}(\Sigma)$.

Two directives $d_1, d_2 \in D_{\mathrm{SWA}}(\Sigma)$ are mutually exclusive exactly if, for all $w \in \Sigma^*$ and $u \in V_w$, $\mathrm{dom}(R_w(d_1)) \cap \mathrm{dom}(R_w(d_2)) = \emptyset$. The following pairs of directives in $D_{\mathrm{SWA}}(\Sigma)$ are mutually exclusive:

- $\{\rightarrow, \mathrm{tail}\}, \{\leftarrow, \mathrm{head}\}$
- $\{\mathrm{head}, \neg\mathrm{head}\}, \{\mathrm{tail}, \neg\mathrm{tail}\}$
- $\{\mathrm{lab}_{\sigma_1}, \mathrm{lab}_{\sigma_2}\}$ with $\sigma_1 \neq \sigma_2$

## 1.3 String-Walking Pebble Automata

We will extend the concept of string-walking automata with a pebble [BH65]. A string-walking pebble automaton is able to drop its pebble at its current position in the string, pick it up at any (later) time, if it is at the same position again, and check whether or not the pebble is present at the automaton's current position. Let $\Sigma$ be an alphabet. The directives for a string-walking pebble automaton are

$$\begin{aligned} D_{\mathrm{SWPA}}(\Sigma) &= D_{\mathrm{SWA}}(\Sigma) \cup \{\mathrm{put}, \mathrm{lift}, \mathrm{here}, \neg\mathrm{here}\} \\ &= \{\leftarrow, \rightarrow, \mathrm{head}, \neg\mathrm{head}, \mathrm{tail}, \neg\mathrm{tail}, \mathrm{put}, \mathrm{lift}, \mathrm{here}, \neg\mathrm{here}\} \cup \{\mathrm{lab}_\sigma \mid \sigma \in \Sigma\} \end{aligned}$$

The meaning of the new directives is as follows:

- put means drop the pebble at the current position in the string.
- lift means lift the pebble from the current position in the string (only if it is there now).
- here checks whether the pebble is at the current position.
- $\neg$here is the negation of here.

For all strings $w \in \Sigma^*$, we will denote the pebble positions with elements of $P_w = V_w \cup \{\bot\}$. Here $\bot$ denotes that the pebble is not placed at any position of the string, i.e., the automaton "carries" the pebble. We now need to extend the binary relations $R_w(d)$ (for $d \in D_{\mathrm{SWA}}(\Sigma)$) to be on pairs $(u, p) \in V_w \times P_w$ to take pebble-handling into account. Let, for all strings $w \in \Sigma^*$ and directives $d \in D_{\mathrm{SWA}}(\Sigma)$, $\tilde{R}_w(d)$ denote the binary relation $R_w(d)$ as defined for string-walking automata in the previous section. We now define $R_w(d)$ for string-walking pebble automata as follows:

$$R_w(d) = \{((i,p),(i',p)) \mid (i,i') \in \tilde{R}_w(d) \text{ and } p \in P_w\}.$$

This means that the directives from $D_{\mathrm{SWA}}(\Sigma)$ do not alter the pebble position.

The relations for the new directives are as follows.

$$\begin{aligned}
R_w(\text{put}) &= \{((i,\bot),(i,i)) \mid 1 \le i \le |w|\} \\
R_w(\text{lift}) &= \{((i,i),(i,\bot)) \mid 1 \le i \le |w|\} \\
R_w(\text{here}) &= \{((i,i),(i,i)) \mid 1 \le i \le |w|\} \\
R_w(\neg\text{here}) &= \{((i,p),(i,p)) \mid 1 \le i \le |w| \text{ and } p \ne i\}
\end{aligned}$$

Let $\Sigma$ be an alphabet. A *string-walking pebble automaton* is a finite automaton $A$ over $D_{\mathrm{SWPA}}(\Sigma)$. For a string-walking pebble automaton $A = (Q, D_{\mathrm{SWPA}}(\Sigma), \delta, I, F)$ and a string $w \in \Sigma^*$, an element $(q, u, p) \in Q \times V_w \times P_w$ is a *configuration* of the automaton. The set of all configurations of $A$ and $w$ is denoted by $\mathbb{C}_{A,w}$. A configuration $(q, u, p)$ signifies that $A$ is in state $q$ at position $u$, while the pebble is at position $p$.

The step relation $\twoheadrightarrow_{A,w}$ is extended in the obvious way. For a string-walking pebble automaton $A$, a string $w \in \Sigma^*$ and configurations $(q,u,p),(q',u',p') \in \mathbb{C}_{A,w}$,

$$(q,u,p) \twoheadrightarrow^*_{A,w} (q',u',p') \text{ iff } \exists d \in D_{\mathrm{SWPA}}(\Sigma) : (q,d,q') \in \delta \text{ and } ((u,p),(u',p')) \in R_w(d).$$

For the relation that $A$ computes, we now demand that $A$ begins and ends with the pebble not on the string, or "in its pocket". For $w \in \Sigma^*$,

$$R_w(A) = \left\{(u,v) \in V_w \times V_w \mid (q,u,\bot) \twoheadrightarrow^*_{A,w} (q',v,\bot) \text{ for some } q \in I \text{ and } q' \in F\right\}.$$

The definitions of $R(A)$ and $L(A)$ remain the same. The class of all languages recognized by a string-walking pebble automaton is named SWPA.

Transitions of the form $(p, s, q)$ with $s \in D_{\mathrm{SWPA}}(\Sigma)^*$ are treated as with string-walking automata. The definition of a *deterministic* string-walking pebble automaton is similar to that of a deterministic string-walking automaton. The only difference is that the directives are now taken from $D_{\mathrm{SWPA}}(\Sigma)$. The following pairs of directives in $D_{\mathrm{SWPA}}(\Sigma)$ are mutually exclusive:

- $\{\rightarrow, \text{tail}\}, \{\leftarrow, \text{head}\}$
- $\{\text{here}, \text{put}\}, \{\text{lift}, \text{put}\}, \{\neg\text{here}, \text{lift}\}$
- $\{\text{head}, \neg\text{head}\}, \{\text{tail}, \neg\text{tail}\}, \{\text{here}, \neg\text{here}\}$
- $\{\text{lab}_{\sigma_1}, \text{lab}_{\sigma_2}\}$ with $\sigma_1 \neq \sigma_2$

## 1.4 String-Walking Automata with MSO Tests

Another way to extend string-walking automata (as done in [BE97] for tree-walking automata, see also Section 3.7) is to allow the automaton to test any MSO definable property of the current position of the string, using MSO formulas with one free position variable. This extension also requires the introduction of new directives. Let $\Sigma$ be an alphabet. The set of directives over $\Sigma$ is

$$D_{\text{SWA+M}}(\Sigma) = \{\leftarrow, \rightarrow\} \cup \text{MSOL}_1(\Sigma).$$

Note that head, tail, $\text{lab}_\sigma$, etc. are MSO definable properties so they need not be explicitely stated in the definition of $D_{\text{SWA+M}}(\Sigma)$. Note also that $D_{\text{SWA+M}}(\Sigma)$ is an infinite set. Therefore we need to define for each string-walking automaton with MSO tests a finite subset of $D_{\text{SWA+M}}(\Sigma)$ as the directives used by the automaton. For the directive $d = \psi(x)$ with $\psi \in \text{MSOL}_1(\Sigma)$, we define the following binary relation on $V_w$ ($w \in \Sigma^*$):

$$R_w(\psi(x)) = \{(i,i) \mid (w,i) \models \psi(x)\}.$$

A *string-walking automaton with MSO tests* over $\Sigma$ is a finite automaton $A$ over a finite subset of $D_{\text{SWA+M}}(\Sigma)$. The definitions of configurations, $\twoheadrightarrow_{A,w}$, $R_w(A)$, etc. are unchanged with respect to the definitions for (ordinary) string-walking automata. The definitions of deterministic string-walking automata and mutually exclusive directives are also unchanged, although it is no longer possible to list all mutually exclusive pairs of directives. Note that it is decidable whether a given pair of directives in $\text{MSOL}_1(\Sigma)$ is mutually exclusive.

The class of all languages recognized by a string-walking automaton with MSO tests is named SWA+M.

In [BE97] the following results are proven for trees. They also hold for the special case of strings.

**Proposition 5** *The following three statements hold:*

- *SWA+M=REG*
- *For every alphabet* $\Sigma$ *and every binary MSO formula* $\phi(x, y) \in MSOL_2(\Sigma)$, *there exists a string-walking automaton with MSO tests A such that* $R(A) = R(\phi)$.
- *For every alphabet* $\Sigma$ *and every string-walking automaton with MSO tests A over* $\Sigma$, *there exists a binary MSO formula* $\phi(x, y) \in MSOL_2(\Sigma)$ *such that* $R(\phi) = R(A)$.

## 1.5 From MSO Formulas to String-Walking Pebble Automata

Both MSO formulas with two position variables and string-walking pebble automata compute binary position relations. In this section, we show that every binary position relation that can be defined by an MSO formula, can also be computed by a string-walking pebble automaton. In Section 1.8 we will show that the reverse is also true, i.e., every binary position relation that can be computed by a string-walking pebble automaton can also be defined by a binary MSO formula.

Let $\Sigma$ be an alphabet and let $\phi(x, y)$ be a binary MSO formula over $\Sigma$. We have seen (Lemma 4) that there exists a finite automaton $M$ over $\Sigma \cup (\Sigma \times B_2)$ such that, for all $w \in \Sigma^*$ and $u, v \in V_w$,

$$(w, u, v) \models \phi(x, y) \text{ iff } \mathrm{mark}(w, u, v) \in L(M).$$

The following lemma states that there is a string-walking pebble automaton that computes the same relation as $\phi(x, y)$ defines.

**Lemma 6** *For every finite automaton M over* $\Sigma \cup (\Sigma \times B_2)$ *there exists a string-walking pebble automaton A over* $\Sigma$ *such that, for all* $w \in \Sigma^*$ *and* $u, v \in V_w$,

$$\mathit{mark}(w, u, v) \in L(M) \text{ iff } (u, v) \in R_w(A).$$

*Proof.* Let $M = (Q_M, \Sigma \cup (\Sigma \times B_2), \delta_M, I_M, F_M)$ be a finite automaton over $\Sigma \cup (\Sigma \times B_2)$. We will define a string-walking pebble automaton over $\Sigma$, $A$, that simulates $M$, using the pebble to simulate the marked positions in $M$. $A$ is defined as the (intuitive) union of three automata $A_1, A_2, A_3$. We define $A$ as $(Q_1 \cup Q_2 \cup Q_3, D_{\mathrm{SWPA}}(\Sigma), \delta_1 \cup \delta_2 \cup \delta_3, I_1 \cup I_2 \cup I_3, F_1 \cup F_2 \cup F_3)$. $A_1$ deals with the case that $u$ comes "before" $v$ in $w$, $A_2$ deals with the case that $v$ comes first and $A_3$ deals with the case $u = v$. For $1 \leq i \leq 3$, $A_i = (Q_i, D_{\mathrm{SWPA}}(\Sigma), \delta_i, I_i, F_i)$.

$A_1$ simulates $M$, assuming that $u$ comes before $v$ in $w$. $A_1$ first drops its pebble at the start position $u$. This makes it possible to determine later where the automaton started. Then it walks to the head of the string. Next it simulates $M$ in three stages. In the first stage of the simulation, it simulates $M$ (only taking into account transitions of the form $(p, \sigma, q)$ with $\sigma \in \Sigma$) until it finds the pebble. It picks up the pebble and treats the symbol $\sigma$ on this position as $(\sigma, 1, 0)$ in $M$. The automaton now continues simulating $M$ (the second stage of the simulation), again without transitions with labels in $\Sigma \times B_2$. Then, nondeterministically, $A_1$ drops the pebble at some position $v$ and treats the symbol $\sigma$ at this position as $(\sigma, 0, 1)$ in $M$. $A$ continues simulating $M$ (the third stage) until it reaches the end of the string. If it reaches the end of the string in a final state of $M$, the choice for $v$ was a good choice. The automaton then backs up until it finds the pebble at position $v$. This ends $A_1$.

We use the following states in $A_1$.

- $\mathrm{in}_1$ is the initial state.
- $\mathrm{tobegin}_1$ is used to walk to the head of the string.
- $q_{1,i}$ (for $q \in Q_M$) is used in the $i$th stage of the simulation.
- $\mathrm{backup}_1$ is used to back up to $v$ after the three stages of simulation.
- $\mathrm{final}_1$ is the final state.

Formally, $A_1 = (Q_1, D_{\mathrm{SWPA}}(\Sigma), \delta_1, I_1, F_1)$ is constructed as follows:

$$
\begin{aligned}
Q_1 &= \{\mathrm{in}_1, \mathrm{tobegin}_1, \mathrm{backup}_1, \mathrm{final}_1\} \cup \\
&\quad \{q_{1,i} \mid q \in Q_M, 1 \le i \le 3\} \\
\delta_1 &= \{(\mathrm{in}_1, \mathrm{put}, \mathrm{tobegin}_1), (\mathrm{tobegin}_1, \leftarrow, \mathrm{tobegin}_1)\} \cup \\
&\quad \{(\mathrm{tobegin}_1, \mathrm{head}, q_{1,1}) \mid q \in I_M\} \cup \\
&\quad \{(p_{1,1}, (\mathrm{lab}_\sigma, \rightarrow), q_{1,1}) \mid (p, \sigma, q) \in \delta_M\} \cup \\
&\quad \{(p_{1,1}, (\mathrm{lift}, \mathrm{lab}_\sigma, \rightarrow), q_{1,2}) \mid (p, (\sigma, 1, 0), q) \in \delta_M\} \cup \\
&\quad \{(p_{1,2}, (\mathrm{lab}_\sigma, \rightarrow), q_{1,2}) \mid (p, \sigma, q) \in \delta_M\} \cup \\
&\quad \{(p_{1,2}, (\mathrm{put}, \mathrm{lab}_\sigma, \rightarrow), q_{1,3}) \mid (p, (\sigma, 0, 1), q) \in \delta_M\} \cup \\
&\quad \{(p_{1,2}, (\mathrm{put}, \mathrm{lab}_\sigma, \mathrm{tail}), \mathrm{backup}_1) \mid (p, (\sigma, 0, 1), q_{\mathrm{fin}}) \in \delta_M \text{ for some } q_{\mathrm{fin}} \in F_M\} \cup \\
&\quad \{(p_{1,3}, (\mathrm{lab}_\sigma, \rightarrow), q_{1,3}) \mid (p, \sigma, q) \in \delta_M\} \cup \\
&\quad \{(p_{1,3}, (\mathrm{lab}_\sigma, \mathrm{tail}), \mathrm{backup}_1) \mid (p, \sigma, q_{\mathrm{fin}}) \in \delta_M \text{ for some } q_{\mathrm{fin}} \in F_M\} \cup \\
&\quad \{(\mathrm{backup}_1, \leftarrow, \mathrm{backup}_1), (\mathrm{backup}_1, \mathrm{lift}, \mathrm{final}_1)\} \\
I_1 &= \{\mathrm{in}_1\} \\
F_1 &= \{\mathrm{final}_1\}
\end{aligned}
$$

$A_2$ is similar to $A_1$, only here the automaton, after dropping its pebble on $u$, first walks to the *right* until the end of the string, then simulates $M$ (inverted), starting in a state corresponding to a state in $F_M$, until it finds the pebble at $u$. It picks up the pebble and treats the symbol $\sigma$ at this position as $(\sigma, 0, 1)$ in $M$. $A_2$ continues with the second stage of the simulation. Then, nondeterministically, $A_2$ drops the pebble at some position $v$ and continues with the third stage of the simulation until the beginning of the string. If it reaches the beginning of the string in an initial state of $M$, $A_2$ walks back to the pebble at $v$ and the simulation is finished.

Formally, $A_2 = (Q_2, D_{\mathrm{SWPA}}(\Sigma), \delta_2, I_2, F_2)$ is constructed as follows:

$$
\begin{aligned}
Q_2 &= \{\mathrm{in}_2, \mathrm{toend}_2, \mathrm{backup}_2, \mathrm{final}_2\} \cup \\
&\quad \{q_{2,i} \mid q \in Q_M, 1 \le i \le 3\} \\
\delta_2 &= \{(\mathrm{in}_2, \mathrm{put}, \mathrm{toend}_2), (\mathrm{toend}_2, \rightarrow, \mathrm{toend}_2)\} \cup \\
&\quad \{(\mathrm{toend}_2, \mathrm{tail}, q_{2,1}) \mid q \in F_M\} \cup \\
&\quad \{(q_{2,1}, (\mathrm{lab}_\sigma, \leftarrow), p_{2,1}) \mid (p, \sigma, q) \in \delta_M\} \cup \\
&\quad \{(q_{2,1}, (\mathrm{lift}, \mathrm{lab}_\sigma, \leftarrow), p_{2,2}) \mid (p, (\sigma, 1, 0), q) \in \delta_M\} \cup \\
&\quad \{(q_{2,2}, (\mathrm{lab}_\sigma, \leftarrow), p_{2,2}) \mid (p, \sigma, q) \in \delta_M\} \cup \\
&\quad \{(q_{2,2}, (\mathrm{put}, \mathrm{lab}_\sigma, \leftarrow), p_{2,3}) \mid (p, (\sigma, 0, 1), q) \in \delta_M\} \cup \\
&\quad \{(q_{2,2}, (\mathrm{put}, \mathrm{lab}_\sigma, \mathrm{head}), \mathrm{backup}_2) \mid (p, (\sigma, 0, 1), q_{\mathrm{in}}) \in \delta_M \text{ for some } q_{\mathrm{in}} \in I_M\} \cup \\
&\quad \{(q_{2,3}, (\mathrm{lab}_\sigma, \leftarrow), p_{2,3}) \mid (p, \sigma, q) \in \delta_M\} \cup \\
&\quad \{(q_{2,3}, (\mathrm{lab}_\sigma, \mathrm{head}), \mathrm{backup}_2) \mid (p, \sigma, q_{\mathrm{in}}) \in \delta_M \text{ for some } q_{\mathrm{in}} \in I_M\} \cup
\end{aligned}
$$

$$
\begin{array}{rcl}
 & & \{(\mathrm{backup}_2, \rightarrow, \mathrm{backup}_2), (\mathrm{backup}_2, \mathrm{lift}, \mathrm{final}_2)\} \\
I_2 & = & \{\mathrm{in}_2\} \\
F_2 & = & \{\mathrm{final}_2\}
\end{array}
$$

$A_3$ looks again like the previous two parts of $A$. $A_3$ however assumes that $u = v$, which implies that the only useful transitions with labels in $\Sigma \times B_2$ are the ones of the form $(p, (\sigma, 1, 1), q) \in \delta_M$. So $A_3$ first drops its pebble at position $u$ $(= v)$, then (like $A_1$) walks to the head of the string and simulates $M$ until it gets to the starting position (the pebble) again. It does not lift the pebble, because it needs to know the position of $u = v$. $A_3$ treats the symbol $\sigma$ at this position as $(\sigma, 1, 1)$ in $M$. Then it continues with the second stage of the simulation of $M$ until the end of the string. If it arrives there in a final state of $M$, $A_3$ backs up to the pebble which ends the simulation of $M$.

Formally, $A_3 = (Q_3, D_{\mathrm{SWPA}}(\Sigma), \delta_3, I_A, F_A)$ is constructed as follows:

$$
\begin{array}{rcl}
Q_3 & = & \{\mathrm{in}_3, \mathrm{tobegin}_3, \mathrm{backup}_3, \mathrm{final}_3\} \cup \\
 & & \{q_{3,i} \mid q \in Q_M, 1 \leq i \leq 2\} \\
\delta_3 & = & \{(\mathrm{in}_3, \mathrm{put}, \mathrm{tobegin}_3), (\mathrm{tobegin}_3, \leftarrow, \mathrm{tobegin}_3)\} \cup \\
 & & \{(\mathrm{tobegin}_3, \mathrm{head}, q_{3,1}) \mid q \in I_M\} \cup \\
 & & \{(p_{3,1}, (\mathrm{lab}_\sigma, \rightarrow), q_{3,1}) \mid (p, \sigma, q) \in \delta_M\} \cup \\
 & & \{(p_{3,1}, (\mathrm{here}, \mathrm{lab}_\sigma, \rightarrow), q_{3,2}) \mid (p, (\sigma, 1, 1), q) \in \delta_M\} \cup \\
 & & \{(p_{3,1}, (\mathrm{here}, \mathrm{lab}_\sigma, \mathrm{tail}), \mathrm{backup}_3) \mid (p, (\sigma, 1, 1), q_{\mathrm{fin}}) \in \delta_M \text{ for some } q_{\mathrm{fin}} \in F_M\} \cup \\
 & & \{(p_{3,2}, (\mathrm{lab}_\sigma, \rightarrow), q_{3,2}) \mid (p, \sigma, q) \in \delta_M\} \cup \\
 & & \{(p_{3,2}, (\mathrm{lab}_\sigma, \mathrm{tail}), \mathrm{backup}_3) \mid (p, \sigma, q_{\mathrm{fin}}) \in \delta_M \text{ for some } q_{\mathrm{fin}} \in F_M\} \cup \\
 & & \{(\mathrm{backup}_3, \leftarrow, \mathrm{backup}_3), (\mathrm{backup}_3, \mathrm{lift}, \mathrm{final}_3)\} \\
I_3 & = & \{\mathrm{in}_3\} \\
F_3 & = & \{\mathrm{final}_3\}
\end{array}
$$

$\square$

Using this lemma the following theorem becomes trivial to prove.

