Tree Regular Model Checking: A Simulation-Based Approach ⋆

Parosh Aziz Abdulla [a,2], Axel Legay [b,1], Julien d'Orso [c], Ahmed Rezine [a,2]

[a] *Dept. of Information Technology*
Uppsala University
P.O. Box 337
SE-751 05 Uppsala, Sweden

[b] *Université de Liège*
Institut Montefiore, B28
4000 Liège, Belgium

[c] *LIAFA - Université Paris 7*
175, rue du chevaleret
75013 Paris, France

Abstract

Regular model checking is the name of a family of techniques for analyzing infinite-state systems in which states are represented by words, sets of states by finite automata, and transitions by finite-state transducers. In this framework, the central problem is to compute the transitive closure of a transducer. Such a representation allows to compute the set of reachable states of the system and to detect loops between states. A main obstacle of this approach is that there exists many systems for which the reachable set of states is not regular. Recently, regular model checking has been extended to systems with tree-like architectures. In this paper, we provide a procedure, based on a new implementable acceleration technique, for computing the transitive closure of a tree transducer. The procedure consists of incrementally adding new transitions while merging states which are related according to a pre-defined equivalence relation. The equivalence is induced by a *downward* and an *upward* simulation relation which can be efficiently computed. Our technique can also be used to compute the set of reachable states without computing the transitive closure. We have implemented and applied our technique to various protocols.

Key words: Tree regular model checking, transducer, semi-algorithm, simulation, rewrite systems.

1 Introduction

Regular model checking is the name of a family of techniques for analyzing infinite-state systems in which states are represented by words, sets of states by finite automata, and transitions by finite automata operating on pairs of states, i.e. finite-state transducers. The central problem in regular model checking is to compute the transitive closure of a finite-state transducer. Such a representation allows to compute the set of reachable states of the system (which is useful to verify safety properties) and to detect loops between states (which is useful to verify liveness properties). However, computing the transitive closure is in general undecidable; consequently any method for solving the problem is necessarily incomplete. One of the goals of regular model checking is to provide semi-algorithms that terminate on many practical applications. Such semi-algorithms have already been successfully applied to parameterized systems with linear topologies, and to systems that operate on linear unbounded data structures such as queues, integers, reals, and hybrid systems [1–5].

While using a finite-word representation is well-suited for systems with a linear topology, many interesting infinite-state systems fall outside of its scope. This is either because the behavior of the system cannot be captured by a regular relation [6], or because the topology of the system is not linear. A solution to the latter problem is to extend the applicability of the regular model checking approach beyond systems with linear topologies.

The present work aims at extending the regular model checking approach to verify systems which operate on tree-like architectures. This includes several interesting protocols such as the percolate protocol [7] or the Tree-arbiter protocol [8].

To verify such systems, we use an extension of regular model checking called *tree regular model checking*, which was first introduced in [7,9,10]. In tree regular model checking, states of the systems are represented by trees, sets of states by tree automata, and transitions by tree automata operating on pairs of trees, i.e. tree transducers. As in the case of regular model checking, the central problem is to provide semi-algorithms for computing the transitive

⋆ The present article is an extended version of a paper which appears in the Proceedings of 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005).

Email addresses: parosh@it.uu.se (Parosh Aziz Abdulla), legay@montefiore.ulg.ac.be (Axel Legay), julien.dorso@liafa.jussieu.fr (Julien d'Orso), rahmed@it.uu.se (Ahmed Rezine).

[1] This author is supported by a F.R.I.A grant.
[2] This author is supported in part by the *Swedish Research Council*.

closure of a tree transducer. This problem was considered in [9,10]; however the proposed algorithms are inefficient or non-implementable.

In this work, we provide an efficient and implementable semi-algorithm for computing the transitive closure of a tree transducer. Starting from a tree transducer D, describing the set of transitions of the system, we derive a transducer, called the *history transducer* whose states are *columns* (words) of states of D. The history transducer characterizes the transitive closure of the rewriting relation corresponding to D. The set of states of the history transducer is infinite, which makes it inappropriate for computational purposes. Therefore, we present a method for computing a finite-state transducer which is an abstraction of the history transducer. The abstract transducer is generated on-the-fly by a procedure which starts from the original transducer D, and then incrementally adds new transitions and merges equivalent states. To compute the abstract transducer, we define an equivalence relation on columns (states of the history transducer). We identify *good* equivalence relations, i.e., equivalence relations which can be used by our on-the-fly algorithm. An equivalence relation is considered to be *good* if it satisfies the following two conditions:

- *Soundness and completeness:* merging two equivalent columns must not add any traces which are not present in the history transducer. Consequently, the abstract transducer accepts the same language as the history transducer (and therefore characterizes exactly the transitive closure of D).
- *Computability of the equivalence relation:* This allows on-the-fly merging of equivalent states during the generation of the abstract transducer.