**Theorem 7** *For each formula $\phi(x, y) \in MSOL_2(\Sigma)$, there exists a string-walking pebble automaton $A$ over $\Sigma$ such that $R(\phi) = R(A)$.*

## 1.6 Another proof

There is another way to prove Theorem 7. This method makes use of Proposition 5 to simulate the binary MSO formula by a string-walking automaton with MSO tests. The MSO tests are then simulated by making use of a pebble. Let $\Sigma$ be an alphabet, and let $\phi(x, y) \in \mathrm{MSOL}_2(\Sigma)$ be an MSO formula with two free position variables. Then there exists a string-walking automaton with MSO tests $A = (Q_A, \Delta_A, \delta_A, I_A, F_A)$ with $\Delta_A$ a finite subset of $D_{\mathrm{SWA+M}}(\Sigma)$ such that $R(\phi) = R(A)$ (Proposition 5). We will construct a string-walking pebble automaton $A' = (Q', D_{\mathrm{SWPA}}(\Sigma), \delta', I', F')$ such that $R(A') = R(A) = R(\phi)$. We need the following result, which is Lemma 4 for the case $k = 1$.

**Lemma 8** *Let $\Sigma$ be an alphabet. For every MSO test $\psi(x) \in MSOL_1(\Sigma)$, there exists a finite automaton $M$ over $\Sigma \cup (\Sigma \times B_1)$ such that, for all $w \in \Sigma^*$ and $u \in V_w$,*

$$(w, u) \models \psi(x) \textit{ iff } \textit{mark}(w, u) \in L(M).$$

We now construct $A'$. In first approximation, $A'$ equals $A$ for transitions $(p, d, q)$ with $d \in \{\leftarrow, \rightarrow\}$. We define $T = \{(p, d, q) \in \delta_A \mid d \in \mathrm{MSOL}_1(\Sigma)\}$, the collection of all transitions with MSO tests in $A$. For each transition $\tau = (p, \psi(x), q) \in T$, we use Lemma 8 to construct the finite automaton $M_\tau$ that calculates the MSO test $\psi(x)$. We will use this automaton to construct a "subroutine" in $A'$ that simulates $\tau$. The idea is as follows, similar to the construction of $A_3$ in the proof of Lemma 6. $A'$ first drops its pebble. Then it walks to the head of the string. From there, it checks the MSO test by following $M_\tau$. Any transitions in $\delta_{M_\tau}$ with labels *not* in $\Sigma$ are handled by using the pebble. The only possible form of such a transition is $(p, (\sigma, 1), q)$. In place of this transition, we add transitions to check whether the position in the string where the automaton is now is marked, i.e., the pebble is there. Then the symbol $\sigma$ is checked and the automaton moves on to the next position in the string in state $q$. If $M_\tau$ reaches a final state at the end of the string, $A'$ moves back to the position where it dropped the pebble and lifts it. At this moment the MSO test $\psi(x)$ is successfully verified by the automaton.

The following states are used for $A'$. First, all states from $A$, $Q_A$. Then we need extra states for each MSO test. For all transitions $\tau \in T$ we use, next to the states of $M_\tau$, $\mathrm{tobegin}_\tau$ to move to the beginning of the string and $\mathrm{backup}_\tau$ to back up to the pebble, the place where the simulation of $M_\tau$ started. We assume that all the $M_\tau$ for all $\tau \in T$ have unique states that do not overlap with either $Q_A$ or states of other $M_\tau$'s.

Formally, $A' = (Q', D_{\mathrm{SWPA}}(\Sigma), \delta', I', F')$ is constructed as follows:

$$\begin{aligned}
Q' &= Q_A \cup \{\mathrm{tobegin}_\tau, \mathrm{backup}_\tau \mid \tau \in T\} \cup \bigcup_{\tau \in T} Q_{M_\tau} \\
I' &= I_A \\
F' &= F_A \\
\delta' &= \{(p, d, q) \in \delta_A \mid d \in \{\leftarrow, \rightarrow\}\} \cup \bigcup \{\delta_\tau \mid \tau \in T\}
\end{aligned}$$

Here the transitions needed to simulate $\tau = (p, \psi(x), q)$ are as follows:

$$\begin{aligned}
\delta_\tau = \; & \{(p, \mathrm{put}, \mathrm{tobegin}_\tau), (\mathrm{tobegin}_\tau, \leftarrow, \mathrm{tobegin}_\tau)\} \cup \\
& \{(\mathrm{tobegin}_\tau, \mathrm{head}, s_{\mathrm{in}}) \mid s_{\mathrm{in}} \in I_{M_\tau}\} \cup \\
& \{s, (\neg\mathrm{tail}, \neg\mathrm{here}, \mathrm{lab}_\sigma, \rightarrow), t) \mid (s, \sigma, t) \in \delta_{M_\tau}\} \cup \\
& \{s, (\neg\mathrm{tail}, \mathrm{here}, \mathrm{lab}_\sigma, \rightarrow), t) \mid (s, (\sigma, 1), t) \in \delta_{M_\tau}\} \cup \\
& \{s, (\mathrm{tail}, \neg\mathrm{here}, \mathrm{lab}_\sigma), \mathrm{backup}_\tau) \mid (s, \sigma, t_{\mathrm{fin}}) \in \delta_{M_\tau} \text{ for some } t_{\mathrm{fin}} \in F_{M_\tau}\} \cup \\
& \{s, (\mathrm{tail}, \mathrm{here}, \mathrm{lab}_\sigma), \mathrm{backup}_\tau) \mid (s, (\sigma, 1), t_{\mathrm{fin}}) \in \delta_{M_\tau} \text{ for some } t_{\mathrm{fin}} \in F_{M_\tau}\} \cup \\
& \{(\mathrm{backup}_\tau, (\neg\mathrm{here}, \leftarrow), \mathrm{backup}_\tau), (\mathrm{backup}_\tau, \mathrm{lift}, q)\}
\end{aligned}$$

## 1.7 Pebble Necessity

The following theorem states that we really need the pebble in Theorem 7. The proof of this theorem is a simpler form of the proof for tree-walking automata (Theorem 15 in [BE97]). It is included here for completeness.

**Theorem 9** *There exist an alphabet $\Sigma$ and a formula $\phi(x,y) \in MSOL_2(\Sigma)$ such that there is no string-walking automaton $A$ with $R(A) = R(\phi)$.*

*Proof.* We construct an MSO formula $\phi(x,y) \in \mathrm{MSOL}_2(\Sigma)$ with $\Sigma = \{b, r\}$, denoting black and red. For a string $w$ over $\Sigma$, the binary relation $R(\phi)$ connects certain positions of $w$. If $w$ has *exactly* one position with symbol $r$, say $v_{\mathrm{red}}$, then $R(\phi)$ will connect any position to $v_{\mathrm{red}}$. Otherwise, i.e., if there is no red symbol or more than one, $R(\phi)$ will connect any position to the next in left-to-right circular order. We use the following abbreviations to define $\phi(x,y)$:

$$\begin{aligned}
\mathrm{ors} &= \exists y\,(\mathrm{lab}_r(y) \wedge \forall z\,(\mathrm{lab}_r(z) \to z = y)) \\
\mathrm{succ}(x,y) &= \mathrm{pre}(x,y) \vee (\mathrm{tail}(x) \wedge \mathrm{head}(y)) \\
\phi(x,y) &= (\mathrm{ors} \wedge \mathrm{lab}_r(y)) \vee (\neg\mathrm{ors} \wedge \mathrm{succ}(x,y))
\end{aligned}$$

Here ors is a closed MSO formula which is true if there is exactly one red symbol in $w$, and $\mathrm{succ}(x,y)$ is an MSO formula which is true if $y$ follows $x$ in left-to-right circular order. Clearly, $\phi(x,y)$ is a binary MSO formula with $R(\phi)$ as indicated above.

Now we will show that there is no string-walking automaton $A = (Q, D_{\mathrm{SWA}}(\Sigma), \delta, I, F)$ with $R(A) = R(\phi)$. Suppose that such an $A$ exists. We may assume that $I = \{q_{\mathrm{in}}\}$. The idea is that when $A$ starts at any position $u$ of a string with only $b$'s, it first has to visit all positionsc of the string to check there is no $r$ in the string. Because there is no way for $A$ to remember its starting point $u$, $A$ also cannot find the position after $u$ in left-to-right circular order.

Let $w$ be $b^{\#Q+1}$, a string of $\#Q+1$ black symbols. Let $w'$ be $rb^{\#Q}$. Thus, $w'$ is $w$ with the first symbol changed to red. We will use the following function:

$$succ(k) = \begin{cases} k+1 & \text{if } k \leq \#Q \\ 1 & \text{otherwise } (k = \#Q+1) \end{cases}$$

which gives the successor to position $k$ in left-to-right circular order. Because $w$ does not contain a symbol $r$, there exists for every $k \in [1, \#Q+1]$ a state $f_k \in F$ such that

$$(q_{\mathrm{in}}, k) \twoheadrightarrow^*_{A,w} (f_k, succ(k)).$$

On the other hand, because $w'$ contains exactly one symbol $r$, for all $k \in [1, \#Q]$, there is no state $f \in F$ such that

$$(q_{\mathrm{in}}, k) \twoheadrightarrow^*_{A,w'} (f, succ(k)).$$

This difference implies that for all $k \in [1, \#Q]$ the walk of $A$ on $w$ from $k$ to $succ(k)$ must visit position 1, because the only different symbol is the first, which is black for $w$ and red for $w'$. Because the walk of $A$ on $w$ from $\#Q+1$ to 1 also visits position 1, there is for all $k \in [1, \#Q+1]$ a $q_k \in Q$ such that

$$(q_{\mathrm{in}}, k) \twoheadrightarrow^*_{A,w} (q_k, 1) \twoheadrightarrow^*_{A,w} (f_k, succ(k)).$$

There are $\#Q+1$ possibilities for $k$ and only $\#Q$ states, which means there are $k, k' \in [1, \#Q+1]$ with $k \neq k'$ and $q_k = q_{k'}$. Then

$$(q_{\text{in}}, k) \twoheadrightarrow^*_{A,w} (q_k, 1) = (q_{k'}, 1) \twoheadrightarrow^*_{A,w} (f_{k'}, succ(k')).$$

This implies that $A$ walks from $k$ both to $succ(k)$ and to $succ(k')$ with $k \neq k'$, a contradiction to the fact that $R(A) = R(\phi)$. □

## 1.8 From String-Walking Pebble Automata to MSO Formulas

In this section we will prove that a string-walking pebble automaton can be simulated by a string-walking automaton with MSO tests. Because it is already known that these automata can be simulated by binary MSO formulas (Proposition 5), and because we have shown in previous sections that a binary MSO formula can be simulated by a string-walking pebble automaton, we obtain the equivalence of binary MSO formulas and string-walking pebble automata.

To prove that it is possible to simulate the use of a pebble with MSO tests, we will need the following lemma. In the lemma, $A$ is a string-walking pebble automaton that is not allowed to move its pebble, but it is allowed to check the presence of the pebble at the current position. The lemma states that the round-trip of $A$ from the position where the pebble is, back to that position can be simulated by a unary MSO formula.

**Lemma 10** *Let $A$ be a string-walking pebble automaton, with no transitions with directives put or lift, over $\Sigma$. Then, for every pair of states $p, q \in Q_A$, there exists a formula $\psi_{pq}(x) \in MSOL_1(\Sigma)$ such that, for all $w \in \Sigma^*$ and $x \in V_w$,*

$$(p, x, x) \twoheadrightarrow^*_{A,w} (q, x, x) \textit{ iff } (w, x) \models \psi_{pq}(x).$$

*Proof.* Let $A$ be a string-walking pebble automaton over $\Sigma$ without put and lift. Let $p, q \in Q_A$. First note that the pebble does not move during $A$'s walk. We can therefore consider the pebble position a constant and mark it using a special symbol $(\sigma, 1)$ where $\sigma$ is the original symbol at the pebble position. We will now define a string-walking automaton $A'$ (without pebble) over $\Sigma \cup (\Sigma \times B_1)$ that simulates the walk of $A$ from the pebble position and back. We define $A' = (Q_A, D_{\text{SWA}}(\Sigma \cup (\Sigma \times B_1)), \delta', \{p\}, \{q\})$. Thus, the states of $A'$ are the same as those of $A$, $A'$'s initial state is $p$, and its final state is $q$. Furthermore, $\delta'$ consists of the following transitions:

$$\begin{aligned}
\delta' \quad = \quad & \delta_A \cap (Q_A \times D_{\text{SWA}}(\Sigma) \times Q_A) \cup \\
& \{(s, \text{lab}_{(\sigma,1)}, t) \mid (s, \text{here}, t) \in \delta_A, \sigma \in \Sigma\} \cup \\
& \{(s, \text{lab}_{\sigma}, t) \mid (s, \neg\text{here}, t) \in \delta_A, \sigma \in \Sigma\} \cup \\
& \{(s, \text{lab}_{(\sigma,1)}, t) \mid (s, \text{lab}_{\sigma}, t) \in \delta_A, \sigma \in \Sigma\}.
\end{aligned}$$

The directive "here" is simulated by checking whether the symbol at the current position is in $\Sigma \times B_1$. Similarly, "$\neg$here" is simulated by checking whether the symbol at the current position is in $\Sigma$. Otherwise, the new symbols from $\Sigma \times B_1$ are treated just like the corresponding symbols from $\Sigma$. Now, for all $s, t \in Q_A$, $w \in \Sigma^*$ and $x, y, z \in V_w$, it is easy to see that

$$(s, x, z) \twoheadrightarrow^*_{A,w} (t, y, z) \text{ iff } (s, x) \twoheadrightarrow^*_{A',w'} (t, y)$$

where $w' = \mathrm{mark}(w, z)$. This means that $A'$ on a string marked at position $z$ walking from $x$ to $y$ simulates $A$ on the unmarked string with the pebble at position $z$. Both automata start in state $s$ and end in state $t$. Also, from the definition of $R_{w'}(A')$,

$$(p, x) \twoheadrightarrow^*_{A',w'} (q, y) \text{ iff } (x, y) \in R_{w'}(A').$$

According to Proposition 5 we can construct a formula $\phi'_{pq}(x, y) \in \mathrm{MSOL}_2(\Sigma \cup (\Sigma \times B_1))$ that simulates the walk of $A'$ on a marked string, so that, for all $w \in \Sigma^*$, $x, y, z \in V_w$ and $w' = \mathrm{mark}(w, z)$,

$$(x, y) \in R_{w'}(A') \text{ iff } (w', x, y) \models \phi'_{pq}(x, y).$$

Using Lemma 2 yields a formula $\phi_{pq}(x, y, z) \in \mathrm{MSOL}_3(\Sigma)$ such that, for all $w \in \Sigma^*$ and $x, y, z \in V_w$,

$$(\mathrm{mark}(w, z), x, y) \models \phi'_{pq}(x, y) \text{ iff } (w, x, y, z) \models \phi_{pq}(x, y, z).$$

We now define $\psi_{pq}(x) = \phi_{pq}(x, x, x) \in \mathrm{MSOL}_1(\Sigma)$. Then, for all $w \in \Sigma^*$ and $u \in V_w$,

$$(p, x, x) \twoheadrightarrow^*_{A,w} (q, x, x) \text{ iff } (w, x) \models \psi_{pq}(x).$$

□

**Theorem 11** *For every string-walking pebble automaton $A$ there exists a string-walking automaton with MSO tests $A'$ such that $R(A) = R(A')$.*

*Proof.* When $A$ drops its pebble at some position $u$ of string $w$, it must return there some later time to pick it up, because of the definition of $R_w(A)$. In the mean time, i.e., until $A$ returns to $u$, $A$ does not move its pebble. By using Lemma 10, the actions of this automaton until its arrival in $u$ can be simulated by a single MSO test. The details are as follows.

Let $A = (Q, D_{\mathrm{SWPA}}(\Sigma), \delta, I, F)$ be a string-walking pebble automaton over $\Sigma$. We will define a string-walking automaton with MSO tests $A'$. The MSO tests $\psi_{pq}$ are obtained by applying Lemma 10 to the automaton derived from $A$ by removing all transitions of the form $(p, d, q)$ with $d \in \{\mathrm{put}, \mathrm{lift}\}$. We define $A' = (Q, \Delta, \delta', I, F)$ with

$$\begin{array}{rcl}
\Delta & = & \{\leftarrow, \rightarrow, \mathrm{true}(x), \mathrm{head}(x), \mathrm{tail}(x), \neg\mathrm{head}(x), \neg\mathrm{tail}(x)\} \cup \\
& & \{\mathrm{lab}_\sigma(x) \mid \sigma \in \Sigma\} \cup \{\psi_{pq}(x) \mid p, q \in Q\} \\
\delta' & = & \{(p, d, q) \in \delta \mid d \in \{\leftarrow, \rightarrow\}\} \cup \\
& & \{(p, \mathrm{head}(x), q) \mid (p, \mathrm{head}, q) \in \delta\} \cup \\
& & \{(p, \neg\mathrm{head}(x), q) \mid (p, \neg\mathrm{head}, q) \in \delta\} \cup \\
& & \{(p, \mathrm{tail}(x), q) \mid (p, \mathrm{tail}, q) \in \delta\} \cup \\
& & \{(p, \neg\mathrm{tail}(x), q) \mid (p, \neg\mathrm{tail}, q) \in \delta\} \cup \\
& & \{(p, \mathrm{lab}_\sigma(x), q) \mid (p, \mathrm{lab}_\sigma, q) \in \delta\} \cup \\
& & \{(p, \mathrm{true}(x), q) \mid (p, \neg\mathrm{here}, q) \in \delta\} \cup \\
& & \{(p, \psi_{qr}(x), s) \mid (p, \mathrm{put}, q) \in \delta \text{ and } (r, \mathrm{lift}, s) \in \delta\}
\end{array}$$

The directives $\leftarrow$, $\rightarrow$, head, $\neg$head, tail, $\neg$tail and $\text{lab}_\sigma$ (for $\sigma \in \Sigma$) are simulated straightforwardly. The MSO test $\psi_{qr}(x)$ simulates the actions of $A$ between the put and the lift. $\square$

This theorem leads us to the grand finale of this chapter.

**Theorem 12** *The following two statements hold:*

- *For every MSO formula $\phi(x,y) \in MSOL_2(\Sigma)$ there exists a string-walking pebble automaton $A$ over $\Sigma$ such that $R(A) = R(\phi)$.*
- *For every string-walking pebble automaton $A$ over $\Sigma$ there exists an MSO formula $\phi(x,y) \in MSOL_2(\Sigma)$ such that $R(\phi) = R(A)$.*

There is another way to prove that any string-walking pebble automaton can be simulated by an MSO formula with two position variables. Let $A$ be a string-walking pebble automaton over $\Sigma$ that recognizes the relation $R(A)$. Then a string-walking pebble automaton $A'$ over $\Sigma \cup (\Sigma \times B_2)$ can easily be constructed such that $(w,u,v) \in R(A)$ just if $\text{mark}(w,u,v) \in L(A')$. $A'$ walks right until it encounters a symbol $(\sigma,1,x)$ with $x \in \{0,1\}$. Then $A'$ starts simulating $A$ directly until it encounters a symbol $(\sigma,y,1)$ with $y \in \{0,1\}$ in a final state. Because SWPA=REG [BH65] and REG=MSO (Proposition 1), there exists an MSO formula $\psi \in \text{MSOL}_0(\Sigma \cup (\Sigma \times B_2))$ such that $L(A') = L(\psi)$. Using Corollary 3, we obtain an MSO formula $\phi(x,y) \in \text{MSOL}_2(\Sigma)$ such that, for all $w \in \Sigma^*$, $\text{mark}(w,u,v) \in L(\psi)$ just if $(w,u,v) \models \phi(x,y)$.

Note that we can also *use* Theorem 12 to derive that SWPA = REG, as follows:

**Theorem 13** *SWPA = REG*

*Proof.* First we will show that any language defined by a string-walking pebble automaton is MSO definable and hence regular. Let $A$ be a string-walking pebble automaton over $\Sigma$. By definition, the language recognized by $A$ is

$$L(A) = \{w \in \Sigma^* \mid \text{ there exists } v \in V_w \text{ such that } (1,v) \in R_w(A)\}.$$

Using the second part of Theorem 12 we obtain a formula $\phi(x,y) \in \text{MSOL}_2(\Sigma)$ such that, for all $w \in \Sigma^*$ and $u,v \in V_w$,

$$(u,v) \in R_w(A) \text{ iff } (w,u,v) \models \phi(x,y).$$

We now define

$$\psi = \forall x(\exists y(\text{head}(x) \rightarrow \phi(x,y)))$$

to obtain that, for all $w \in \Sigma^*$ and $v \in V_w$,

$$\exists v \in V_w((1,v) \in R_w(A)) \text{ iff } \exists v \in V_w((w,1,v) \models \phi(x,y)) \text{ iff } w \models \psi$$

so

$$L(A) = L(\psi)$$

and, since MSO=REG, $L(A)$ is a regular language.

The other way around, we show that any regular language can be described by a string-walking pebble automaton. Let $L$ be a regular language over an alphabet $\Sigma$. Let $M = (Q, \Sigma, \delta, I, F)$ be a finite automaton such that $L = L(M)$. We define a string-walking pebble automaton $A = (Q', D_{\mathrm{SWPA}}(\Sigma), \delta', I, F')$ with

$$
\begin{aligned}
Q' &= Q \cup \{q_{\mathrm{fin}}\} \\
\delta' &= \{(p, (\mathrm{lab}_\sigma, \rightarrow), q) \mid (p, \sigma, q) \in \delta\} \cup \\
&\quad\ \{(p, (\mathrm{lab}_\sigma, \mathrm{tail}), q_{\mathrm{fin}}) \mid (p, \sigma, q) \in \delta \text{ for some } q \in F\} \\
F' &= \{q_{\mathrm{fin}}\}
\end{aligned}
$$

It is obvious to see that $L(A) = L(M) = L$. □

# Chapter 2

# Extensions to String-Walking Pebble Automata

In this chapter we will introduce two extensions to the concept of string-walking pebble automata. These extensions will be constructed in such a way that the class of languages recognized by the automata is the same as the class of languages that the (ordinary) string-walking automata recognize, i.e., the class of regular languages.

## 2.1 String-Walking Multi-Pebble Automata

The first extension to the string-walking automaton is the string-walking multi-pebble automaton. This automaton can use more than one pebble. The number of pebbles available to the automaton is fixed. The pebbles will be taken from a finite set and must be used "nested". This means that, at any point in the automaton's walk, only the pebble that was dropped last can be lifted. It can easily be seen that we can, without changing the automaton's power, number the pebbles $\{1, 2, \ldots, n\}$ and demand that the pebbles must be used in order. In order to see this, suppose the pebbles are taken from a set $P$ with $|P| = n$. Consider the set $M$ of bijections from $P$ to $\{1, 2, \ldots, n\}$ (there are $n!$ such mappings). We can now simulate an automaton with pebble set $P$ by an automaton with pebble set $[1, n]$ by coding the pebble mapping into the states: for every state $q$ we use additional states $\{q_m \mid m \in M\}$. We also modify the transitions with put and lift so that the information on the mapping is maintained. Note that the constraint of the nested use of pebbles is necessary to make sure that the automaton does not recognize non-regular languages. A string-walking 2-pebble automaton without this limitation could recognize the language $L = \{a^n cb^n \mid n \in \mathbb{N}\}$ by putting the pebbles at both ends of the string and moving them towards the middle of the string in turn, checking that both pebbles arrive at the middle $c$ after the same number of moves. This kind of use of the pebbles is prohibited by demanding that the pebbles be used nested. In broader terms, it can be shown that unnested multi-pebble automata are equivalent to two-way multi-head automata, which in turn are equivalent to logarithmic space Turing machines.

Let $n \geq 1$. We use the following set of directives for a string-walking $n$-pebble automaton over $\Sigma$:

$$D_{\mathrm{SWnPA}}(\Sigma) \quad = \quad D_{\mathrm{SWA}}(\Sigma) \cup \{\mathrm{put}_i, \mathrm{lift}_i, \mathrm{here}_i, \neg\mathrm{here}_i \mid 1 \leq i \leq n\}$$

For each $w \in \Sigma^*$ the relations for the new directives are as follows. The relations are on tuples $(u, k, p) \in V_w \times [0, n] \times ([1, n] \to P_w)$. Here $k$ represents that pebbles $1, \ldots, k$ are currently on the string and $p$ maps the pebbles to string positions. We define $p_0 : [1, n] \to P_w$ with $p_0(i) = \perp$ for all $i \in [1, n]$ as the mapping with no pebbles on the string.

The relations hold for any $u \in V_w$, $k \in [0, n]$, and pebble placement function $p : [1, n] \to P_w$.

$$\begin{array}{rcl}
R_w(\mathrm{put}_i) & = & \{((u, i-1, p), (u, i, p(i \mapsto u))) \mid 1 \leq i \leq n\} \\
R_w(\mathrm{lift}_i) & = & \{((u, i, p), (u, i-1, p(i \mapsto \perp))) \mid 1 \leq i \leq n, p(i) = u\} \\
R_w(\mathrm{here}_i) & = & \{((u, k, p), (u, k, p)) \mid 1 \leq i \leq k \text{ and } p(i) = u\} \\
R_w(\neg\mathrm{here}_i) & = & \{((u, k, p), (u, k, p)) \mid 1 \leq i \leq n \text{ and } (i > k \text{ or } p(i) \neq u)\}
\end{array}$$

A *string-walking $n$-pebble automaton over* $\Sigma$ is a finite automaton $A$ over $D_{\mathrm{SWnPA}}(\Sigma)$. For a string-walking $n$-pebble automaton $A = (Q, D_{\mathrm{SWnPA}}(\Sigma), \delta, I, F)$ and a string $w \in \Sigma^*$, a configuration is a tuple $(q, u, k, p) \in Q \times V_w \times [0, n] \times ([1, n] \to P_w)$. The set of all configurations of $A$ and $w$ is names $\mathbb{C}_{A,w}$. The step relation $\twoheadrightarrow_{A,w}$ is defined in the obvious way. For all configurations $(q, u, k, p), (q', u', k', p') \in \mathbb{C}_{A,w}$,

$$(q, u, k, p) \twoheadrightarrow_{A,w} (q', u', k', p') \text{ iff } \exists d \in D_{\mathrm{SWnPA}}(\Sigma) : (q, d, q') \in \delta, ((u, k, p), (u', k', p')) \in R_w(d).$$

For each string $w \in \Sigma^*$, $A$ computes the relation

$$R_w(A) = \{(u, v) \in V_w \times V_w \mid (q, u, 0, p_0) \twoheadrightarrow^*_{A,w} (q', v, 0, p_0) \text{ for some } q \in I \text{ and } q' \in F\}.$$

The position relation $R(A)$ and the language $L(A)$ that $A$ computes are defined in the usual way, as well as the behaviour of transitions of the form $(q, s, q')$ with $d \in D_{\mathrm{SWnPA}}(\Sigma)^*$.

## 2.2 From String-Walking Multi-Pebble Automata to MSO Formulas

In this section we show that any position relation that can be computed by a string-walking multi-pebble automaton can also be defined by a binary MSO formula. Note that the other way around is obvious, because string-walking multi-pebble automata are an extension to string-walking pebble automata.

**Theorem 14** *For every string-walking $n$-pebble automaton $A$ over an alphabet $\Sigma$ there exists a binary MSO formula $\phi(x, y) \in MSOL_2(\Sigma)$ such that $R(\phi) = R(A)$.*

*Proof.* We will prove this theorem using natural induction on the number of pebbles $n$. Note that the case $n = 1$ is equal to the second part of Theorem 12. We present a proof similar to that in Section 1.8. Choose a number of pebbles $n > 1$. In the induction step, it can be assumed that Theorem 14 is valid for any string-walking $(n-1)$-pebble automaton (the induction hypothesis).

The following Claim (similar to Lemma 10) states that after a string-walking $n$-pebble automaton $A$ drops its first pebble, its round-trips from the pebble position back to that same position can be simulated by a unary MSO formula. In the proof of this lemma we use the induction hypothesis.

**Claim** *Let $A = (Q, D_{\mathrm{SWnPA}}(\Sigma), \delta, I, F)$ be a string-walking $n$-pebble automaton, with no transitions with directives $put_1$ or $lift_1$. Then, for every pair of states $p, q \in Q$, there exists a formula $\psi_{pq}(x) \in MSOL_1(\Sigma)$ such that, for all $w \in \Sigma^*$ and $x \in V_w$,*

$$(p, x, 1, p_0(1 \mapsto x)) \twoheadrightarrow^*_{A,w} (q, x, 1, p_0(1 \mapsto x)) \text{ iff } (w, x) \models \psi_{pq}(x).$$

To prove this Claim, we construct a string-walking $(n-1)$-pebble automaton $A'$ over $\Sigma \cup (\Sigma \times B_1)$ that simulates the walk of $A$ from the position of the first pebble and back (similar to the proof of Lemma 10). We define $A' = (Q, D_{\mathrm{SW}(n-1)\mathrm{PA}}(\Sigma \cup (\Sigma \times B_1)), \delta', \{p\}, \{q\})$ where

$$\begin{aligned}
\delta' \;=\;& \delta \cap (Q \times D_{\mathrm{SWA}}(\Sigma) \times Q) \cup \\
& \{(s, \mathrm{lab}_{(\sigma,1)}, t) \mid (s, \mathrm{here}_1, t) \in \delta, \sigma \in \Sigma\} \cup \\
& \{(s, \mathrm{lab}_{\sigma}, t) \mid (s, \neg\mathrm{here}_1, t) \in \delta, \sigma \in \Sigma\} \cup \\
& \{(s, \mathrm{lab}_{(\sigma,1)}, t) \mid (s, \mathrm{lab}_{\sigma}, t) \in \delta, \sigma \in \Sigma\} \cup \\
& \{(s, \mathrm{put}_{i-1}, t) \mid (s, \mathrm{put}_i, t) \in \delta, 2 \le i \le n\} \cup \\
& \{(s, \mathrm{lift}_{i-1}, t) \mid (s, \mathrm{lift}_i, t) \in \delta, 2 \le i \le n\} \cup \\
& \{(s, \mathrm{here}_{i-1}, t) \mid (s, \mathrm{here}_i, t) \in \delta, 2 \le i \le n\} \cup \\
& \{(s, \neg\mathrm{here}_{i-1}, t) \mid (s, \neg\mathrm{here}_i, t) \in \delta, 2 \le i \le n\}
\end{aligned}$$

The states of $A'$ are the same as those of $A$, $A'$'s only initial state is $p$ and $A'$'s only final state is $q$. All directives that do not use or test the pebble are simulated directly. Pebble 1 is simulated by the marking, while pebbles $2, \ldots, n$ are simulated by pebbles $1, \ldots, n-1$. Now it is easy to see that, for all $s, t \in Q$, $w \in \Sigma^*$, $x, y, z \in V_w$ and $w' = \mathrm{mark}(w, z)$,

$$(s, x, 1, p_0(1 \mapsto z)) \twoheadrightarrow^*_{A,w} (t, y, 1, p_0(1 \mapsto z)) \text{ iff } (s, x, 0, p_0) \twoheadrightarrow^*_{A',w'} (t, y, 0, p_0).$$

This means that $A'$ on a string marked at position $z$, starting in a configuration without pebbles and walking from position $x$ to $y$ simulates $A$ on the unmarked string, in a configuration with pebble 1 at position $z$. Note that, following the definition of $R_{w'}(A')$ for a string-walking $(n-1)$-pebble automaton,

$$(p, x, 0, p_0) \twoheadrightarrow^*_{A',w'} (q, y, 0, p_0) \text{ iff } (x, y) \in R_{w'}(A').$$

According to the induction hypothesis, there exists a binary MSO formula

$$\phi'_{pq}(x, y) \in \mathrm{MSOL}_2(\Sigma \cup (\Sigma \times B_1))$$

that simulates the walk of $A'$ on a marked string, such that, for all $w \in \Sigma^*$, $x, y, z \in V_w$ and $w' = \mathrm{mark}(w, z)$,

$$(x, y) \in R_{w'}(A') \text{ iff } (w', x, y) \models \phi'_{pq}(x, y).$$

The remainder of the proof of the Claim is again similar to the proof of Lemma 10. Again using Lemma 2 to pull the marking into the free position variables of the MSO formula, we obtain a formula $\phi_{pq}(x, y, z) \in \mathrm{MSOL}_3(\Sigma)$ such that, for all $w \in \Sigma^*$, $x, y, z \in V_w$ and $w' = \mathrm{mark}(w, z)$,

$$(w', x, y) \models \phi'_{pq}(x, y) \text{ iff } (w, x, y, z) \models \phi_{pq}(x, y, z).$$

We now define the abbreviation $\psi_{pq}(x) = \phi_{pq}(x, x, x) \in \mathrm{MSOL}_1(\Sigma)$. Then, for all $w \in \Sigma^*$ and $x \in V_w$,

$$(p, x, 1, p_0(1 \mapsto x)) \twoheadrightarrow^*_{A,w} (q, x, 1, p_0(1 \mapsto x)) \text{ iff } (w, x) \models \psi_{pq}(x).$$

This ends the proof of the Claim. We now continue with the proof of Theorem 14. Let $A = (Q, D_{\mathrm{SWnPA}}(\Sigma), \delta, I, F)$ be a string-walking $n$-pebble automaton over $\Sigma$. We will, like in the proof of Theorem 11, define a string-walking automaton with MSO tests $A'$ over $\Sigma$ such that $R(A) = R(A')$. The MSO tests $\psi_{pq}(x)$ in this automaton are obtained by applying the Claim to the automaton derived from $A$ by removing all transitions of the form $(p, d, q)$ with $d \in \{\mathrm{put}_1, \mathrm{lift}_1\}$. We define $A' = (Q, \Delta, \delta', I, F)$ exactly the same as in the proof of Theorem 11, but with $\neg\mathrm{here}_1$, $\mathrm{put}_1$ and $\mathrm{lift}_1$ instead of $\neg$here, put and lift:

$$\begin{array}{rcl}
\Delta & = & \{\leftarrow, \rightarrow, \mathrm{true}(x), \mathrm{head}(x), \mathrm{tail}(x), \neg\mathrm{head}(x), \neg\mathrm{tail}(x)\} \cup \\
 & & \{\mathrm{lab}_\sigma(x) \mid \sigma \in \Sigma\} \cup \{\psi_{pq}(x) \mid p, q \in Q\} \\
\delta' & = & \{(p, d, q) \in \delta \mid d \in \{\leftarrow, \rightarrow\}\} \cup \\
 & & \{(p, \mathrm{head}(x), q) \mid (p, \mathrm{head}, q) \in \delta\} \cup \\
 & & \{(p, \neg\mathrm{head}(x), q) \mid (p, \neg\mathrm{head}, q) \in \delta\} \cup \\
 & & \{(p, \mathrm{tail}(x), q) \mid (p, \mathrm{tail}, q) \in \delta\} \cup \\
 & & \{(p, \neg\mathrm{tail}(x), q) \mid (p, \neg\mathrm{tail}, q) \in \delta\} \cup \\
 & & \{(p, \mathrm{lab}_\sigma(x), q) \mid (p, \mathrm{lab}_\sigma, q) \in \delta\} \cup \\
 & & \{(p, \mathrm{true}(x), q) \mid (p, \neg\mathrm{here}_1, q) \in \delta\} \cup \\
 & & \{(p, \psi_{qr}(x), s) \mid (p, \mathrm{put}_1, q) \in \delta \text{ and } (r, \mathrm{lift}_1, s) \in \delta\}
\end{array}$$

Again, the directives $\{\leftarrow, \rightarrow, \mathrm{head}, \neg\mathrm{head}, \mathrm{tail}, \neg\mathrm{tail}\}$ and $\mathrm{lab}_\sigma$ (for $\sigma \in \Sigma$) are simulated straightforwardly. The directive $\psi_{qr}(x)$ simulates the actions of $A$ from $\mathrm{put}_1$ to the following $\mathrm{lift}_1$ (at the same location), including all $\mathrm{put}_i$ and $\mathrm{lift}_i$ for $2 \leq i \leq n$. It is easy to see that $R(A) = R(A')$. Now, using Proposition 5, there exists a binary MSO formula $\phi(x, y) \in \mathrm{MSOL}_2(\Sigma)$ such that $R(\phi) = R(A') = R(A)$. □

## 2.3 String-Walking Marble Automata

Another way to extend string-walking pebble automata is the introduction of marbles instead of pebbles. They come in several *colours*, that are labeled with a colour alphabet $\Gamma$. For each colour $\gamma \in \Gamma$, the number of marbles of colour $\gamma$ is not limited. In this paper, we give only the definition of string-walking marble automata and of their behaviour. The idea is that every marble colour can be used for a quantifier of MSO formulas; however, it is not proven here that string-walking marble automata compute exactly the position relations that binary MSO formulas recognize. This may not even be the case.

$$D_{\mathrm{SWMA}}(\Sigma, \Gamma) = D_{\mathrm{SWA}}(\Sigma) \cup \{\mathrm{put}_\gamma, \mathrm{lift}_\gamma, \mathrm{here}_\gamma, \neg\mathrm{here}_\gamma \mid \gamma \in \Gamma\}$$

For all directives $d$, $R_w(d)$ is a binary relation over $V_w \times (\Gamma \to 2^{V_w}) \times \mathbb{N} \times (\mathbb{N} \to ((V_w \times \Gamma) \cup \{\perp\}))$. An element $(u, p, n, s)$ of this set means that

- The current position of the automaton is $u$.
- For every marble colour $\gamma \in \Gamma$, $p(\gamma)$ is the set of positions where a marble of colour $\gamma$ is lying.
- $n$ is the total number of marbles of any colour on the string.
- For every natural number $1 \leq i \leq n$, if $s(i) = (v, \gamma) \in V_w \times \Gamma$, then the $i$th marble that was laid down is at position $v$ and of colour $\gamma$. For $i > n$, $s(i) = \perp$. Effectively, $s$ is a stack of marble colours and positions. This information is needed to enforce the nested use of the marbles.

Note that $p$ can be determined from $n$ and $s$: for every $\gamma \in \Gamma$,

$$p(\gamma) = \{u \in V_w \mid \exists i \in [1, n] : s(i) = (u, \gamma)\}.$$

The relations for the directives from $D_{\mathrm{SWA}}(\Sigma)$ are extended straightforwardly. These directives do not alter $p$, $n$ or $s$. For the directives $\mathrm{put}_\gamma$, $\mathrm{lift}_\gamma$, $\mathrm{here}_\gamma$ and $\neg\mathrm{here}_\gamma$, the relations are as follows. The definition of $R_w(\mathrm{put}_\gamma)$ is complicated, because we demand that the marbles are nested as usual and that all marbles of colour $\gamma$ must be put down after each other, without putting down a marble of another colour in between.

$$\begin{aligned}
R_w(\mathrm{put}_\gamma) &= \{((u,p,n,s),(u,p(\gamma \mapsto p(\gamma) \cup \{u\}), n+1, s(n+1 \mapsto (u,\gamma)))) \mid u \notin p(\gamma), \\
&\qquad \forall 1 \leq i < j \leq n (\neg \exists v, v' \in V_w, \gamma' \in \Gamma : (\gamma' \neq \gamma \wedge s(i) = (v,\gamma) \wedge s(j) = (v',\gamma')))\} \\
R_w(\mathrm{lift}_\gamma) &= \{((u,p,n,s),(u,p(\gamma \mapsto p(\gamma) \setminus \{u\}), n-1, s(n \mapsto \perp))) \mid s(n) = (u,\gamma)\} \\
R_w(\mathrm{here}_\gamma) &= \{((u,p,n,s),(u,p,n,s)) \mid u \in p(\gamma)\} \\
R_w(\neg\mathrm{here}_\gamma) &= \{((u,p,n,s),(u,p,n,s)) \mid u \notin p(\gamma)\}
\end{aligned}$$

We define $p_0 : \Gamma \to 2^{V_w}$ with $p_0(\gamma) = \emptyset$ for all $\gamma \in \Gamma$. We define $s_0 : \mathbb{N} \to ((V_w \times \Gamma) \cup \{\perp\})$ with $s(i) = \perp$ for all $i \in \mathbb{N}$.