We present a methodology for deriving good equivalence relations. More precisely, an equivalence relation is induced by two simulation relations; namely a *downward* and an *upward* simulation relation, both of which are defined on tree automata. We provide sufficient conditions on the simulation relations which guarantee that the induced equivalence is good. Furthermore, we give examples of concrete simulations which satisfy the sufficient conditions.

We also show that our technique can be directly adapted in order to compute the set of reachable states of a system without computing the transitive closure. When checking for safety properties, such an approach is often (but not always) more efficient.

We have implemented our algorithms in a prototype which we have applied to a number of protocols including a Two-Way Token protocol, the Percolate Protocol [7], a parametrized version of the Tree-arbiter protocol [8], and a tree-parametrized version of a leader election protocol.

Related Work: There are several works on efficient computation of transitive closures for *word* transducers [2,11,3–5]. There has also been some work on extending regular model checking to *tree* transducers [9,10]. However, all current algorithms devoted to the computation of the transitive closure of a *tree* transducer are not efficient or not implementable. In [9], we presented a method for computing transitive closures of tree transducers. The method presented in [9] is very heavy and relies on several layers of expensive automata-theoretic constructions. The method of this paper is much more light-weight and efficient, and can therefore be applied to a larger class of protocols. The work in [10] also considers tree transducers, but it is based on *widening* rather than acceleration. The idea is to compute successive powers of the transducer relation, and detect *increments* in the produced transducers. Based on the detected increments, the method makes a guess of the transitive closure. One of the main disadvantages of this work is that the widening procedure in [10] is not implemented. Furthermore, no efficient method is provided to detect the increments. This indicates that any potential implementation of the widening technique would be inefficient. In [11], a technique for computing the transitive closure of a word transducer is given. This technique is also based on computing simulations. However, as explained in Section 6, those simulations cannot be extended to trees, and therefore the technique of [11] cannot be applied to tree transducers. In [2], Dams, Lakhnech, and Steffen present a non-implemented extension of the word case to trees. However, this is done for top-down tree automata which are not closed under determinization (and thus many other operations). In [2], the authors consider several definitions of simulations and bisimulations on top-down tree automata without providing methods for computing them. Hence, it is not clear how to implement their algorithms.

Outline: In the next section, we introduce basic concepts related trees and tree automata. In Section 3, we describe tree relations and transducers. In Section 4, we introduce tree regular model checking. Section 5 introduces *history transducers* which characterize the transitive closure of a given transducer. In Section 6, we introduce *downward* and *upward* simulations on tree automata, and give sufficient conditions which guarantee that the induced equivalence relation is exact and computable. Section 7 gives an example of simulations which satisfy the sufficient conditions. In section 8, we describe how to compute the reachable states. In Section 9 we report on the results of running a prototype on a number of examples. Finally, in Section 10 we give conclusions and directions for future work. A description of our examples is given in Appendix A. Algorithms for computing the simulations are given in Appendix B.

2 Tree automata

In this section, we introduce some preliminaries on trees and tree automata that will be used in the paper. The reader interested by this theory can also consult [12] and [13] for more details.

A *ranked alphabet* is a pair (Σ, ρ), where Σ is a finite set of symbols and ρ is a mapping from Σ to the set of natural numbers $\mathbb{N}$. For a symbol $f \in \Sigma$, $\rho(f)$ is the *arity* of f. We use Σ_p to denote the set of symbols in Σ with arity p. Intuitively, each node in a tree is labeled with a symbol in Σ with the same arity as the out-degree (i.e. number of successors) of the node. Sometimes, we abuse notation and use Σ to denote the ranked alphabet (Σ, ρ).

Following [13], the nodes in a tree are represented by words over $\mathbb{N}$. More precisely, the empty word ϵ represents the root of the tree, while a node $b_1 b_2 ... b_k$ is a child of the node $b_1 b_2 ... b_{k-1}$. Each node is also labeled by a symbol from Σ. Formally, we have the following definition.

Definition 1 [**Trees**]
A tree T over a ranked alphabet Σ is a pair (S, λ), where

- *S, called the tree structure, is a finite set of sequences over $\mathbb{N}$ (i.e, a finite subset of $\mathbb{N}^*$). Each sequence n in S is called a node of T. If S contains a node $n = b_1 b_2 ... b_k$, then S will also contain the node $n' = b_1 b_2 ... b_{k-1}$, and the nodes $n_r = b_1 b_2 ... b_{k-1} r$, for $r : 0 \leq r < b_k$. We say that n' is the parent of n, and that n is a child of n'. A leaf of T is a node n which does not have any child, i.e., there is no $b \in \mathbb{N}$ with $nb \in S$.*
- *λ is a mapping from S to Σ. The number of children of n is equal to $\rho(\lambda(n))$. Observe that if n is a leaf then $\lambda(n) \in \Sigma_0$.*

We use $T(\Sigma)$ to denote the set of all trees over Σ. The term *tree language* is used to reference a possible infinite set of trees.

Example 2 *Let $\Sigma = \{n, t, N, T\}$, with $\rho(n) = \rho(t) = 0$ and $\rho(N) = \rho(T) = 2$. Consider the tree $T = (S, \lambda)$ over Σ defined as follows:*

- $S = \{\epsilon, 0, 1, 00, 01, 10, 11\}$;
- $\lambda(00) = \lambda(01) = \lambda(10) = n$;
- $\lambda(11) = t$;
- $\lambda(\epsilon) = \lambda(0) = \lambda(1) = N$.

A graphical representation of T is shown in Figure 1. A node is represented as a circle, with its equivalent word on the right-hand side. Symbols inside circles are the respective labels of the nodes. An edge in the graph relates a child node and its parent node. □

We now introduce tree automata which are used to describe tree languages. There exist various kinds of tree automata. In this paper, we use bottom-up tree automata since they are closed under all operations needed by the classical model checking procedure: intersection, union, minimization, determinization,

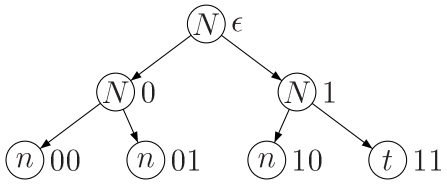

Fig. 1. A graphical view of the tree in Example 2.

inclusion test, complementation, etc. In the sequel, we will omit the term bottom-up.

Definition 3 [**Tree Automata**]
A tree automaton [13,12] over a ranked alphabet Σ is a tuple $A = (Q, F, \delta)$, where Q is a set of states, $F \subseteq Q$ is a set of final states, and δ is the transition relation, represented by a set of rules each of the form

$$(q_1, \ldots, q_p) \xrightarrow{f} q$$

where $f \in \Sigma_p$ and $q_1, \ldots, q_p, q \in Q$.

Unless stated otherwise, Q and δ are assumed to be finite. The tree automaton A is said to be deterministic when δ does not contain two rules of the form $(q_1, \ldots, q_p) \xrightarrow{f} q$ and $(q_1, \ldots, q_p) \xrightarrow{f} q'$ with $q \neq q'$.

Intuitively, the automaton A takes a tree $T \in T(\Sigma)$ as input. It proceeds from the leaves to the root, annotating states to the nodes of T. A transition rule of the form $(q_1, \ldots, q_p) \xrightarrow{f} q$ tells us that if the children of a node n are already annotated from left to right with $q_1, \ldots, q_p$ respectively, and if $\lambda(n) = f$, then the node n can be annotated by q. As a special case, a transition rule of the form $\xrightarrow{f} q$ implies that a leaf labeled with $f \in \Sigma_0$ can be annotated by q. The tree is accepted by the automaton if its root is labeled by a final state.

We now formally characterize the set of trees accepted by a tree automaton. We first define the notion of a *run*.