The relation computed by $A$ on a string $w \in \Sigma^*$ is

$$R_w(A) = \{(u, v) \in V_w \times V_w \mid (q, u, p_0, 0, s_0) \twoheadrightarrow^*_{A,w} (q', v, p_0, 0, s_0) \text{ for some } q \in I \text{ and } q' \in F\}.$$

# Chapter 3

# Binary MSO Formulas and Tree-Walking Automata

## 3.1 Preliminaries

In this section we recall the well-known concepts of trees, finite tree automata, regular tree languages, and monadic second order logic.

### Trees

In the usual way, trees are defined as finite, directed graphs with labeled nodes and edges. Let $\Sigma$ and $\Gamma$ be alphabets of node labels and edge labels, respectively. A *graph* over $(\Sigma, \Gamma)$ is a triple $(V, E, \mathrm{lab})$, with $V$ a finite set of nodes, $E \subseteq V \times \Gamma \times V$ the set of labeled edges, and $\mathrm{lab} : V \to \Sigma$ the node-labeling function. For a given graph $g$, its nodes, edges and node-labeling functions are denoted $V_g$, $E_g$ and $\mathrm{lab}_g$, respectively.

The trees we consider have their nodes labeled by symbols from a ranked alphabet. The edge from a parent to its $i$-th child is labeled with the number $i$. A *ranked alphabet* $\Sigma$ is an alphabet $\Sigma$ together with a *rank function* $\mathrm{rk} : \Sigma \to \mathbb{N}$. For any symbol $\sigma \in \Sigma$, $\mathrm{rk}(\sigma) = k$ denotes that the *rank* of $\sigma$ is $k$. In terms of trees this means that a node labeled with $\sigma$ will have $k$ children. For all $k \in \mathbb{N}$, $\Sigma_k = \{\sigma \in \Sigma \mid \mathrm{rk}(\sigma) = k\}$ is the set of symbols of rank $k$. The *rank interval* of the ranked alphabet $\Sigma$ is $[1, m]$, where $m$ is the maximal rank of the elements of $\Sigma$; it is denoted $\mathrm{rki}(\Sigma)$.

A *tree* over $\Sigma$ is an acyclic connected graph $g$ over $(\Sigma, \mathrm{rki}(\Sigma))$ such that

- No node of $g$ has more than one incoming edge.
- For every node $u$ of $g$, $u$ has only outgoing edges with labels in $[1, \mathrm{rk}(\mathrm{lab}_g(u))]$.
- For every node $u$ of $g$ and every $i \in [1, \mathrm{rk}(\mathrm{lab}_g(u))]$, $u$ has exactly one outgoing edge with label $i$.

The set of all trees over $\Sigma$ is denoted $T_\Sigma$. A subset of $T_\Sigma$ is also called a *tree language*.

The root of a tree $t$ (the node with no incoming edges) is denoted $\mathrm{root}_t$. For nodes $u, v$ of $t$, if $(u, i, v) \in E_t$, then $u$ is called the parent of $v$, and $v$ is called the $i$-th child of $u$, denoted by $u \cdot i$. We also define $u \cdot 0 = u$. We define the set of ancestors $\mathrm{anc}(u)$ of a node $u$ as $u$ itself and the ancestors of its parent. The *least common ancestor* of two nodes $u$ and $v$ (denoted $\mathrm{lca}(u, v)$) is defined as the node $w$ such that

- $w$ is an ancestor of both $u$ and $v$.
- If $w'$ is an ancestor of both $u$ and $v$, then $w'$ is an ancestor of $w$.

If $u$ is an ancestor of $v$, then $v$ is a *descendant* of $u$. The set $\mathrm{des}(u)$ denotes the set of all descendants of $u$. For a node $u$ of $t$, $t_u$ denotes the subtree of $t$ with root $u$, i.e., the subgraph of $t$ induced by the set of all descendants of $u$.

Let $\Sigma$ be a ranked alphabet and let $k \in \mathbb{N}$. We define $k$-ary node relations in analogy to $k$-ary position relations for strings. A $k$-ary *node relation* over $\Sigma$ is a subset of $\{(t, u_1, \ldots, u_k) \mid t \in T_\Sigma \text{ and } u_i \in V_t \text{ for all } i \in [1, k]\}$.

We define *marked trees* as follows, in analogy to marked strings. Let $\Sigma$ be a ranked alphabet and let $k \geq 1$. The ranked alphabet $\Sigma \cup (\Sigma \times B_k)$ is defined as usual, where we assign to the symbol $(\sigma, b_1, \ldots, b_k)$ the rank $\mathrm{rk}(\sigma)$ (for $\sigma \in \Sigma$ and $b_i \in [1, k]$ for $i \in [1, k]$). Let $t \in T_\Sigma$ and let $u_1, \ldots, u_k \in V_t$. We define $\mathrm{mark}(t, u_1, \ldots, u_k)$ as a tree over $\Sigma \cup (\Sigma \times B_k)$, with node-set $V_t$, edges $E_t$ and node-labeling function $\mathrm{lab}' : V_t \to \Sigma \cup (\Sigma \times B_k)$ with $\mathrm{lab}'(u) = \mathrm{lab}_t(u)$ if $u \neq u_i$ for all $i \in [1, k]$, and $\mathrm{lab}'(u) = (\mathrm{lab}_t(u), (u = u_1), \ldots, (u = u_k))$ otherwise (where $(u = u_i) = 1$ iff $u$ equals $u_i$).

## Finite Tree Automata

A finite tree automaton [GS84] traverses a tree bottom-up. It starts in all the leaves of the tree, in parallel, and works its way up to the root. A finite tree automaton has a finite set of states $Q$ and a transition function. The state of the automaton in a node of the tree is determined by the states of the children of the node and by the label of the node. For a symbol $\sigma$ with $\mathrm{rk}(\sigma) = k$, the transition function $\delta_\sigma$ is therefore a function from $Q^k$ to $Q$. A finite tree automaton accepts a tree if it ends at the root of the tree in one of a set of pre-determined final states.

Let $\Sigma$ be a ranked alphabet. A (deterministic) *finite tree automaton* over $\Sigma$ is a 4-tuple $M = (Q, \Sigma, \delta, F)$ where $Q$ is a finite set of states, $\Sigma$ is the input alphabet, $\delta = \{\delta_\sigma \mid \sigma \in \Sigma\}$ with, for $\sigma \in \Sigma_k$, $\delta_\sigma : Q^k \to Q$ the transition function for $\sigma$, and $F \subseteq Q$ the set of final states. For every tree $t \in T_\Sigma$ and node $u \in V_t$, the state in which $M$ reaches $u$, denoted $\mathrm{state}_{M,t}(u)$, is defined by bottom-up induction as $\mathrm{state}_{M,t}(u) = \delta_\sigma(\mathrm{state}_{M,t}(u \cdot 1), \cdots, \mathrm{state}_{M,t}(u \cdot k))$ where $\mathrm{lab}_t(u) = \sigma$ with $\mathrm{rk}(\sigma) = k$. The tree language recognized by $M$ is $L(M) = \{t \in T_\Sigma \mid \mathrm{state}_{M,t}(\mathrm{root}(t)) \in F\}$. A tree language $L$ is called a *regular tree language* if it is recognized by a finite tree automaton. The class of all regular tree languages is named REGT.

For any finite tree automaton $M = (Q, \Sigma, \delta, F)$, tree $t \in T_\Sigma$ and node

$u \in V_t$, the set of successful states of $M$ at $u$, denoted by $\mathrm{succ}_{M,t}(u)$, is defined (by top-down induction) as follows: $\mathrm{succ}_{M,t}(\mathrm{root}_t) = F$ and, for a node $u \in V_t$ with $\mathrm{lab}_t(u) = \sigma$, $\mathrm{rk}(\sigma) = k > 0$, $\mathrm{succ}_{M,t}(u \cdot i)$ is the set of all states $q \in Q$ such that $\delta_\sigma(q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_k) \in \mathrm{succ}_{M,t}(u)$, where $q_j = \mathrm{state}_{M,t}(u \cdot j)$ for $j \in [1, i-1] \cup [i+1, k]$. Intuitively, $q \in \mathrm{succ}_{M,t}(u)$ means that $M$ will reach the root of $t$ in a final state assuming that it reaches $u$ in state $q$ (instead of $\mathrm{state}_{M,t}(u)$). Thus, $t \in L(M)$ iff $\mathrm{state}_{M,t}(u) \in \mathrm{succ}_{M,t}(u)$. This can easily be proven ([BE97], Lemma 1).

### Monadic Second Order Logic (on trees)

The monadic second order logic on strings described in Section 1.1 can be extended to trees (see, e.g., [TW68, Don70, BE97]). We will in short describe the differences. Let $\Sigma$ be a ranked alphabet.

- The variables $x, y, \ldots$ are now node variables; the variables $X, Y, \ldots$ are now node-set variables. For a given tree $t \in T_\Sigma$, node variables range over $V_t$ and node-set variables range over subsets of $V_t$.
- The atomic formulas are $\mathrm{lab}_\sigma(x)$, for every $\sigma \in \Sigma$, denoting that $x$ has label $\sigma$; $\mathrm{edg}_i(x, y)$, for every $i \in \mathrm{rki}(\Sigma)$, denoting that $y$ is the $i$th child of $x$; and $x \in X$, denoting that $x$ is an element of $X$.
- The language defined by a formula $\phi \in \mathrm{MSOL}_0(\Sigma)$ is now a tree language, $L(\phi) = \{t \in T_\Sigma \mid t \models \phi\}$

The class of all MSO definable tree languages is named MSOT.

The following classical result from [Don70, TW68] states that MSOT=REGT, i.e., that formulas from monadic second order logic with no free variables define exactly the regular tree languages:

**Proposition 15** *A tree language is MSO definable if and only if it is regular.*

As stated in Section 1.1, the results of Lemma 2, Corollary 3, and Lemma 4 are also valid for trees (with $\Sigma$ a ranked alphabet and $t \in T_\Sigma$ instead of $w \in \Sigma^*$).

## 3.2 Tree-Walking Automata

A tree-walking automaton (see, e.g., [KS81]) is very similar to a string-walking automaton (Section 1.2). A tree-walking automaton can walk freely on the tree. In this paper we will give a definition of tree-walking automaton that is slightly different from the definition in [KS81]. First of all, we will make use of directives to define the behaviour of the automaton, while [KS81] directly defines a binary step relation on the set of configurations. Secondly, with our definition the automaton is able to check which child of its parent the current node is. In [KS81], the automaton can only move up, without any restriction to child number. This makes our automaton more powerful (i.e., it can recognize more tree languages).

Let $\Sigma$ be a ranked alphabet. Like a string-walking automaton, a tree-walking automaton over $\Sigma$ is a finite automaton with a special set of directives as input alphabet. For a tree-walking automaton, we define the set of directives as

$$D_{\mathrm{TWA}}(\Sigma) = \{\uparrow_i, \downarrow_i \mid i \in \mathrm{rki}(\Sigma)\} \cup \{\mathrm{root}, \neg\mathrm{root}\} \cup \{\mathrm{lab}_\sigma \mid \sigma \in \Sigma\}.$$

The meaning of the directives is as follows:

- $\uparrow_i$ means go up via an edge labeled by $i$, i.e., move to the parent, provided the current node is the $i$th child of its parent. This directive can be used to check which child of its parent the current node is.
- $\downarrow_i$ means go down via an edge labeled by $i$, i.e., move to the $i$th child of the current node.
- root checks whether the current node is the root of the tree.
- $\neg$root is the negation of root.
- $\mathrm{lab}_\sigma$ checks whether the current node is labeled by $\sigma$.

Note that the directive root corresponds to the directive head of the string-walking automaton. A directive "leaf" (corresponding to tail) is not necessary since the automaton can test whether the label of the current node has rank 0. Also, $\neg$root can be simulated by $\uparrow_i\downarrow_i$ for all $i \in \mathrm{rki}(\Sigma)$. To formalize the behaviour of these directives we define for each tree $t \in T_\Sigma$ and each directive $d \in D_{\mathrm{TWA}}(\Sigma)$ the following binary relation $R_t(d)$ on $V_t$:

$$\begin{aligned}
R_t(\uparrow_i) &= \{(u,v) \mid (v,i,u) \in E_t\} \\
R_t(\downarrow_i) &= \{(u,v) \mid (u,i,v) \in E_t\} \\
R_t(\mathrm{root}) &= \{(u,u) \mid u = \mathrm{root}_t\} \\
R_t(\neg\mathrm{root}) &= \{(u,u) \mid u \neq \mathrm{root}_t\} \\
R_t(\mathrm{lab}_\sigma) &= \{(u,u) \mid \mathrm{lab}_t(u) = \sigma\}
\end{aligned}$$

Like the definition of string-walking automata, a *tree-walking automaton* over $\Sigma$ is a finite automaton $A$ over $D_{\mathrm{TWA}}(\Sigma)$. For a tree-walking automaton $A = (Q, D_{\mathrm{TWA}}(\Sigma), \delta, I, F)$ and a tree $t \in T_\Sigma$, an element $(q,u)$ of $Q \times V_t$ is a configuration of the automaton. The set of all configurations of $A$ and $t$ is denoted by $\mathbb{C}_{A,t}$. A configuration $(q,u)$ denotes that the automaton $A$ is in state $q$ at node $u$.

Let $A = (Q, D_{\mathrm{TWA}}(\Sigma), \delta, I, F)$ be a tree-walking automaton over $\Sigma$ and let $t \in T_\Sigma$. Like with string-walking automata, we define a step-relation $\twoheadrightarrow_{A,t}$ on the set of configurations as follows. For every $(q,u),(q',u') \in \mathbb{C}_{A,t}$,

$$(q,u) \twoheadrightarrow_{A,t} (q',u') \text{ iff } \exists d \in D_{\mathrm{TWA}}(\Sigma) : (q,d,q') \in \delta \text{ and } (u,u') \in R_t(d).$$

The automaton $A$ computes on each tree $t \in T_\Sigma$ the binary relation

$$R_t(A) = \left\{(u,u') \in V_t \times V_t \mid (q_{\mathrm{in}},u) \twoheadrightarrow^*_{A,t} (q_{\mathrm{fin}},u') \text{ for some } q_{\mathrm{in}} \in I \text{ and } q_{\mathrm{fin}} \in F\right\}.$$

The node relation that $A$ computes is

$$R(A) = \{(t,u,v) \mid t \in T_\Sigma \text{ and } (u,v) \in R_t(A)\}.$$

The definitions of these relations ($\twoheadrightarrow_{A,t}$, $R_t(A)$ and $R(A)$) are mutatis mutandis equal to the definitions for string-walking automaton in Section 1.2.

The tree language that $A$ recognizes is defined as

$$L(A) = \{t \in T_\Sigma \mid (\text{root}_t, v) \in R_t(A) \text{ for some } v \in V_t\}.$$

Note that we can assume that the walk of the automaton ends as well as begins in $\text{root}_t$, since we can easily add extra transitions to make the automaton walk to the root at the end of its walk. The class of all tree languages recognized by any tree-walking automaton is named TWA. It is an open problem if TWA=REGT.

Like in the case of string-walking automata, we will also allow transitions of the form $(p, s, q)$ with $s \in D_{\text{TWA}}(\Sigma)^*$. If $s = d_1 \cdots d_n$, we define, for all $t \in T_\Sigma$,

$$R_t(s) = R_t(d_1) \circ \cdots \circ R_t(d_n).$$

As with string-walking automata, these extended directives can easily be implemented by adding extra transitions and states.

Determinism of tree-walking automata in defined in analogy with determinism of string-walking automata. A tree-walking automaton $A = (Q, D_{\text{TWA}}(\Sigma), \delta, I, F)$ is deterministic if the following conditions hold:

1. $\#I = 1$.
2. If $(p, d, q) \in \delta$, then $p \notin F$.
3. For all distinct transitions $(p, d_1, q_1), (p, d_2, q_2) \in \delta$, $d_1$ and $d_2$ are two mutually exclusive directives in $D_{\text{TWA}}(\Sigma)$.

Two directives $d_1, d_2 \in D_{\text{TWA}}(\Sigma)$ are again mutually exclusive if, for all $t \in T_\Sigma$, $\text{dom}(R_t(d_1)) \cap \text{dom}(R_t(d_2)) = \emptyset$. The following pairs of directives in $D_{\text{TWA}}(\Sigma)$ are mutually exclusive:

- $\{\uparrow_i, \uparrow_j\}$ with $i \neq j$.
- $\{\uparrow_i, \text{root}\}$ for $i \in \text{rki}(\Sigma)$
- $\{\text{root}, \neg\text{root}\}$
- $\{\text{lab}_{\sigma_1}, \text{lab}_{\sigma_2}\}$ with $\sigma_1 \neq \sigma_2$
- $\{\downarrow_i, \text{lab}_\sigma\}$ if $i > \text{rk}(\sigma)$.

## 3.3 Push-Down Tree-Walking Automata

A push-down tree-walking automaton walks on a tree, starting in the root of the tree. It has a push-down store. At each step of the automaton's walk, its possible moves are determined by the automaton's state, by the symbol at the current position of the automaton and by the push-down symbol at the top of the store. If the automaton moves up in the tree, the top symbol is removed from the push-down store; if the automaton moves down, a new symbol is added to the push-down store. There are two main differences between the push-down tree-walking automata as described in this section and those in literature (e.g., [KS81]). Here, directives are used to describe the different possible actions of the automaton. Furthermore, [KS81] lets the automaton accept a tree if it walks up from the root, leaving the tree in a final state; in the description here, the walk can end in any node of the tree in a final state. These differences have no consequences for the power of the automaton. Note that the tree-walking automaton in [KS81] is the push-down tree-walking automaton with only one push-down symbol.

We will present the push-down tree-walking automaton as an extension of the tree-walking automaton. Let $\Sigma$ be a ranked alphabet and let $\Gamma$ be an alphabet. The directives of a push-down tree-walking automaton are

$$\begin{aligned} D_{\mathrm{PDTWA}}(\Sigma,\Gamma) &= \{\uparrow\} \cup \{\downarrow_{i,\gamma} \mid i \in \mathrm{rki}(\Sigma), \gamma \in \Gamma\} \cup \\ &\quad \{\mathrm{lab}_\sigma \mid \sigma \in \Sigma\} \cup \left\{\mathrm{stay}_{\gamma_1,\gamma_2} \mid \gamma_1,\gamma_2 \in \Gamma\right\}. \end{aligned}$$

The meaning of these directives is as follows:

- $\uparrow$ means go up to the parent of the current node and remove the topmost symbol from the push-down store. Note that the subscript $i$ is not necessary for this automaton. As the automaton traverses the tree, routing information can be placed in the push-down store.
- $\downarrow_{i,\gamma}$ means go down via an edge labeled by $i$ and add $\gamma$ to the push-down store.
- $\mathrm{lab}_\sigma$ checks whether the current node is labeled by $\sigma$.
- $\mathrm{stay}_{\gamma,\gamma'}$ checks whether the topmost symbol from the push-down store is $\gamma$ and, if so, replaces it by $\gamma'$.

The push-down tree-walking automaton does not need the directives root and $\neg$root. By using marked push-down symbols on the bottom of the push-down store, the automaton can check whether the current node is the root of the tree by inspecting the top symbol of the push-down store. If convenient, however, we will use the directives root and $\neg$root since a push-down tree-walking automaton with these directives can be simulated by a push-down tree-walking automaton without root and $\neg$root.

To formalize the directives, we define for each tree $t \in T_\Sigma$ and each directive $d \in D_{\mathrm{PDTWA}}(\Sigma,\Gamma)$ the following binary relation $R_t(d)$ on pairs $(u,\beta)$ where $u \in V_t$ is the current node, and $\beta = \gamma_1 \cdots \gamma_n \in \Gamma^*$ is the current contents of the push-down store with $n$ equal to the number of ancestors of $u$.

$$\begin{aligned} R_t(\uparrow) &= \{((u,\gamma_1\cdots\gamma_n),(v,\gamma_1\cdots\gamma_{n-1})) \mid v \text{ is the parent of } u\} \\ R_t(\downarrow_{i,\gamma}) &= \{((u,\gamma_1\cdots\gamma_n),(v,\gamma_1\cdots\gamma_n\gamma)) \mid v \text{ is the } i\text{th child of } u\} \\ R_t(\mathrm{lab}_\sigma) &= \{((u,\gamma_1\cdots\gamma_n),(u,\gamma_1\cdots\gamma_n)) \mid \mathrm{lab}_t(u) = \sigma\} \\ R_t(\mathrm{stay}_{\gamma,\gamma'}) &= \{((u,\gamma_1\cdots\gamma_{n-1}\gamma_n),(u,\gamma_1\cdots\gamma_{n-1}\gamma')) \mid \gamma_n = \gamma\} \end{aligned}$$

Let $\Sigma$ be a ranked alphabet (of node labels) and let $\Gamma$ be an alphabet (of push-down symbols) with a designated element $\gamma_{\mathrm{in}}$ (the initial push-down symbol). A *push-down tree-walking automaton* over $(\Sigma,\Gamma)$ is a finite automaton $A$ over $D_{\mathrm{PDTWA}}(\Sigma,\Gamma)$. For a push-down tree-walking automaton $A = (Q, D_{\mathrm{PDTWA}}(\Sigma,\Gamma), \delta, I, F)$ over $(\Sigma,\Gamma)$ and a tree $t \in T_\Sigma$, the configurations of $A$ are triples $(q,u,\beta)$, with $q \in Q$ the state of the automaton,

$u \in V_t$ the current node, and $\beta = \gamma_1 \cdots \gamma_n \in \Gamma^*$ the contents of the push-down store, where $n$ is equal to the number of ancestors of $u$. The set of all configurations on $A$ and $t$ is denoted by $\mathbb{C}_{A,t}$.