Definition 4 *A run r of $A = (Q, F, \delta)$ on a tree $T = (S, \lambda) \in T(\Sigma)$ is a mapping from S to Q such that for each node $n \in T$ with children $n_1, \ldots, n_k$ we have $\left((r(n_1), \ldots, r(n_k)) \xrightarrow{\lambda(n)} r(n)\right) \in \delta$.*

For a state q, we let $T \overset{r}{\Longrightarrow}_A q$ denote that r is a run of A on T such that $r(\epsilon) = q$. We use $T \Longrightarrow_A q$ denote that $T \overset{r}{\Longrightarrow}_A q$ for some r. For a set $S \subseteq Q$ of states, we let $T \overset{r}{\Longrightarrow}_A S$ ($T \Longrightarrow_A S$) denote that $T \overset{r}{\Longrightarrow}_A q$ ($T \Longrightarrow_A q$) for some $q \in S$. We say that A *accepts* T if $T \Longrightarrow_A F$. We define $L(A) = \{T |\, T \text{ is accepted by } A\}$. A tree language K is said to be *regular* if there is a tree automaton A such that $K = L(A)$.

Example 5 *Let* $\Sigma = \{n, t, N, T\}$*, with* $\rho(n) = \rho(t) = 0$ *and* $\rho(N) = \rho(T) = 2$*. Consider the automaton* $A = (Q, F, \delta)$*, where* $Q = \{q_0, q_1\}$ *and* $F = \{q_1\}$*. The transition rules in* δ *are as follows:*

$$\begin{array}{ccc} \xrightarrow{n} q_0 & \xrightarrow{t} q_1 & (q_0, q_0) \xrightarrow{T} q_1 \\ (q_0, q_0) \xrightarrow{N} q_0 & (q_0, q_1) \xrightarrow{N} q_1 & (q_1, q_0) \xrightarrow{N} q_1 \end{array}$$

The automaton A *accepts all trees over* Σ *containing exactly one occurrence of* t *or* T*. For instance, the tree of Example 2 is accepted by* A*.* $\square$

We now define the notion of context. Intuitively, a context is a tree with "holes" instead of leaves. Those holes are encoded with a special symbol $\square \notin \Sigma$ whose arity is 0. A *context* over Σ is a tree (S_C, λ_C) over $\Sigma \cup \{\square\}$ such that for all leaves $n_c \in S_C$, we have $\lambda_C(n_c) = \square$. For a context $C = (S_C, \lambda_C)$ with holes at leaves $n_1, \ldots, n_k \in S_C$, and trees $T_1 = (S_1, \lambda_1), \ldots, T_k = (S_k, \lambda_k)$, we define $C[T_1, \ldots, T_k]$ to be the tree (S, λ), where

- $S = S_C \cup \bigcup_{i \in \{1,\ldots,k\}} \{n_i \cdot n' |\ n' \in S_i\}$;
- for each $n = n_i \cdot n'$ with $n' \in S_i$ for some $1 \leq i \leq k$, we have $\lambda(n) = \lambda_i(n')$;
- for each $n \in S_C - \{n_1, \ldots, n_k\}$, we have $\lambda(n) = \lambda_C(n)$.

Intuitively, $C[T_1, \ldots, T_k]$ is the result of appending the trees $T_1, \ldots, T_k$ to the holes of C. Consider a tree automaton $A = (Q, F, \delta)$ over a ranked alphabet Σ. We extend the notion of runs to contexts. Let $C = (S_C, \lambda_C)$ be a context with leaves $n_1, \ldots, n_k$. A *run* r of A on C from $(q_1, \ldots, q_k)$ is defined in a similar manner to a run except that for leaf n_i, we have $r(n_i) = q_i$. In other words, each leaf labeled with $\square$ is annotated by one q_i. We use $C[q_1, \ldots, q_k] \stackrel{r}{\Longrightarrow}_A q$ to denote that r is a run of A on C from $(q_1, \ldots, q_k)$ such that $r(\epsilon) = q$. The notation $C[q_1, \ldots, q_k] \Longrightarrow_A q$ and its extension to sets of states are explained in a similar manner to runs on trees.

Definition 6 [**Suffix and Prefix**]
For an automaton $A = (Q, F, \delta)$*, we define the* suffix *of a tuple of states* $(q_1, \ldots, q_n)$ *to be* $\mathrm{suff}(q_1, \ldots, q_n) = \{C : \textit{context} |\ C[q_1, \ldots, q_n] \Longrightarrow_A F\}$*. For a state* $q \in Q$*, its* prefix *is the set of trees* $\mathrm{pref}(q) = \{T : \textit{tree} |\ T \Longrightarrow_A q\}$*.*

Remark 7 *Our definition of a context coincides with the one of [14] where all leaves are holes. On the other hand, a context in [13] and [9] is a tree with a single hole.*

3 Tree Relations and Transducers

In this section we introduce tree relations and transducers.

For a ranked alphabet Σ and $m \geq 1$, $\Sigma^\bullet(m)$ is the ranked alphabet which contains all tuples $(f_1, \ldots, f_m)$ such that $f_1, \ldots, f_m \in \Sigma_p$ for some p. We define $\rho((f_1, \ldots, f_m)) = \rho(f_1)$. In other words, the set $\Sigma^\bullet(m)$ contains the m-tuples, where all the elements in the same tuple have equal arities. Furthermore, the arity of a tuple in $\Sigma^\bullet(m)$ is equal to the arity of any of its elements. For trees $T_1 = (S_1, \lambda_1)$ and $T_2 = (S_2, \lambda_2)$, we say that T_1 and T_2 are *structurally equivalent*, denoted $T_1 \cong T_2$, if $S_1 = S_2$.

Consider structurally equivalent trees $T_1, \ldots, T_m$ over an alphabet Σ, where $T_i = (S, \lambda_i)$ for $i : 1 \leq i \leq m$. We let $T_1 \times \cdots \times T_m$ be the tree $T = (S, \lambda)$ over $\Sigma^\bullet(m)$ such that $\lambda(n) = (\lambda_1(n), \ldots, \lambda_m(n))$ for each $n \in S$. An m-ary *tree relation* on the alphabet Σ is a set of tuples of the form $(T_1, \ldots, T_m)$, where $T_1, \ldots, T_m \in T(\Sigma)$ and $T_1 \cong \cdots \cong T_m$. A tree language K over $\Sigma^\bullet(m)$ characterizes an m-ary tree relation $[K]$ on Σ as follows: $(T_1, \ldots, T_m) \in [K]$ iff $T_1 \times \cdots \times T_m \in K$. Tree automata can also be used to characterize tree relations. A tree automaton A over $\Sigma^\bullet(m)$ characterizes an m-ary tree relation on Σ, namely the relation $[L(A)]$. A tree relation is said to be *regular* if it is equal to $[L(A)]$, for some tree automaton A. In such a case, this relation is denoted by $R(A)$.

In [13], it is shown that regular tree languages are closed under all boolean operations. As a consequence, regularity of tree relations is preserved by the operators for union, intersection, and complementation.

For an n-ary relation R, and for $i : 1 \leq i \leq n$, we define the projection w.r.t. i as follows: $R|_i = \{(T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_n) | (T_1, \ldots, T_n) \in R\}$. We denote the usual composition of two binary relations R_1 and R_2 by $R_1 \circ R_2 = \{(T_1, T_2) \mid \exists T \text{ s.t. } (T_1, T) \in R_1 \wedge (T, T_2) \in R_2\}$. We define a notion of Cartesian product as follows. Let R_1 be an n-ary regular tree relation and R_2 be an m-ary regular tree relation. Then, the Cartesian product $R_1 \times R_2$ is the tree relation [3]

$$\{(T_1, \ldots, T_n, T'_1, \ldots, T'_m) \mid (T_1 \ldots, T_n) \in R_1 \wedge (T'_1 \ldots, T'_m) \in R_2 \wedge T_1 \cong T'_1\}.$$

The Cartesian product, projection and composition of regular tree relations also preserve regularity, as stated in the following Lemma:

[3] Note that because we define tree relations to include only tuples of structurally equivalent trees, the Cartesian product of tree relations should preserve that property.

Lemma 8 *The Cartesian product, projection and composition preserve regularity of tree relations.*

PROOF. First, consider the projection operation.

Consider a regular n-ary tree relation R, and index i. Let $A = (Q, F, \delta_A)$ be the automaton such that $R = [L(A)]$.

We construct the automaton $B = (Q, F, \delta_B)$ as follows. For each rule

$$(q_1, \ldots, q_p) \stackrel{(f_1,\ldots,f_n)}{\longrightarrow} q \in \delta_A$$

we add a corresponding rule

$$(q_1, \ldots, q_p) \stackrel{(f_1,\ldots,f_{i-1},f_{i+1},\ldots,f_n)}{\longrightarrow} q \in \delta_B$$

It is easy to see that $R|_i = [L(B)]$.