Let $A = (Q, D_{\text{PDTWA}}(\Sigma,\Gamma), \delta, I, F)$ be a push-down tree-walking automaton over $(\Sigma,\Gamma)$ and let $t \in T_\Sigma$. The definition of the step-relation $\twoheadrightarrow_{A,t}$ is almost identical to the definition of the step-relation for tree-walking automata. For every $(q,u,\beta),(q',u',\beta') \in \mathbb{C}_{A,t}$,

$$(q,u,\beta) \twoheadrightarrow_{A,t} (q',u',\beta') \text{ iff } \exists d \in D_{\text{PDTWA}}(\Sigma,\Gamma) : (q,d,q') \in \delta \text{ and } ((u,\beta),(u',\beta')) \in R_t(d).$$

The tree language that $A$ recognizes is defined as

$$L(A) = \left\{ t \in T_\Sigma \mid (q_{\text{in}}, \text{root}_t, \gamma_{\text{in}}) \twoheadrightarrow^*_{A,t} (q_{\text{fin}}, u, \beta) \text{ for some } q_{\text{in}} \in I, q_{\text{fin}} \in F, u \in V_t \text{ and } \beta \in \Gamma^* \right\}.$$

The automaton starts in the root of a tree with only the initial push-down symbol $\gamma_{\text{in}}$ on the push-down store. It recognizes the tree if it reaches a final state in any node and with any contents of the push-down store. The class of all tree languages recognized by any push-down tree-walking automaton is named PDTWA. Note that because of the push-down store there is no reasonable definition for a binary node relation for push-down tree-walking automata.

Transitions of the form $(p,s,q)$ with $s \in D_{\text{TWA}}(\Sigma)^*$ are treated in the same way as with tree-walking automata. Determinism of push-down tree-walking automata is defined in the same way as for tree-walking automata. The following pairs of directives in $D_{\text{PDTWA}}(\Sigma,\Gamma)$ are mutually exclusive:

- $\{\text{lab}_{\sigma_1}, \text{lab}_{\sigma_2}\}$ with $\sigma_1 \neq \sigma_2$
- $\{\text{stay}_{\gamma_1,\gamma_1'}, \text{stay}_{\gamma_2,\gamma_2'}\}$ with $\gamma_1 \neq \gamma_2$.

## 3.4 Equivalence of Finite Tree Automata and Push-Down Tree-Walking Automata

In this section we show the known result [KS81] that push-down tree-walking automata recognize exactly the regular tree languages. First we show that for every finite tree automaton there exists a push-down tree-walking automaton that recognizes the same language; then we show that, the other way around, for every push-down tree-walking automaton there exists a finite tree automaton that recognizes the same language.

**Lemma 16** *For every finite tree automaton $M$ there exists a push-down tree-walking automaton $A$ such that $L(A) = L(M)$.*

*Proof.* Let $M = (Q, \Sigma, \delta, F)$ be a finite tree automaton. We construct a push-down tree-walking automaton $A = (Q', D_{\text{PDTWA}}(\Sigma,\Gamma), \delta', \{q_{\text{in}}\}, \{q_{\text{fin}}\})$ over $(\Sigma,\Gamma)$ such that $L(A) = L(M)$. The automaton $A$ walks on a tree $t$ in a single pass from left to right. It enters each subtree of $t$ with root $u$ in state $q_{\text{in}}$ and leaves it in state $\text{state}_{M,t}(u)$. It uses the pushdown store to remember, (a) how many children of $u$ have already been processed and (b) what the resulting states were. Therefore we define $Q' = Q \cup \{q_{\text{in}}, q_{\text{fin}}\}$

(with $q_{\rm in}, q_{\rm fin} \notin Q$), $\Gamma = \bigcup\{Q^i \mid i \in \mathrm{rki}(\Sigma)\} \cup \{\lambda\}$ and $\gamma_{\rm in} = \lambda$. The push-down symbol $\lambda$ (the empty string) indicates that no subtrees have been traversed yet. If the automaton is at node $u$ and the pushdown symbol at the top of the pushdown store is $q_1 \cdots q_k$, this means that the first $k$ subtrees of $u$ have been traversed and that, for every $i$ with $1 \leq i \leq k$, $\mathrm{state}_{M,t}(u \cdot i) = q_i$.

The transition relation $\delta'$ is as follows:

$$
\begin{aligned}
\delta' \quad = \quad & \{(q_{\rm in}, (\neg\mathrm{root}, \mathrm{lab}_\sigma, \uparrow), \delta_\sigma) \mid \sigma \in \Sigma, \mathrm{rk}(\sigma) = 0\} \cup \\
& \{(q_{\rm in}, (\mathrm{root}, \mathrm{lab}_\sigma), q_{\rm fin}) \mid \sigma \in \Sigma, \mathrm{rk}(\sigma) = 0, \delta_\sigma \in F\} \cup \\
& \{(q_{\rm in}, (\mathrm{lab}_\sigma, \downarrow_{1,\lambda}), q_{\rm in}) \mid \sigma \in \Sigma, \mathrm{rk}(\sigma) > 0\} \cup \\
& \{(q_k, (\mathrm{lab}_\sigma, \mathrm{stay}_{q_1\cdots q_{k-1}, q_1\cdots q_k}, \downarrow_{k+1,\lambda}), q_{\rm in}) \mid \\
& \qquad\qquad \sigma \in \Sigma, \forall j \in [1,k](q_j \in Q), k < \mathrm{rk}(\sigma)\} \cup \\
& \{(q_k, (\neg\mathrm{root}, \mathrm{lab}_\sigma, \mathrm{stay}_{q_1\cdots q_{k-1}, q_1\cdots q_{k-1}}, \uparrow), \delta_\sigma(q_1, \ldots, q_k)) \mid \\
& \qquad\qquad \sigma \in \Sigma, \forall j \in [1,k](q_j \in Q), k = \mathrm{rk}(\sigma) > 0\} \cup \\
& \{(q_k, (\mathrm{root}, \mathrm{lab}_\sigma, \mathrm{stay}_{q_1\cdots q_{k-1}, q_1\cdots q_{k-1}}), q_{\rm fin}) \mid \\
& \qquad\qquad \sigma \in \Sigma, \forall j \in [1,k](q_j \in Q), k = \mathrm{rk}(\sigma) > 0, \delta_\sigma(q_1, \ldots, q_k) \in F\}
\end{aligned}
$$

□

**Lemma 17** *For every push-down tree-walking automaton $A$ there exists a finite tree automaton $M$ such that $L(M) = L(A)$.*

*Proof.* Let $A = (Q, D_{\rm PDTWA}(\Sigma, \Gamma), \delta, I, F)$ be a push-down tree-walking automaton over $(\Sigma, \Gamma)$. From this automaton, we first construct another push-down tree walking automaton $A'$. $A'$ is equal in behaviour to $A$, except that when $A$ is in a final state, $A'$ has extra transitions to move up to the root of the tree and then further up "out" of the tree in an extra state $q_{\rm fin}$. Note that $A'$ can never execute the latter instruction; it is added only to make the construction of $M$ easier. We define $A' = (Q', D_{\rm PDTWA}(\Sigma, \Gamma), \delta', I, \{q_{\rm fin}\})$ with $Q' = Q \cup \{q_{\rm fin}\}$, $q_{\rm fin} \notin Q$ and $\delta' = \delta \cup \{(q, \uparrow, q_{\rm fin}) \mid q \in F\} \cup \{(q_{\rm fin}, \uparrow, q_{\rm fin})\}$. We now construct a finite tree automaton $M = (Q_M, \Sigma, \delta_M, F_M)$ such that $L(M) = L(A)$. We use the method of transition tables (see, e.g., [KS81]). In this method, the states of the finite tree automaton are relations $R \subseteq (\Gamma \times Q') \times Q'$. When the finite tree automaton computes a relation $R$ as state for a node $u$ with $((\gamma, q), q') \in R$ , this means that the push-down tree-walking automaton $A'$ can traverse the subtree with $u$ as root, starting at node $u$ in state $q$ with symbol $\gamma$ on the top of the push-down store, and emerging from the subtree (by an edge from $u$ to its parent) in state $q'$, with $\gamma$ removed from the push-down store. At the root, $A'$ "emerging" from the tree in state $q_{\rm fin}$ means that $A$ arrived in a final state.

Formally, we define

$$
\begin{aligned}
Q_M &= 2^{(\Gamma \times Q') \times Q'} \\
\delta_M &= \{\delta_\sigma \mid \sigma \in \Sigma\}, \text{ with } \delta_\sigma \ (\sigma \in \Sigma) \text{ defined below} \\
F_M &= \{R \subseteq (\Gamma \times Q') \times Q' \mid ((\gamma_{\rm in}, q_{\rm in}), q_{\rm fin}) \in R \text{ for some } q_{\rm in} \in I\}
\end{aligned}
$$

Here, for every $\sigma \in \Sigma$, the transition function for symbol $\sigma$ is $\delta_\sigma : {Q_M}^{\mathrm{rk}(\sigma)} \to Q_M$ with

$$\delta_\sigma(R_1, \ldots, R_n) \quad = \quad \{((\gamma, q), q'') \mid \gamma \in \Gamma, q, q'' \in Q' \text{ and } (\gamma, q)\, R(\sigma, R_1, \ldots, R_n)^*\, (\gamma', q')$$

$$\text{for some } (\gamma', q') \in \Gamma \times Q' \text{ with } (q', \uparrow, q'') \in \delta'\}.$$

In this definition, $R(\sigma, R_1, \ldots, R_n)$ is the binary relation on $\Gamma \times Q'$ that contains the results of the possible actions of the push-down tree-walking automaton $A'$ while staying on the same node or entering a subtree and emerging again from it. The symbol $\sigma$ is the symbol at the current position and the relations $R_i \subseteq (\Gamma \times Q') \times Q'$ are the transition tables of the children of the current node. The relation $R$ is formally defined as

$$\begin{aligned} R(\sigma, R_1, \ldots, R_n) \quad = \quad & \{((\gamma, q), (\gamma, q'')) \mid (q, \downarrow_{i,\gamma'}, q') \in \delta' \text{ and } ((\gamma', q'), q'') \in R_i \\ & \qquad\qquad \text{for some } i \in [1, \mathrm{rk}(\sigma)]\} \cup \\ & \{((\gamma, q), (\gamma', q')) \mid (q, \mathrm{stay}_{\gamma,\gamma'}, q') \in \delta'\} \cup \\ & \{((\gamma, q), (\gamma, q')) \mid (q, \mathrm{lab}_\sigma, q') \in \delta'\}. \end{aligned}$$

Note that these relations can effectively be computed. The computation of $R(\sigma, R_1, \ldots, R_n)$ is straightforward. Since it is a binary relation on a finite set, its reflexive transition closure can be computed by regarding the relation as a graph and computing all paths in the graph (the nodes of the graph are the elements of $\Gamma \times Q'$ and there is an edge from $(\gamma_1, q_1)$ to $(\gamma_2, q_2)$ iff $((\gamma_1, q_1), (\gamma_2, q_2))$ is in the relation). □

Combining these two lemmas yields the following theorem.

**Theorem 18** *PDTWA = REGT*

## 3.5 Tree-Walking Marble Automata

Another way to extend tree-walking automata is with *marbles*. The concept of marbles is the same as that of Section 2.3 for strings, but they are used here in a different way. Marbles are coloured (with symbols from an alphabet) and there are infinitely many marbles of each colour available. Like pebbles, the automaton can drop a marble at a certain node, check for its presence any later time and lift it again. It is not allowed to put more than one marble of the same colour on the same node. Note that by taking the powerset of the marble alphabet we can even pose that not more than one marble (of any colour) is on any node. In the rest of this paper we will implicitly assume that this is the case. Unlike pebbles, putting down a marble on a certain node restrains the automaton to the subtree of which that node is the root, until the marble is lifted again. This restriction makes sure that the marbles are "almost" nested: unnested use of marbles is only possible if the marbles are at the same node. We will show that this restriction makes the tree-walking marble automaton very similar to the push-down tree-walking automaton, while it allows for the definition of the binary relation computed by a tree-walking marble automaton on a tree, in a natural way.

Let $\Sigma$ be a ranked alphabet and let $\Gamma$ be an alphabet of marble labels. Note that there is no $\gamma_{\mathrm{in}}$ necessary now. For a tree-walking marble automaton over $(\Sigma, \Gamma)$, we define the set of directives as

$$D_{\mathrm{TWMA}}(\Sigma, \Gamma) = D_{\mathrm{TWA}}(\Sigma) \cup \{\mathrm{put}_\gamma, \mathrm{lift}_\gamma, \mathrm{here}_\gamma, \neg\mathrm{here}_\gamma \mid \gamma \in \Gamma\}.$$

The meaning of the extra directives is as follows:

- $\text{put}_\gamma$ means drop a marble with label (or "colour") $\gamma$ on the current node, if one is not yet there.
- $\text{lift}_\gamma$ means pick up a marble with label $\gamma$, if one is here.
- $\text{here}_\gamma$ means check whether there is a marble with label $\gamma$ at the current node.
- $\neg\text{here}_\gamma$ is the negation of $\text{here}_\gamma$.

For $t \in T_\Sigma$, the position of the marbles is formalized by a function $m : \Gamma \to 2^{V_t}$. For this function, $m(\gamma) = S \subseteq V_t$ means that exactly the nodes in $S$ have a marble with label $\gamma$. The set of all marble position functions is denoted by $M_{t,\Gamma} = \Gamma \to 2^{V_t}$. The behaviour of the directives, including the restrictions mentioned above, is formalized by defining for each tree $t \in T_\Sigma$ and each directive $d \in D_{\text{TWMA}}(\Sigma, \Gamma)$ the following binary relation $R_t(d)$ on $V_t \times M_{t,\Gamma}$.

$$\begin{aligned}
R_t(\uparrow_i) &= \{((u,m),(u',m)) \mid u \text{ is the } i\text{th child of } u' \text{ and for all } \gamma \in \Gamma,\, u \notin m(\gamma)\}\\
R_t(\downarrow_i) &= \{((u,m),(u',m)) \mid u' \text{ is the } i\text{th child of } u\}\\
R_t(\text{root}) &= \{((\text{root}_t, m),(\text{root}_t, m))\}\\
R_t(\neg\text{root}) &= \{((u,m),(u,m)) \mid u \neq \text{root}_t\}\\
R_t(\text{lab}_\sigma) &= \{((u,m),(u,m)) \mid \text{lab}_t(u) = \sigma\}\\
R_t(\text{put}_\gamma) &= \{((u,m),(u,m')) \mid u \notin m(\gamma), m' = m(\gamma \mapsto m(\gamma) \cup \{u\})\}\\
R_t(\text{lift}_\gamma) &= \{((u,m),(u,m')) \mid u \in m(\gamma), m' = m(\gamma \mapsto m(\gamma) \setminus \{u\})\}\\
R_t(\text{here}_\gamma) &= \{((u,m),(u,m)) \mid u \in m(\gamma)\}\\
R_t(\neg\text{here}_\gamma) &= \{((u,m),(u,m)) \mid u \notin m(\gamma)\}
\end{aligned}$$

The relation for $d = \uparrow_i$ makes sure that the marbles are used nested, since it only allows moving up if there are no marbles on the current node.

Let $\Sigma$ be a ranked alphabet and let $\Gamma$ be an alphabet. A *tree-walking marble automaton* over $(\Sigma, \Gamma)$ is a finite automaton $A$ over $D_{\text{TWMA}}(\Sigma, \Gamma)$. For a tree-walking marble automaton $A = (Q, D_{\text{TWMA}}(\Sigma, \Gamma), \delta, I, F)$ over $(\Sigma, \Gamma)$ and a tree $t \in T_\Sigma$, the configurations of $A$ are triples $(q, u, m) \in Q \times V_t \times M_{t,\Gamma}$, with $q$ the state of the automaton, $u$ the current node and $m$ the marble position function. The set of all configurations of $A$ and $t$ is denoted by $\mathbb{C}_{A,t}$.

Let $A = (Q, D_{\text{TWMA}}(\Sigma, \Gamma), \delta, I, F)$ be a tree-walking marble automaton over $(\Sigma, \Gamma)$ and let $t \in T_\Sigma$. As usual, we define the binary step-relation $\twoheadrightarrow_{A,t}$ on $\mathbb{C}_{A,t}$. For every $(q,u,m), (q',u',m') \in \mathbb{C}_{A,t}$,

$$(q,u,m) \twoheadrightarrow_{A,t} (q',u',m') \text{ iff } \exists d \in D_{\text{TWMA}}(\Sigma, \Gamma) : (q,d,q') \in \delta \text{ and } (u,m)\, R_t(d)\, (u',m').$$

The automaton $A$ computes on each tree $t \in T_\Sigma$ the binary relation

$$R_t(A) = \left\{(u,u') \in V_t \times V_t \mid (q_{\text{in}}, u, m_0) \twoheadrightarrow^*_{A,t} (q_{\text{fin}}, u', m_0) \text{ for some } q_{\text{in}} \in I \text{ and } q_{\text{fin}} \in F\right\},$$

where $m_0 : \Gamma \to 2^{V_t}$ with $m_0(\gamma) = \emptyset$ for all $\gamma \in \Gamma$. Note that this definition demands that $A$ removes all marbles from the tree. The node relation that $A$ computes is, as usual,

$$R(A) = \{(t,u,v) \mid t \in T_\Sigma \text{ and } (u,v) \in R_t(A)\}.$$

The tree language that $A$ recognizes is also defined in the familiar way:

$$L(A) = \{t \in T_\Sigma \mid (t, \mathrm{root}_t, v) \in R(A) \text{ for some } v \in V_t\}.$$

The class of all tree languages recognized by a tree-walking marble automaton is named TWMA. Transitions of the form $(q, s, q')$ with $s \in D_{\mathrm{TWMA}}(\Sigma, \Gamma)^*$ are treated in the usual way. Determinism of tree-walking marble automata is defined in the same way as of (simple) tree-walking automata. The following pairs of directives in $D_{\mathrm{TWMA}}(\Sigma, \Gamma)$ are mutually exclusive:

- $\{\uparrow_i, \uparrow_j\}$ with $i \neq j$.
- $\{\uparrow_i, \mathrm{root}\}$.
- $\{\mathrm{root}, \neg\mathrm{root}\}$.
- $\{\mathrm{lab}_{\sigma_1}, \mathrm{lab}_{\sigma_2}\}$ with $\sigma_1 \neq \sigma_2$.
- $\{\downarrow_i, \mathrm{lab}_\sigma\}$ with $i > \mathrm{rk}(\sigma)$.
- $\{\uparrow_i, \mathrm{here}_\gamma\}$ and $\{\uparrow_i, \mathrm{lift}_\gamma\}$ with $i \in \mathrm{rki}(\Sigma)$ and $\gamma \in \Gamma$ (if there is any marble, the automaton is not allowed to move up).
- $\{\mathrm{put}_\gamma, \mathrm{lift}_\gamma\}$, $\{\mathrm{put}_\gamma, \mathrm{here}_\gamma\}$ and $\{\mathrm{lift}_\gamma, \neg\mathrm{here}_\gamma\}$ with $\gamma \in \Gamma$.

We define the abbreviation $\neg\mathrm{here}_\Gamma$ for $\neg\mathrm{here}_{\gamma_1}, \ldots, \neg\mathrm{here}_{\gamma_k}$ with $\Gamma = \{\gamma_1, \ldots, \gamma_k\}$. These directives together test that there is no marble on the current node.

Note that root and $\neg$root are needed for the tree-walking marble automaton because, unlike the push-down tree-walking automaton, a tree-walking marble automaton does not have to start its walk at the root of the tree, so it is not always possible to place a special marble at the root. Likewise, directives $\uparrow_i$ for $i \in \mathrm{rki}(\Sigma)$ are needed because, when the automaton starts in a certain node not equal to the root, the path from the root to that node cannot be coded into marbles.

**Theorem 19** *TWMA = REGT*

*Proof.* Since PDTWA=REGT (Theorem 18), it is sufficient to show that TWMA=PDTWA. First we prove that any push-down tree walking automaton can be simulated by a tree-walking marble automaton. Let $A = (Q, D_{\mathrm{PDTWA}}(\Sigma, \Gamma), \delta, I, F)$ be a push-down tree-walking automaton over $(\Sigma, \Gamma)$. We define a tree-walking marble automaton

$$A' = (Q \cup \{q_{\mathrm{in}}, q_{\mathrm{fin}}\}, D_{\mathrm{TWMA}}(\Sigma, \Gamma), \delta', \{q_{\mathrm{in}}\}, \{q_{\mathrm{fin}}\})$$

with a new initial state $q_{\mathrm{in}}$ and a new final state $q_{\mathrm{fin}}$. This automaton simulates $A$ in the following way. First, a marble with colour $\gamma_{\mathrm{in}}$, representing the initial push-down symbol, is put on the current node. Then the directive $\downarrow_{i,\gamma}$ is simulated by moving down to the $i$th child of the current node and then putting down a marble of colour $\gamma$. On the way up, the directive $\uparrow$ is simulated by picking up *any* marble (there is always exactly one on the current node) and moving up to the parent of the current node. The directive $\mathrm{lab}_\sigma$ is also available in $D_{\mathrm{TWMA}}(\Sigma, \Gamma)$, so it can be simulated directly. When $A$ reaches a final state, $A'$ moves up to the root, picking up all the marbles on the way up, so that it accepts the tree when it picks up the last marble at the root. A configuration of the push-down tree-walking automaton $(u, \gamma_1 \cdots \gamma_n)$ is simulated by a configuration $(u, m)$ of the tree-walking marble automaton, where, for all $i \in [1, n]$, $m$ has a marble $\gamma_i$ on the $i$th node of the path from the root to $u$, and no other marbles. $A'$ has the following transitions:

$$\begin{aligned}
\delta' \quad = \quad & \{(q_{\mathrm{in}}, \mathrm{put}_{\gamma_{\mathrm{in}}}, q) \mid q \in I\} \cup \\
& \{(q, (\mathrm{lift}_\gamma, \uparrow_i), q') \mid (q, \uparrow, q') \in \delta, \gamma \in \Gamma, i \in \mathrm{rki}(\Sigma)\} \cup \\
& \{(q, (\downarrow_i, \mathrm{put}_\gamma), q') \mid (q, \downarrow_{i,\gamma}, q') \in \delta\} \cup \\
& \{(q, \mathrm{lab}_\sigma, q') \mid (q, \mathrm{lab}_\sigma, q') \in \delta\} \cup \\
& \{(q, (\mathrm{lift}_{\gamma_1}, \mathrm{put}_{\gamma_2}), q') \mid (q, \mathrm{stay}_{\gamma_1,\gamma_2}, q') \in \delta\} \cup \\
& \{(q, \mathrm{lift}_\gamma, q_{\mathrm{fin}}) \mid q \in F, \gamma \in \Gamma\} \cup \\
& \{(q_{\mathrm{fin}}, (\uparrow_i, \mathrm{lift}_\gamma), q_{\mathrm{fin}}) \mid i \in \mathrm{rki}(\Sigma), \gamma \in \Gamma\}.
\end{aligned}$$

Obviously, $L(A') = L(A)$.

The other way around, let $A = (Q, D_{\mathrm{TWMA}}(\Sigma, \Gamma), \delta, I, F)$ be a tree-walking marble automaton over $(\Sigma, \Gamma)$. As observed before, we may assume that the automaton is constructed in such a way that, at any time, there is at most one marble on any node. Note that, due to the restriction of the automaton to the subtree of the current node whenever a marble is put down, at any moment all marbles lie on the path from the root to the current node. This path can be viewed as a push-down store, except that here "empty" places in the store are allowed. To account for these empty places, we add an extra marble colour $\gamma_0$. In order to simulate root, $\neg$root and the directives $\uparrow_i$ (for $i \in \mathrm{rki}(\Sigma)$), extra information is put on the push-down store. The new push-down symbols are of the form $(\gamma, i, r)$, where $\gamma \in \Gamma$ is the marble colour (or $\gamma = \gamma_0$ if there is no marble on the current node), the current node is the $i$th child of its father, and $r = 1$ if the current node is the root and $r = 0$ if the current node is not the root. The initial pushdown symbol is $\gamma_{\mathrm{in}} = (\gamma_0, 1, 1)$. Now the push-down store can be used to simulate the marbles. We define a push-down tree-walking automaton $A' = (Q', D_{\mathrm{PDTWA}}(\Sigma, \Gamma'), \delta', I, F')$.

- When $A$ moves down ($\downarrow_i$), $A'$ also moves down and puts the symbol $(\gamma_0, i, 0)$ on the push-down store, which means that there is no marble on the (new) current node, the current node is the $i$th child of its father and the current node is not the root.
- When $A$ moves up ($\uparrow_i$), $A'$ checks the symbol on top of the push-down store to make sure that the child number is correct and that there is no marble on the current node.
- $A'$ can simulate root and $\neg$root by checking the symbol on top of the push-down store, $(\gamma, i, r)$ with $r = 1$ for root and $r = 0$ for $\neg$root.
- The directive $\mathrm{lab}_\sigma$ can be simulated directly.
- The directive $\mathrm{put}_\gamma$ can be simulated by first checking that there is no marble on the current node, so the symbol on top of the push-down store is $(\gamma_0, i, r)$ and then replacing $\gamma_0$ by $\gamma$.
- To simulate $\mathrm{lift}_\gamma$, $A'$ checks that there is a marble on the current node with colour $\gamma$ and then replaces it with "colour" $\gamma_0$.

- The directive $\text{here}_\gamma$ is simulated by inspecting the top symbol of the push-down store and making sure that it is of the form $(\gamma, i, r)$. Similarly for $\neg\text{here}_\gamma$ and a top symbol of any of the forms $(\gamma', i, r)$ with $\gamma' \neq \gamma$.

- When $A$ reaches a final state, $A'$ must make sure that there are no marbles left on the tree before it can accept the tree. Because all marbles are on the path from the current node to the root, $A'$ walks up to the root, checking the absence of marbles on the way, and enter the final state $q_{\text{fin}}$ when it arrives at the root and there is no marble there either.

Formally, the push-down alphabet and the transitions of $A'$ are defined as follows.

$$
\begin{aligned}
\Gamma' &= (\Gamma \cup \{\gamma_0\}) \times \text{rki}(\Sigma) \times \{0,1\} \text{ with } \gamma_{\text{in}} = (\gamma_0, 1, 1) \\
Q' &= Q \cup \{\text{up}, q_{\text{fin}}\} \\
F' &= \{q_{\text{fin}}\} \\
\delta' &= \{(q, (\text{stay}_{(\gamma_0,i,r),(\gamma_0,i,r)}, \uparrow), q') \mid r \in \{0,1\}, (q, \uparrow_i, q') \in \delta\} \cup \\
&\quad \{(q, \downarrow_{i,(\gamma_0,i,0)}, q') \mid (q, \downarrow_i, q') \in \delta\} \cup \\
&\quad \{(q, \text{stay}_{(\gamma,i,1),(\gamma,i,1)}, q') \mid \gamma \in \Gamma \cup \{\gamma_0\}, i \in \text{rki}(\Sigma), (q, \text{root}, q') \in \delta\} \cup \\
&\quad \{(q, \text{stay}_{(\gamma,i,0),(\gamma,i,0)}, q') \mid \gamma \in \Gamma \cup \{\gamma_0\}, i \in \text{rki}(\Sigma), (q, \neg\text{root}, q') \in \delta\} \cup \\
&\quad \{(q, \text{lab}_\sigma, q') \mid (q, \text{lab}_\sigma, q') \in \delta\} \cup \\
&\quad \{(q, \text{stay}_{(\gamma_0,i,r),(\gamma,i,r)}, q') \mid i \in \text{rki}(\Sigma), r \in \{0,1\}, (q, \text{put}_\gamma, q') \in \delta\} \cup \\
&\quad \{(q, \text{stay}_{(\gamma,i,r),(\gamma_0,i,r)}, q') \mid i \in \text{rki}(\Sigma), r \in \{0,1\}, (q, \text{lift}_\gamma, q') \in \delta\} \cup \\
&\quad \{(q, \text{stay}_{(\gamma,i,r),(\gamma,i,r)}, q') \mid i \in \text{rki}(\Sigma), r \in \{0,1\}, (q, \text{here}_\gamma, q') \in \delta\} \cup \\
&\quad \{(q, \text{stay}_{(\gamma,i,r),(\gamma,i,r)}, q') \mid i \in \text{rki}(\Sigma), r \in \{0,1\}, \gamma \in (\Gamma \cup \{\gamma_0\}), \\
&\qquad\qquad (q, \neg\text{here}_{\gamma'}, q') \in \delta \text{ for some } \gamma' \neq \gamma\} \cup \\
&\quad \{(q, \text{stay}_{(\gamma_0,i,1),(\gamma_0,i,1)}, q_{\text{fin}}) \mid q \in F, i \in \text{rki}(\Sigma)\} \cup \\
&\quad \{(q, (\text{stay}_{(\gamma_0,i,0),(\gamma_0,i,0)}, \uparrow), \text{up}) \mid q \in F, i \in \text{rki}(\Sigma)\} \cup \\
&\quad \{(\text{up}, (\text{stay}_{(\gamma_0,i,0),(\gamma_0,i,0)}, \uparrow), \text{up}) \mid i \in \text{rki}(\Sigma)\} \cup \\
&\quad \{(\text{up}, \text{stay}_{(\gamma_0,i,1),(\gamma_0,i,1)}, q_{\text{fin}}) \mid i \in \text{rki}(\Sigma)\}
\end{aligned}
$$

Since this automaton simulates $A$, we have $L(A') = L(A)$. □

One might wonder why it is necessary to restrict a tree-walking automaton, when it puts down a marble on a node, to the subtree of which that node is the root. We will show that just demanding that the marbles be used nested is not enough; this would allow the automaton to recognize non-regular tree languages. As an example, consider the non-regular tree language $L = \{t_n \mid n \in \mathbb{N}\}$ over the ranked alphabet $\Sigma = \{a, b, c\}$ with $\text{rk}(a) = \text{rk}(b) = 1$ and $\text{rk}(c) = 0$, with, for all $n \in \mathbb{N}$,

$$
\begin{aligned}
V_{t_n} &= \{x_i, y_i \mid i \in [1,n]\} \cup \{z\} \\
E_{t_n} &= \{(x_i, 1, x_{i+1}), (y_i, 1, y_{i+1}) \mid i \in [1, n-1]\} \cup \{(x_n, 1, y_1), (y_n, 1, z)\} \\
\text{lab}_{t_n} &= \{(x_i, a), (y_i, b) \mid i \in [1,n]\} \cup \{(z, c)\}
\end{aligned}
$$

This tree language $L$ is similar to the (non-regular) language $\{a^n b^n c \mid n \in \mathbb{N}\}$. If we only demand the nested use of marbles, $L$ is recognized by the tree-walking marble automaton $A$ over $(\Sigma, \{\gamma\})$, that puts the first marble on the first "a", walks to the bottom of the tree, checks whether there is a "c" there and a "b" directly above, puts a marble on that "b", walks up the tree to the first "a" again, puts a marble on the second "a" and continues walking up and down until it reaches the middle, where it can check if there are as much "a"'s as "b"'s. It then removes the marbles in the reverse order, walking up and down the tree in a similar way.

Note that tree-walking multi-pebble automata, defined in analogy to the string-walking multi-pebble automata of Section 2.1, recognize only regular tree languages (but it is open whether or not they recognize them all). Multi-pebble automata have only a finite number of pebbles available, contrary to the infinite number of marbles of marble automata. This means the above described automaton will not work with a multi-pebble automaton.

## 3.6 Tree-Walking Marble/Pebble Automata

There is one feature to be added to tree-walking marble automata to make sure that they have the same power as binary MSO formulas with respect to describing binary node relations. A tree-walking marble/pebble automaton has one pebble (next to marbles) which it can use in the usual way. The only restriction is that the use of marbles and pebble together must be nested, e.g., it is not allowed to put down a marble, then the pebble and then pick up the marble before the pebble. Unlike the marbles, the pebble does not restrict the automaton to the subtree induced by all descendants of the node where the pebble is. The pebble will be used to mark one special node, e.g., the starting node of a walk on the tree. Like with tree-walking marble automata, we will always assume that the automaton is constructed in such a way that there is not more than one marble on any node, at any time.

Let $\Sigma$ be a ranked alphabet and let $\Gamma$ be an alphabet of marble colours. We define the set of directives for a tree-walking marble/pebble automaton as

$$\begin{aligned} D_{\mathrm{TWMPA}}(\Sigma,\Gamma) &= D_{\mathrm{TWMA}}(\Sigma,\Gamma) \cup \{\mathrm{put}, \mathrm{lift}, \mathrm{here}, \neg\mathrm{here}\} \\ &= \{\uparrow_i, \downarrow_i \mid i \in \mathrm{rki}(\Sigma)\} \cup \{\mathrm{root}, \neg\mathrm{root}\} \cup \{\mathrm{lab}_\sigma \mid \sigma \in \Sigma\} \cup \\ &\qquad \{\mathrm{put}_\gamma, \mathrm{lift}_\gamma, \mathrm{here}_\gamma, \neg\mathrm{here}_\gamma \mid \gamma \in \Gamma\} \cup \{\mathrm{put}, \mathrm{lift}, \mathrm{here}, \neg\mathrm{here}\} \end{aligned}$$

Here put, lift, here and ¬here (without subscripts) are the directives for the pebble, while $\mathrm{put}_\gamma$, $\mathrm{lift}_\gamma$, $\mathrm{here}_\gamma$ and $\neg\mathrm{here}_\gamma$ are the directives for a marble of colour $\gamma$. To enforce nesting, we now define the binary relations for the directives on triples $(u, k, p) \in V_t \times \mathbb{N} \times (\mathbb{N} \to \mathrm{MP}_t)$ with $\mathrm{MP}_t = (V_t \times \Gamma) \cup V_t \cup \{\bot\}$. Here an element $m$ of $\mathrm{MP}_t$ represents the action of putting down a marble of colour $\gamma$ on node $u$ ($m = (u, \gamma)$) or of putting down the pebble on node $u$ ($m = u$), or no action ($m = \bot$). Of a triple $(u, k, p)$, $u$ is the current node of the automaton, $k$ is the number of marbles and pebbles currently on the tree and $p$ is the marble/pebble placement function. Intuitively, $p(1), \ldots, p(k)$ are the actions executed in the past (in that order) of putting down marbles or the pebble. The function $p$ is used as a kind of push-down stack, putting another item on the stack if a marble or pebble is put down and removing the topmost item from the stack if a marble or pebble is lifted. At all times, $p(n) = \bot$ if $n > k$.

For each tree $t \in T_\Sigma$ and each directive $d \in D_{\mathrm{TWMPA}}(\Sigma, \Gamma)$ we define the following binary relations:

$$
\begin{aligned}
R_t(\uparrow_i) &= \{((u,k,p),(u',k,p)) \mid u \text{ is the } i\text{th child of } u' \text{ and} \\
&\qquad\qquad p(j) \neq (u,\gamma) \text{ for all } j \in [1,k], \gamma \in \Gamma\} \\
R_t(\downarrow_i) &= \{((u,k,p),(u',k,p)) \mid u' \text{ is the } i\text{th child of } u\} \\
R_t(\mathrm{root}) &= \{((\mathrm{root}_t,k,p),(\mathrm{root}_t,k,p))\} \\
R_t(\neg\mathrm{root}) &= \{((u,k,p),(u,k,p)) \mid u \neq \mathrm{root}_t\} \\
R_t(\mathrm{lab}_\sigma) &= \{((u,k,p),(u,k,p)) \mid \mathrm{lab}_t(u) = \sigma\} \\
R_t(\mathrm{put}_\gamma) &= \{((u,k,p),(u,k+1,p')) \mid p' = p(k+1 \mapsto (u,\gamma)), \\
&\qquad\qquad p(i) \neq (u,\gamma) \text{ for all } i \in [1,k]\} \\
R_t(\mathrm{lift}_\gamma) &= \{((u,k,p),(u,k-1,p')) \mid k \geq 1, p(k) = (u,\gamma), p' = p(k \mapsto \perp)\} \\
R_t(\mathrm{here}_\gamma) &= \{((u,k,p),(u,k,p)) \mid p(i) = (u,\gamma) \text{ for some } i \in [1,k]\} \\
R_t(\neg\mathrm{here}_\gamma) &= \{((u,k,p),(u,k,p)) \mid p(i) \neq (u,\gamma) \text{ for all } i \in [1,k]\} \\
R_t(\mathrm{put}) &= \{((u,k,p),(u,k+1,p')) \mid k \geq 0, p' = p(k+1 \mapsto u), \\
&\qquad\qquad p(i) \in \mathrm{MP}_t \setminus V_t \text{ for all } i \in [1,k]\} \\
R_t(\mathrm{lift}) &= \{((u,k,p),(u,k-1,p')) \mid k \geq 1, p(k) = u, p' = p(k \mapsto \perp)\} \\
R_t(\mathrm{here}) &= \{((u,k,p),(u,k,p)) \mid p(i) = u \text{ for some } i \in [1,k]\} \\
R_t(\neg\mathrm{here}) &= \{((u,k,p),(u,k,p)) \mid p(i) \neq u \text{ for all } i \in [1,k]\}
\end{aligned}
$$

Let $\Sigma$ be a ranked alphabet and let $\Gamma$ be an alphabet. A *tree-walking marble/pebble automaton* over $(\Sigma, \Gamma)$ is a finite automaton over $D_{\mathrm{TWMPA}}(\Sigma, \Gamma)$. We continue with the usual definitions. For a tree-walking marble/pebble automaton $A = (Q, D_{\mathrm{TWMPA}}(\Sigma,\Gamma), \delta, I, F)$ over $(\Sigma, \Gamma)$ and a tree $t \in T_\Sigma$, the configurations of $A$ are 4–tuples $(q,u,k,p) \in Q \times V_t \times \mathbb{N} \times (\mathbb{N} \to \mathrm{MP}_t)$, with $q$ the state, $u$ the current node, $k$ the number of marbles and pebbles currently on the tree and $p$ the marble/pebble placement function. The set of all configurations of $A$ and $t$ is denoted by $\mathbb{C}_{A,t}$. We define the binary step-relation $\twoheadrightarrow_{A,t}$ on $\mathbb{C}_{A,t}$ in the usual way. For every $(q,u,k,p),(q',u',k',p') \in \mathbb{C}_{A,t}$,

$$(q,u,k,p) \twoheadrightarrow_{A,t} (q',u',k',p') \text{ iff } \exists d \in D_{\mathrm{TWMPA}}(\Sigma,\Gamma) : (q,d,q') \in \delta \text{ and } (u,k,p)\, R_t(d)\, (u',k',p').$$

The automaton $A$ computes on each tree $t \in T_\Sigma$ the binary relation

$$R_t(A) = \{(u,u') \in V_t \times V_t \mid (q_{\mathrm{in}},u,0,p_0) \twoheadrightarrow^*_{A,t} (q_{\mathrm{fin}},u',0,p_0) \text{ for some } q_{\mathrm{in}} \in I,\, q_{\mathrm{fin}} \in F\},$$

where $p_0$ is the empty marble/pebble placement function, with $p_0(i) = \perp$ for all $i \in \mathbb{N}$. The node relation that $A$ computes is, as usual,

$$R(A) = \{(t,u,v) \mid t \in T_\Sigma \text{ and } (u,v) \in R_t(A)\}.$$

The tree language that $A$ recognizes is, also as usual,

$$L(A) = \{t \in T_\Sigma \mid (t, \mathrm{root}_t, v) \in R(A) \text{ for some } v \in V_t\}.$$

The class of all languages that are recognized by a tree-walking marble/pebble automaton is named TWMPA. Transitions of the form $(q,s,q')$ with $s \in D_{\mathrm{TWMPA}}(\Sigma,\Gamma)^*$ are treated in the usual way. The definition of deterministic tree-walking marble/pebble automata is equal to the definition of deterministic string-walking automata. The following pairs of directives in $D_{\mathrm{TWMPA}}(\Sigma,\Gamma)$ are mutually exclusive, next to those pairs already mutually exclusive for tree-walking marble automata:

- {put, lift}, {put, here} and {lift, ¬here}
- {lift, lift$_\gamma$} for any $\gamma \in \Gamma$ (since either the pebble or a marble was put down last and hence must be lifted first).

**Theorem 20** *TWMPA = REGT*

*Proof.* Because TWMA=REGT (Theorem 19) and because each tree-walking marble automaton is also a tree-walking marble/pebble automaton, it is sufficient to show that, for each tree-walking marble/pebble automaton $A$, there exists a tree-walking marble automaton $A'$ such that $L(A') = L(A)$.

Let $A = (Q, D_{\mathrm{TWMPA}}(\Sigma, \Gamma), \delta, I, F)$ be a tree-walking marble/pebble automaton over $(\Sigma, \Gamma)$. We may again assume that $A$ is constructed in such a way that, at any time, there is at most one marble on any node. We also assume that $A$ nevers puts the pebble on an empty tree, i.e., there is always at least one marble on the tree when $A$ puts down the pebble. This can easily be accomplished by adding an extra marble colour and having $A$ put one on the root before starting. This also allows us to assume that $A$ does not use the directives root and ¬root.

We first formulate and prove the following Claim. This Claim states that for a situation where $A$ just put down its pebble on the current node $u$ and the previous marble (of colour $\gamma$) was put on an ancestor $v$ of $u$, there exists a finite tree automaton $M$ such that $A$ can make a round-trip, starting and finishing at $u$, starting in state $q$ and finishing in state $q'$, with the same marbles on the tree at the beginning and at the end of the round-trip (and without lifting the pebble at $u$), if and only if $M$ accepts $t_v$ (the subtree of $t$ with root $v$), marked at node $u$. Due to the nesting constraints, the round-trip of $A$ takes place only on $t_v$ ($A$ cannot move higher up the tree than $v$ before lifting the marble at $u$).

**Claim** *Let $A = (Q, D_{\mathrm{TWMPA}}(\Sigma, \Gamma), \delta, I, F)$ be a tree-walking marble/pebble automaton over $(\Sigma, \Gamma)$ that does not make use of the directives put, lift,* root, *and* ¬root. *Then there exists for every $\gamma \in \Gamma$ and $q, q' \in Q$ a finite tree automaton $M$ over $\Sigma \cup (\Sigma \times B_1)$ such that, for all $t \in T_\Sigma$, $u, v \in V_t$, $k \geq 2$, $p \in \mathbb{N} \to MP_t$ with $p(k) = u$ and $p(k-1) = (v, \gamma)$, with $u \in \mathrm{des}(v)$ and the only marble on $t_v$ is at $v$ (i.e., $\forall i < k$: if $p(i) = (v', \gamma')$ then $v' \notin \mathrm{des}(v)$),*

$$(q, u, k, p) \twoheadrightarrow^*_{A,t} (q', u, k, p) \text{ iff } \mathit{mark}(t_v, u) \in L(M).$$

To prove this Claim, let $A = (Q, D_{\mathrm{TWMPA}}(\Sigma, \Gamma), \delta, I, F)$ be a tree-walking marble/pebble automaton over $(\Sigma, \Gamma)$ that does not make use of the directives put, lift, root, and ¬root, and let $\gamma_r \in \Gamma$ (the $\gamma$ in the Claim) and $q_1, q_2 \in Q$. We first define a tree-walking marble automaton $A' = (Q', D_{\mathrm{TWMA}}(\Sigma \cup (\Sigma \times B_1), \Gamma), \delta', I', F')$ over $(\Sigma \cup (\Sigma \times B_1), \Gamma)$ such that the Claim holds for $A'$ rather than $M$. The automaton $A'$ uses the mark to know the position of the pebble. It walks on $t_v$; there is no marble of colour $\gamma_r$ on $v$. $A'$ starts in the root of $t_v$, i.e., $v$. It first searches for the pebble, then simulates $A$, starting in state $q_1$. When it gets back to the pebble in state $q_2$, $A'$'s mission is accomplished.

$$Q' = Q \cup \{q_{\mathrm{in}}, q_{\mathrm{fin}}\}$$

$$
\begin{aligned}
I' &= \{q_{\text{in}}\} \\
F' &= \{q_{\text{fin}}\} \\
\delta' &= \{(q_{\text{in}}, \downarrow_i, q_{\text{in}}) \mid i \in \text{rki}(\Sigma)\} \cup \{(q_{\text{in}}, \text{lab}_{(\sigma,1)}, q_1) \mid \sigma \in \Sigma\} \cup \\
&\quad \{(p, d, q) \mid (p, d, q) \in \delta, d \notin \{\text{here}, \neg\text{here}, \neg\text{here}_{\gamma_r}, \text{put}_{\gamma_r}\}\} \cup \\
&\quad \{(p, \text{lab}_{(\sigma,1)}, q) \mid (p, \text{lab}_\sigma, q) \in \delta\} \cup \\
&\quad \{(p, \text{lab}_{(\sigma,1)}, q) \mid (p, \text{here}, q) \in \delta, \sigma \in \Sigma\}\} \cup \\
&\quad \{(p, \text{lab}_\sigma, q) \mid (p, \neg\text{here}, q) \in \delta, \sigma \in \Sigma\} \cup \\
&\quad \{(p, \text{root}, q) \mid (p, \text{here}_{\gamma_r}, q) \in \delta, \sigma \in \Sigma\} \cup \\
&\quad \{(p, (\neg\text{root}, \neg\text{here}_{\gamma_r}), q) \mid (p, \neg\text{here}_{\gamma_r}, q) \in \delta\} \cup \\
&\quad \{(p, (\neg\text{root}, \text{put}_{\gamma_r}), q) \mid (p, \text{put}_{\gamma_r}, q) \in \delta\} \cup \\
&\quad \{(q_2, \text{lab}_{(\sigma,1)}, q_{\text{fin}}) \mid \sigma \in \Sigma\}
\end{aligned}
$$

With this definition of $A'$, we have, for all $t \in T_\Sigma$, $u, v \in V_t$, $k \geq 2$, $p \in \mathbb{N} \to \text{MP}_t$ with $p(k) = u$ and $p(k-1) = (v, \gamma)$, with $u \in \text{des}(v)$ and the only marble on $t_v$ at $v$,

$$(q_1, u, k, p) \twoheadrightarrow^*_{A,t} (q_2, u, k, p) \text{ iff } \text{mark}(t_v, u) \in L(A').$$

Because TWMA=REGT (Theorem 19), there exists a finite tree automaton $M$ over $\Sigma \cup (\Sigma \times B_1)$ with $L(M) = L(A')$. The automaton $M$ complies with the demands of the Claim. This ends the proof of the Claim.

We continue with the proof of Theorem 20. We construct a tree-walking marble automaton $A'$ that recognizes the same tree language as the tree-walking marble/pebble automaton $A$. The automaton $A' = (Q', D_{\text{TWMA}}(\Sigma, \Gamma'), \delta', I', F')$ simulates $A$, but at each step of $A$, $A'$ is prepared to simulate the use of the pebble of $A$, by knowing, if it were to put the pebble at the current node $u$, the states in which $A$ could return at the same node, ready to pick up the pebble again (cf. the proof of Theorem 11). To accomplish this, $A'$ uses a special set of marbles in addition to the marbles of $A$. For each $q, q' \in Q$ and $\gamma \in \Gamma$, $M_{qq'\gamma}$ is the automaton that can be constructed according to the Claim, after removing all transitions with directives put and lift from $A$. The meaning of the additional marble colours is as follows.

- The marbles $\hat{\gamma}$ (for $\gamma \in \Gamma$) denote that the last marble that $A$ has put down is of colour $\gamma$. The special marble $\epsilon$ is used to indicate that there are no marbles of $A$ on the tree.
- The marbles in the set "states" are strings of functions from $Q \times Q$ to $Q$. If the string is $f_1 \cdots f_k$ (with $k$ the rank of the symbol $\sigma$ of the node), then $f_i(q, q') = p$ means that $M_{qq'\gamma}$ reaches state $p$ in the $i$th child of the node. Here $\gamma$ is the colour of the latest marble put down by $A$ on the tree.
- The marbles $q \in Q$ denote that $A$ will continue its walk on the tree from there in state $q$. They are used to store the state of $A$ during the computation of the states-marble. They also mark the node where this computation started.
- The marbles in the set "succ" are functions from $Q \times Q$ to subsets of $Q$. A marble coloured $g : Q \times Q \to 2^Q$ on a node $u$ indicates that, for all $q, q' \in Q$, the finite automaton $M_{qq'\gamma}$ has successful states $g(q, q')$

at $u$. Here $M_{qq'\gamma}$ works on the subtree $t_v$, where $v$ is the node where the latest marble, of colour $\gamma$, was put by $A$.

$A'$ uses a method similar to the one used in Lemma 16 to compute the states of the $M_{qq'\gamma}$ and thus the marble colours for the marbles from the set "states". The successful states of the $M_{qq'\gamma}$ for the children of a node $u$ are computed by $A'$ from the successful states of $u$ itself and from the states that $M_{qq'\gamma}$ reaches in the children of $u$. When $A$ puts down its pebble (at node $u$), $A'$ checks whether $M_{qq'\gamma}$ accepts $\mathrm{mark}(t_v, u)$, where $q$ is the state of $A$ after it put down its pebble, $q'$ is a state from which $A$ can pick it up again, $\gamma$ is the colour of the last marble that $A$ put down before the pebble and $v$ is the position of that marble. The colour $\gamma$ is stored in the marble $\hat{\gamma}$ at the current node $u$. During the direct simulation of the steps of $A$, there is always exactly one marble from the set $\{\hat{\gamma} \mid \gamma \in \Gamma\} \cup \{\epsilon\}$ at the current node. When there is at least one marble of $A$ on the simulated tree, there is also exactly one marble from each of the sets states and succ at the current node. The colours of these three marbles determine whether or not the $M_{qq'\gamma}$ accept $\mathrm{mark}(t_v, u)$. We define the abbreviation $\neg\mathrm{here}_Q$ to be $\neg\mathrm{here}_{q_1}, \ldots, \neg\mathrm{here}_{q_n}$ with $\{q_1, \ldots, q_n\} = Q$.

Formally, the states, marble alphabet, initial and final states, and transitions of $A'$ are defined as follows.

$$\begin{aligned}
Q' &= Q \cup \{q_{\mathrm{in}}, q_{\mathrm{fin}}\} \cup \{\mathrm{findstates}_\gamma \mid \gamma \in \Gamma\} \cup \{(f, \gamma) \mid f : (Q \times Q) \to Q, \gamma \in \Gamma\} \\
\Gamma' &= \Gamma \cup Q \cup \{\hat{\gamma} \mid \gamma \in \Gamma\} \cup \{\epsilon\} \cup \mathrm{states} \cup \mathrm{succ} \\
\mathrm{states} &= \{\lambda\} \cup \bigcup_{i \in \mathrm{rki}(\Sigma)} ((Q \times Q) \to Q)^i \\
\mathrm{succ} &= (Q \times Q) \to 2^Q \\
I' &= \{q_{\mathrm{in}}\} \\
F' &= \{q_{\mathrm{fin}}\} \\
\delta' &= \delta_{\mathrm{in}} \cup \delta_{\mathrm{sim}} \cup \delta_{\mathrm{states}} \cup \delta_{\mathrm{fin}}
\end{aligned}$$

The set of transitions $\delta'$ of $A'$ is divided into four parts. The initialization of $A'$ is done in the transitions in $\delta_{\mathrm{in}}$. The actual simulation of the steps of $A$ is done by the transitions in $\delta_{\mathrm{sim}}$. The set $\delta_{\mathrm{states}}$ contains a "subroutine" for calculating the states of $M_{q_1 q_2 \gamma}$ for the children of the current node in a marble from the set states. Finally, the cleaning up is done by the transitions in $\delta_{\mathrm{fin}}$.

We start with describing $\delta_{\mathrm{in}}$. From the initial state $q_{\mathrm{in}}$ of $A'$, a marble $\epsilon$ is laid on the root. This marble indicates that there is no marble on the (simulated) tree yet. This transition prepares the automaton $A'$ for the simulation of $A$.

$$\delta_{\mathrm{in}} = \{(q_{\mathrm{in}}, \mathrm{put}_\epsilon, q) \mid q \in I\}$$

In $\delta_{\mathrm{sim}}$ we place the transitions for the actual simulation of the steps of $A$. This is quite technical. After simulating any of the directives $\{\downarrow_i, \mathrm{put}_\gamma, \mathrm{lift}_\gamma\}$ the marble from states must be re-computed. The marbles from succ are computed with the use of a function "su" that is defined after $\delta_{\mathrm{sim}}$. In the last set of transitions, the putting down and lifting of the pebble is simulated, by making use of the information in the marbles from states and succ.

$$\delta_{\mathrm{sim}} \;=\; \{(p, (\mathrm{here}_\epsilon, \downarrow_i, \mathrm{put}_\epsilon, q) \mid (p, \downarrow_i, q) \in \delta\} \cup$$

$$
\begin{aligned}
&\{(p, (\mathrm{lab}_\sigma, \mathrm{here}_{\hat{\gamma}}, \mathrm{here}_{f_1 \cdots f_k}, \mathrm{here}_g, \downarrow_i, \mathrm{put}_{\hat{\gamma}}, \mathrm{put}_{g'}, \mathrm{put}_q, \mathrm{put}_\lambda), \mathrm{findstates}_\gamma) \mid (p, \downarrow_i, q) \in \delta, \\
&\qquad\qquad \sigma \in \Sigma, k = \mathrm{rk}(\sigma), \gamma \in \Gamma, i \in [1,k], f_j : Q \times Q \to Q \; (j \in [1,k]), \\
&\qquad\qquad g : Q \times Q \to 2^Q, g' = \mathrm{su}(g, \sigma, \gamma, i, f_1, \ldots, f_k)\} \cup \\
&\{(p, (\mathrm{lift}_\epsilon, \uparrow_i), q) \mid (p, \uparrow_i, q) \in \delta\} \cup \\
&\{(p, (\neg \mathrm{here}_\Gamma, \mathrm{lift}_{\hat{\gamma}}, \mathrm{lift}_g, \mathrm{lift}_{f_1 \cdots f_k}, \uparrow_i), q) \mid (p, \uparrow_i, q) \in \delta, \\
&\qquad\qquad \gamma \in \Gamma, g : Q \times Q \to 2^Q, k \geq 0, f_j : Q \times Q \to Q \; (j \in [1,k])\} \cup \\
&\{(p, \mathrm{lab}_\sigma, q) \mid (p, \mathrm{lab}_\sigma, q) \in \delta\} \cup \\
&\{(p, (\mathrm{put}_\gamma, \mathrm{lift}_\epsilon, \mathrm{put}_{\hat{\gamma}}, \mathrm{put}_g, \mathrm{put}_q, \mathrm{put}_\lambda), \mathrm{findstates}_\gamma) \mid (p, \mathrm{put}_\gamma, q) \in \delta, \\
&\qquad\qquad g : Q \times Q \to 2^Q \text{ with } g(q_1, q_2) = F_{M_{q_1 q_2 \gamma}}\} \cup \\
&\{(p, (\mathrm{put}_\gamma, \mathrm{lift}_{\hat{\gamma}'}, \mathrm{put}_{\hat{\gamma}}, \mathrm{lift}_{f_1 \cdots f_k}, \mathrm{lift}_g, \mathrm{put}_{g'}, \mathrm{put}_q, \mathrm{put}_\lambda), \mathrm{findstates}_\gamma) \mid (p, \mathrm{put}_\gamma, q) \in \delta, \\
&\qquad\qquad \gamma' \in \Gamma, k \geq 0, f_j : Q \times Q \to Q \; (j \in [1,k]), g : Q \times Q \to 2^Q, \\
&\qquad\qquad g' : Q \times Q \to 2^Q \text{ with } g'(q_1, q_2) = F_{M_{q_1 q_2 \gamma}}\} \cup \\
&\{(p, (\mathrm{lift}_\gamma, \mathrm{lift}_{\hat{\gamma}}, \mathrm{lift}_{f_1 \cdots f_k}, \mathrm{lift}_g, \uparrow_i, \mathrm{here}_\epsilon, \downarrow_i, \mathrm{put}_\epsilon), q) \mid (p, \mathrm{lift}_\gamma, q) \in \delta, \\
&\qquad\qquad k \geq 0, f_j : Q \times Q \to Q \; (j \in [1,k]), g : Q \times Q \to 2^Q, i \in \mathrm{rki}(\Sigma)\} \cup \\
&\{(p, (\mathrm{lift}_\gamma, \mathrm{lift}_{\hat{\gamma}}, \mathrm{lift}_{f_1 \cdots f_k}, \mathrm{lift}_g, \uparrow_i, \mathrm{here}_{\hat{\gamma}'}, \mathrm{lab}_\sigma, \mathrm{here}_{f''_1 \cdots f''_{k''}}, \mathrm{here}_{g''}, \downarrow_i, \\
&\qquad\qquad \mathrm{put}_{\hat{\gamma}'}, \mathrm{put}_{g'}, \mathrm{put}_q, \mathrm{put}_\lambda), \mathrm{findstates}_{\gamma'}) \mid (p, \mathrm{lift}_\gamma, q) \in \delta, k \geq 0, \\
&\qquad\qquad f_j : Q \times Q \to Q \; (j \in [1,k]), g : Q \times Q \to 2^Q, i \in \mathrm{rki}(\Sigma), \gamma' \in \Gamma, \sigma \in \Sigma, \\
&\qquad\qquad k'' = \mathrm{rk}(\sigma), f''_j : Q \times Q \to Q \; (j \in [1,k'']), g'' : Q \times Q \to 2^Q, \\
&\qquad\qquad g' : Q \times Q \to 2^Q, g' = \mathrm{su}(g'', \sigma, \gamma', i, f''_1, \cdots, f''_{k''})\} \cup \\
&\{(p, (\mathrm{root}, \mathrm{lift}_\gamma, \mathrm{lift}_{\hat{\gamma}}, \mathrm{lift}_{f_1 \cdots f_k}, \mathrm{lift}_g, \mathrm{put}_\epsilon), q) \mid (p, \mathrm{lift}_\gamma, q) \in \delta, \\
&\qquad\qquad k \geq 0, f_j : Q \times Q \to Q \; (j \in [1,k]), g : Q \times Q \to 2^Q\} \cup \\
&\{(p, \mathrm{here}_\gamma, q) \mid (p, \mathrm{here}_\gamma, q) \in \delta\} \cup \\
&\{(p, \neg \mathrm{here}_\gamma, q) \mid (p, \neg \mathrm{here}_\gamma, q) \in \delta\} \cup \\
&\{(p, (\mathrm{lab}_\sigma, \mathrm{here}_{f_1 \cdots f_k}, \mathrm{here}_g, \mathrm{here}_{\hat{\gamma}}), q) \mid (p, \mathrm{put}, p') \in \delta, (q', \mathrm{lift}, q) \in \delta, \\
&\qquad\qquad \sigma \in \Sigma, k = \mathrm{rk}(\sigma), \gamma \in \Gamma, \\
&\qquad\qquad g : Q \times Q \to 2^Q, f_j : Q \times Q \to Q \; (j \in [1,k]), \\
&\qquad\qquad \left(\delta_{(\sigma,1)}\right)_{M_{p' q' \gamma}} (f_1(p', q'), \ldots, f_k(p', q')) \in g(p', q')\}
\end{aligned}
$$

Here $\mathrm{su}(g, \sigma, \gamma, i, f_1, \ldots, f_k)$ computes the next set of successful states. For all states $q_1, q_2 \in Q$, $\mathrm{su}(g, \sigma, \gamma, i, f_1, \ldots, f_k)(q_1, q_2)$ is the set of all states $q$ such that $M_{q_1 q_2 \gamma}$ is successful, assuming it reaches the $i$th child of the current node in state $q$, and the $j$th ($j \neq i$) child of the current node in state $f_j(q_1, q_2)$, or

$$
\begin{aligned}
\mathrm{su}(g, \sigma, \gamma, i, f_1, \ldots, f_k)(q_1, q_2) \quad = \quad \{q \in Q_{M_{q_1 q_2 \gamma}} \mid (\delta_\sigma)_{M_{q_1 q_2 \gamma}} (f_1(q_1, q_2), \ldots, f_{i-1}(q_1, q_2), q, \\
f_{i+1}(q_1, q_2), \ldots, f_k(q_1, q_2)) \in g(q_1, q_2)\}.
\end{aligned}
$$

The "subroutine" to construct the marble from states is as follows: $\delta_{\mathrm{states}} = \bigcup_{\gamma \in \Gamma} \delta_\gamma$ with

$$
\begin{aligned}
\delta_\gamma \quad = \quad &\{(\mathrm{findstates}_\gamma, (\neg \mathrm{here}_Q, \mathrm{lab}_\sigma, \mathrm{lift}_\lambda, \uparrow_i), (f, \gamma)) \mid i \in \mathrm{rki}(\Sigma), \mathrm{rk}(\sigma) = 0, \\
&\qquad\qquad f : (Q \times Q) \to Q \text{ with } f(q_1, q_2) = (\delta_\sigma)_{M_{q_1 q_2 \gamma}}\} \cup \\
&\{(\mathrm{findstates}_\gamma, (\mathrm{lift}_q, \mathrm{lab}_\sigma), q) \mid \mathrm{rk}(\sigma) = 0, q \in Q\} \\
&\{(\mathrm{findstates}_\gamma, (\mathrm{lab}_\sigma, \downarrow_1, \mathrm{put}_\lambda), \mathrm{findstates}_\gamma) \mid \mathrm{rk}(\sigma) > 0\} \cup \\
&\{((f_k, \gamma), (\mathrm{lab}_\sigma, \mathrm{lift}_{f_1 \cdots f_{k-1}}, \mathrm{put}_{f_1 \cdots f_k}, \downarrow_{k+1}, \mathrm{put}_\lambda), \mathrm{findstates}_\gamma) \mid 1 \leq k < \mathrm{rk}(\sigma)\} \cup
\end{aligned}
$$

$$\{((f_k,\gamma),(\neg\mathrm{here}_Q,\mathrm{lab}_\sigma,\mathrm{lift}_{f_1\cdots f_{k-1}},\uparrow_i),(f,\gamma)) \mid k=\mathrm{rk}(\sigma), i\in\mathrm{rki}(\Sigma),$$
$$f(q_1,q_2)=(\delta_\sigma)_{M_{q_1q_2\gamma}}(f_1(q_1,q_2),\ldots,f_k(q_1,q_2))\}\cup$$
$$\{((f_k,\gamma),(\mathrm{lift}_q,\mathrm{lab}_\sigma,\mathrm{lift}_{f_1\cdots f_{k-1}},\mathrm{put}_{f_1\cdots f_k}),q) \mid q\in Q, k=\mathrm{rk}(\sigma)\geq 1\}$$

Finally, to allow $A'$ to accept the tree, $A'$ has to remove its auxiliary pebbles from the tree. Since at a moment that $A$ would accept the tree there are no marbles on it, $A'$ only has to remove the marbles $\epsilon$.

$$\delta_{\mathrm{fin}}=\{(q,\mathrm{lift}_\epsilon,q_{\mathrm{fin}}) \mid q\in F\}\cup\{(q_{\mathrm{fin}},(\uparrow_i,\mathrm{lift}_\epsilon),q_{\mathrm{fin}}) \mid i\in\mathrm{rki}(\Sigma)\}$$

□

## 3.7 Tree-Walking Automata with MSO Tests

A completely different extension (i.e., not involving marbles or pebbles) of tree-walking automata is to add the capability to perform MSO tests (see [BE97], Section 1.4). Let $\Sigma$ be a ranked alphabet. The set of directives for a tree-walking automaton with MSO tests is

$$D_{\mathrm{TWA+M}}(\Sigma)=\{\uparrow_i,\downarrow_i \mid i\in\mathrm{rki}(\Sigma)\}\cup\mathrm{MSOL}_1(\Sigma).$$

As this is an infinite set, we have to choose a finite subset of $D_{\mathrm{TWA+M}}(\Sigma)$ for each tree-walking automaton with MSO tests. For each tree $t\in T_\Sigma$ and each directive $d\in D_{\mathrm{TWA+M}}(\Sigma)$ we define the following binary relation on $V_t$:

$$\begin{aligned} R_t(\uparrow_i) &= \{(u,v) \mid (v,i,u)\in E_t\}\\ R_t(\downarrow_i) &= \{(u,v) \mid (u,i,v)\in E_t\}\\ R_t(\psi(x)) &= \{(u,u) \mid (t,u)\models\psi(x)\}\end{aligned}$$

A *tree-walking automaton with MSO tests* over $\Sigma$ is a finite automaton $A$ over a finite subset of $D_{\mathrm{TWA+M}}(\Sigma)$. The definitions of configurations, $\twoheadrightarrow_{A,t}$, $R_t(A)$, $R(A)$ and $L(A)$ are unchanged. The definition of deterministic tree-walking automata with MSO tests is also unchanged with respect to the definition of deterministic tree-walking automata, although it is no longer possible to list all pairs of mutually exclusive directives. However, it is decidable whether a given pair of directives is mutually exclusive. Directives of the form $(q,s,q')$ with $s\in D_{\mathrm{TWA+M}}(\Sigma)^*$ are treated in the usual way. The class of all tree languages $L$ such that $L=L(A)$ for some tree-walking automaton with MSO tests $A$ is named TWA+M. As stated in Section 1.4, in [BE97] the following is proven.

**Proposition 21** *The following three statements hold:*

- *TWA+M=REGT*
- *For each ranked alphabet $\Sigma$ and each binary MSO formula $\phi(x,y)\in MSOL_2(\Sigma)$, there exists a tree-walking automaton with MSO tests $A$ such that $R(A)=R(\phi)$.*
- *For each ranked alphabet $\Sigma$ and each tree-walking automaton with MSO tests $A$ over $\Sigma$, there exists a binary MSO formula $\phi(x,y)\in MSOL_2(\Sigma)$ such that $R(\phi)=R(A)$.*

## 3.8 Equivalence of Binary MSO Formulas and Tree-Walking Marble/Pebble Automata

In this section we will show that binary MSO formulas, i.e., formulas $\phi(x, y) \in \text{MSOL}_2(\Sigma)$ for some ranked alphabet $\Sigma$, define the same node relations as those computed by tree-walking marble/pebble automata. First we will show that for each binary formula $\phi(x, y) \in \text{MSOL}_2(\Sigma)$ there exists a tree-walking marble/pebble automaton $A$ over $(\Sigma, \Gamma)$ (for some alphabet $\Gamma$) such that they compute the same relation, that is, $R(\phi) = R(A)$. We will make use of the second part of Proposition 21 to construct our tree-walking marble/pebble automaton, similar to the alternate proof of Theorem 7 (Section 1.6). Note that the original proof of Theorem 7 is not valid for trees, because, if this proof were translated to trees, the pebble and marbles would not be used nested. We need the following lemma. This lemma states that it is possible to simulate a unary MSO formula with a tree-walking marble/pebble automaton that makes a round-trip, checking the absence of all marbles and pebbles before entering a final state.

**Lemma 22** *Let $\Sigma$ be a ranked alphabet. For each unary MSO formula $\psi(x) \in MSOL_1(\Sigma)$ there exists a tree-walking marble/pebble automaton $A = (Q, D_{\text{TWMPA}}(\Sigma, \Gamma), \delta, I, F)$ over $(\Sigma, \Gamma)$, for some alphabet $\Gamma$, such that, for all $t \in T_\Sigma$, $R_t(A) = \{(u, u) \mid u \in V_t, (t, u) \models \psi(x)\}$ and, if $(q_{\text{in}}, u, 0, p_0) \twoheadrightarrow^*_{A,t} (q_{\text{fin}}, u', k, p)$ for some $u, u' \in V_t$, $q_{\text{in}} \in I$ and $q_{\text{fin}} \in F$, then $k = 0$ and $p = p_0$.*

*Proof.* Let $\Sigma$ be a ranked alphabet and let $\psi(x) \in \text{MSOL}_1(\Sigma)$ be a unary MSO formula. According to Corollary 3, there exists a closed MSO formula $\psi' \in \text{MSOL}_0(\Sigma \cup (\Sigma \times B_1))$ such that, for all $t \in T_\Sigma$ and $u \in V_t$, $(t, u) \models \psi(x)$ iff $\text{mark}(t, u) \models \psi'$. Since MSOT=REGT (Proposition 15) and TWMA=REGT (Theorem 19), there exists a tree-walking marble automaton

$$A' = (Q', D_{\text{TWMA}}(\Sigma \cup (\Sigma \times B_1), \Gamma), \delta', I', F')$$

over $(\Sigma \cup (\Sigma \times B_1), \Gamma)$ (for some alphabet $\Gamma$) such that $L(A') = L(\psi')$. This implies that for all $t \in T_\Sigma$, $u \in V_t$ and $t' = \text{mark}(t, u)$,

$$t' \models \psi' \text{ iff } (q_{\text{in}}, \text{root}_{t'}, m_0) \twoheadrightarrow^*_{A',t'} (q_{\text{fin}}, v, m_0) \text{ for some } q_{\text{in}} \in I',\ q_{\text{fin}} \in F' \text{ and } v \in V_t.$$

We now define the tree-walking marble/pebble automaton $A = (Q, D_{\text{TWMPA}}(\Sigma, \Gamma), \delta, I, F)$. This automaton first drops its pebble in order to remember the start of the walk. Then it walks to the root of the tree and starts simulating $A'$. It uses the pebble to determine when to interpret a node label $\sigma \in \Sigma$ as $\sigma$ (when there is no pebble at the current node) and when to interpret $\sigma$ as $(\sigma, 1)$ (when there is a pebble at the current node). When it reaches a final state, it walks to the root of the tree, checking on the way that there are no marbles (this can be done since the marbles always lie on the path from the root to the current node), so that $A'$ really accepts the marked tree. Then $A$ searches for the pebble, picks it up and finishes.

$$\begin{aligned}
Q &= Q' \cup \{q_{\text{in}}, \text{toroot}, \text{ready}, \text{findpebble}, q_{\text{fin}}\} \\
I &= \{q_{\text{in}}\} \\
F &= \{q_{\text{fin}}\} \\
\delta &= \{(q_{\text{in}}, \text{put}, \text{toroot})\} \cup \{(\text{toroot}, \uparrow_i, \text{toroot}) \mid i \in \text{rki}(\Sigma)\} \cup
\end{aligned}$$

$$\begin{aligned}
&\{(\text{toroot}, \text{root}, q) \mid q \in I'\} \cup \\
&\{(q, d, q') \mid (q, d, q') \in \delta', d \neq \text{lab}_\sigma \text{ for all } \sigma \in \Sigma\} \cup \\
&\{(q, (\neg\text{here}, \text{lab}_\sigma), q') \mid (q, \text{lab}_\sigma, q') \in \delta'\} \cup \\
&\{(q, (\text{here}, \text{lab}_\sigma), q') \mid (q, \text{lab}_{(\sigma,1)}, q') \in \delta'\} \cup \\
&\{(q, \neg\text{here}_\Gamma, \text{ready}) \mid q \in F'\} \cup \\
&\{(\text{ready}, (\uparrow_i, \neg\text{here}_\Gamma), \text{ready}) \mid i \in \text{rki}(\Sigma)\} \cup \\
&\{(\text{ready}, (\text{root}, \neg\text{here}_\Gamma), \text{findpebble})\} \cup \\
&\{(\text{findpebble}, (\neg\text{here}, \downarrow_i), \text{findpebble}) \mid i \in \text{rki}(\Sigma)\} \cup \\
&\{(\text{findpebble}, (\text{here}, \text{lift}), q_{\text{fin}})\}
\end{aligned}$$

Note that the construction of $A$ ensures that the marbles and pebble are used nested, because first the pebble is put down, then the marbles are used in the same way as $A'$ uses them (i.e., nested), then $A$ makes sure there are no marbles left on the tree and finally $A$ picks up the pebble.

From the construction of $A$ it follows that, for all $t \in T_\Sigma$, $u \in V_t$ and $t' = \text{mark}(t, u)$,

$$(q, \text{root}_{t'}, m_0) \twoheadrightarrow^*_{A',t'} (q', v, m_0) \text{ for some } q \in I',\ q' \in F' \text{ and } v \in V_t \text{ iff } (q_{\text{in}}, u, 0, p_0) \twoheadrightarrow^*_{A,t} (q_{\text{fin}}, u, 0, p_0).$$

This, and the details of the construction of $A$, implies $R_t(A) = \{(u, u) \mid u \in V_t, (t, u) \models \psi(x)\}$ and the other details of the lemma. □

**Lemma 23** *Let $\Sigma$ be a ranked alphabet and let $\phi(x, y) \in MSOL_2(\Sigma)$ be a binary MSO formula. Then there exists a tree-walking marble/pebble automaton $A$ such that $R(A) = R(\phi)$.*

*Proof.* According to Proposition 21, there exists a tree-walking automaton with MSO tests $A' = (Q', \Delta, \delta', I', F')$ such that $R(A') = R(\phi)$. We define a tree-walking marble/pebble automaton $A$ over $(\Sigma, \Gamma)$ that simulates $A'$. The automaton $A$ can directly simulate the directives $\{\uparrow_i, \downarrow_i \mid i \in \text{rki}(\Sigma)\}$. The directives $\psi(x) \in \text{MSOL}_1(\Sigma)$ are handled by constructing a tree-walking marble/pebble automaton according to Lemma 22 and using this automaton as a "subroutine" in $A$. We define $T = \{(q, d, q') \in \delta' \mid d \in \text{MSOL}_1(\Sigma)\}$, the set of all transitions with MSO tests in $A'$. For each transition $\tau = (q, \psi(x), q') \in T$, we use Lemma 22 to construct the tree-walking marble/pebble automaton $A_\tau$ over $(\Sigma, \Gamma_\tau)$ that simulates $\tau$. We will assume that the states of any automaton $A_\tau$ do not overlap with the states of any other $A_\tau$ or with the states of $A'$. We also assume that the marble alphabets $\Gamma_\tau$ are disjoint, or $\Gamma_\tau \cap \Gamma_{\tau'} = \emptyset$ for all $\tau, \tau' \in T$, $\tau \neq \tau'$. The marble alphabet of $A$ is $\Gamma = \bigcup_{\tau \in T} \Gamma_\tau$.

We define $A = (Q, D_{\text{TWMPA}}(\Sigma, \Gamma), \delta, I, F)$ as follows:

$$\begin{aligned}
Q &= Q' \cup \bigcup_{\tau \in T} Q_{A_\tau} \\
I &= I' \\
F &= F' \\
\delta &= \{\tau \in \delta' \mid \tau \notin T\} \cup \\
&\quad \{(q, (\text{put}, \text{lift}), q'), (q'', (\text{put}, \text{lift}), q''') \mid q' \in I_{A_\tau}, q'' \in F_{A_\tau}, \tau = (q, \psi(x), q''') \in T\} \cup \\
&\quad \bigcup_{\tau \in T} \delta_{A_\tau}
\end{aligned}$$

Here the (double) directive (put, lift) acts as a "nop" directive. It can easily be seen that $A$ complies with the demands of nesting and that $R(A) = R(A')$. $\square$

We now show the other way around.

**Lemma 24** *Let $\Sigma$ be a ranked alphabet and let $A$ be a tree-walking marble/pebble automaton over $(\Sigma, \Gamma)$. Then there exists a binary MSO formula $\phi(x, y) \in MSOL_2(\Sigma)$ such that $R(\phi) = R(A)$.*

*Proof.* From the automaton $A$ we first construct another tree-walking marble/pebble automaton $A'$ over $(\Sigma \cup (\Sigma \times B_2), \Gamma)$ such that, for all $t \in T_\Sigma$ and $u, v \in V_t$,

$$\text{mark}(t, u, v) \in L(A') \text{ iff } (t, u, v) \in R(A).$$

$A'$ first finds the node with label $(\sigma, 1, b)$, simulates $A$ and accepts the tree if $A$ has a final state in a node with label $(\sigma, b, 1)$. Because TWMPA=REGT (Theorem 20) and REGT=MSOT (Proposition 15), there exists a MSO formula $\psi \in \text{MSOL}_0(\Sigma \cup (\Sigma \times B_2))$ such that, for all $t' \in T_{\Sigma \cup (\Sigma \times B_2)}$,

$$t' \in L(\psi) \text{ iff } t' \in L(A').$$

Now, using Corollary 3, we obtain a binary MSO formula $\phi(x, y) \in \text{MSOL}_2(\Sigma)$ such that, for all $t \in T_\Sigma$ and $u, v \in V_t$,

$$(t, u, v) \in R(\phi) \text{ iff } \text{mark}(t, u, v) \in L(\psi)$$

and we have, for all $t \in T_\Sigma$ and $u, v \in V_t$,

$$(t, u, v) \in R(\phi) \text{ iff } (t, u, v) \in R(A).$$

$\square$

These two lemmas together prove the grand finale of this chapter.

**Theorem 25** *The following two statements hold:*

- *For every MSO formula $\phi(x, y) \in MSOL_2(\Sigma)$ there exists a tree-walking marble/pebble automaton $A$ over $(\Sigma, \Gamma)$, for some alphabet $\Gamma$, such that $R(A) = R(\phi)$.*
- *For every tree-walking marble/pebble automaton $A$ over $(\Sigma, \Gamma)$ there exists an MSO formula $\phi(x, y) \in MSOL_2(\Sigma)$ such that $R(\phi) = R(A)$.*

This theorem states that binary MSO formulas define the same node relations that tree-walking marble/pebble automata compute. Since the proof of this theorem, and all underlying theorems and lemmas, is constructive, we can actually construct an operational automaton with only local operations to compute the node relation that is defined by a given, descriptive, binary MSO formula, and vice versa.

# Conclusion and Recommendations

## Conclusion

In this paper we have described a number of string-walking and tree-walking automata and we have proved some of their properties, especially with regard to the binary node relations these automata compute. This study has been done in order to find an answer to the following question:

> Is it possible to define a type of tree-walking automaton, with only local operations, that computes the same binary node relations that binary MSO formulas recognize?

In Chapter 1 we have investigated the possibilities to define a string-walking automaton that computes the same binary relations on the positions of strings as binary MSO formulas. It was found that string-walking pebble automata have this property. It is even possible, for every binary MSO formula, to construct a string-walking pebble automaton that computes the same binary relation. The reverse is also true: for each string-walking pebble automaton, it is possible to construct a binary MSO formula that defines the same binary relation. It was also shown that the pebble is necessary. There is a binary MSO formula for which there is no string-walking automaton (without pebble) that computes the same binary relation. Since the pebble operations are local (the pebble can only be handled on the current position of the automaton), this answers our central question for strings.

To answer our central question for trees, we have defined in Chapter 3, among others, tree-walking pebble/marble automata. It has been shown that these automata compute exactly the binary node relations that can be defined by binary MSO formulas. Like with string-walking pebble automata, the appropriate automaton for each binary MSO formula can be constructed, and vice versa. Tree-walking marble/pebble automata have also only local operations. All pebble-handling and marble-handling occurs only on the current node. A note must be made that it may seem a bit complicated to have both marbles and a pebble available to the automaton. We have made no study to check whether it is possible to leave out, e.g., the pebble and still compute the same binary node relations. Also one might argue that the tree-walking marble/pebble automaton is not strictly local, since the automaton "remembers" in some way that it has dropped the pebble, further down the tree, when it is about to pick up the marble that was put down last before the pebble.

## Recommendations

There are a number of questions that have not been answered in this paper, but may be interesting for future study.

- Only binary node relations have been considered. Maybe there is also an operational way to compute the $k$-ary node relations that are defined by $k$-ary MSO formulas.
- The concept of string-walking marble automata (Section 1.4) may have to be adapted to make them recognize exactly the regular languages.
- Do tree-walking automata recognize exactly the regular tree languages?
- It has not been checked that tree-walking marble/pebble automata need the pebble to compute the relations defined by binary MSO formulas.
- Tree-walking multi-pebble automata (in analogy with string-walking multi-pebble automata) have not been studied. It is not known which tree languages they describe, and whether the number of pebbles induces a hierarchy on the recognized languages. It is known that tree-walking multi-pebble automata recognize *at most* the regular tree languages, or TWnPA $\subseteq$ REGT.
- It should be checked whether *deterministic* string-walking pebble automata recognize exactly the *functional* relations that are defined by binary MSO formulas. Also for deterministic tree-walking marble/pebble automata and (functional) binary MSO formulas.

# Bibliography


[BE97] R. Bloem and J. Engelfriet. Characterization of properties and relations defined in monadic second order logic on the nodes of trees. Technical Report 97-04, Leiden University, 1997.

[BH65] M. Blum and C. Hewitt. Automata on a 2-dimensional tape. In *Proc. 8th IEEE Symp. on Switching and Automata Theory*, pages 155–160, 1965.

[Büc60] J. Büchi. Weak second-order arithmetic and finite automata. *Z. Math. Logik Grundl. Math.*, 6:66–92, 1960.

[Don70] J. Doner. Tree acceptors and some of their applications. *J. of Comp. Syst. Sci.*, 4:406–451, 1970.

[GS84] F. Gécseg and M. Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Lamnguages, and Computation*. Addison-Wesley, Reading, Mass., 1979.

[KS81] T. Kamimura and G. Slutzki. Parallel and two-way automata on directed ordered acyclic graphs. *Inform. Contr.*, 49:10–51, 1981.

[RHE91] G. Rozenberg, H.J. Hoogeboom, and J. Engelfriet. Formele talen en automaten 1. Afdeling Wiskunde en Informatica, Rijksuniversiteit Leiden, 1991. In Dutch.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2:57–81, 1968.