Next, we show that the Cartesian product preserves regularity of tree relations. Assume R_1 and R_2 are characterized respectively by tree automata $A_1 = (Q_1, F_1, \delta_1)$ and $A_2 = (Q_2, F_2, \delta_2)$. Then, the Cartesian product $R_1 \times R_2$ is recognized by the automaton $A = (Q, F, \delta)$ defined as follows:

- $Q = Q_1 \times Q_2$;
- $F = F_1 \times F_2$;
- If
  $$(q_1, \ldots, q_p) \xrightarrow{f_1} q \in \delta_1$$
  and
  $$\left(q'_1, \ldots, q'_p\right) \xrightarrow{f_2} q' \in \delta_2,$$
  then, we add the rule
  $$\left((q_1, q'_1), \ldots, (q_p, q'_p)\right) \stackrel{(f_1,f_2)}{\longrightarrow} (q, q') \in \delta$$

Finally, for the case of composition, consider binary tree relations R_1 and R_2. The claim follows from the equality:

$$R_1 \circ R_2 = ((R_1 \times T(\Sigma)) \cap (T(\Sigma) \times R_2))\,|_2$$

□

Although Lemma 8 states that regularity of relations is preserved by a finite number of applications of the $\circ$ operator, it is well-known that regularity is not preserved by application of an infinite number of compositions, even in the case of words.

In the rest of this work, we will use Id to denote the identity relation. For a binary relation R, R^+ is used to denote the transitive closure of R, i.e: $R^+ = \bigcup_{i=1}^{i=\infty} R^i$ with $R^i = \underbrace{R\circ\ldots\circ R}_{i\ times}$.

Definition 9 [**Tree Transducers**]
In the special case where D is a tree automaton over $\Sigma^\bullet(2)$, we call D a tree transducer over Σ.

Remark 10 *Our definition of tree transducers is a restricted version of the one considered in [10] in the sense that we only consider transducers that do not modify the structure of the tree. In [10], such transducers are called relabeling transducers.*

Example 11 *Let $\Sigma = \{n, t, N, T\}$, with $\rho(n) = \rho(t) = 0$ and $\rho(N) = \rho(T) = 2$. Consider the transducer $D = (Q, F, \delta)$, where $Q = \{q_0, q_1, q_2\}$ and $F = \{q_2\}$. The transition rules in δ are:*

$$\begin{array}{llllll} & \xrightarrow{(n,n)} q_0 & & \xrightarrow{(t,n)} q_1 & & \\ (q_0, q_0) \xrightarrow{(T,N)} q_1 & (q_0, q_1) \xrightarrow{(N,T)} q_2 & (q_1, q_0) \xrightarrow{(N,T)} q_2 \\ (q_0, q_0) \xrightarrow{(N,N)} q_0 & (q_0, q_2) \xrightarrow{(N,N)} q_2 & (q_2, q_0) \xrightarrow{(N,N)} q_2 \end{array}$$

The effect of (the relation denoted by) D is to move an (unique) occurrence of t or T one step toward the root of the tree. The meaning of the states of D is as follows:

- *in state q_0, the symbol t or T has never been met yet, and we don't change anything;*
- *in state q_1, we have just met symbol t or T, so we erase it at the current position;*
- *in state q_2, if one of the children nodes is in state q_1, then the current node is rewritten to T; otherwise, the change has occurred strictly below, and no change is needed.*

A graphical view of applying D on the tree T of Example 2 is shown in Figure 2. The left-hand tree shows the result of one application of D, while the right-hand tree shows the result of a second consecutive application. The output symbols only are shown in the nodes. On the right-hand side of each node n, we indicate the corresponding state $r(n)$ in the (unique) accepting run r. □

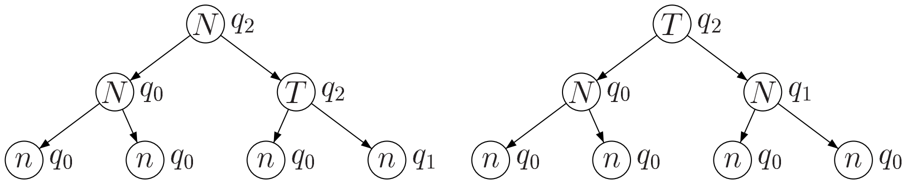

Fig. 2. Applying transducer D of Example 11.

4 Tree Regular Model Checking

In this section, we introduce the modeling framework for our systems. We use the framework known as *tree regular model checking* [9,10,7]. In our framework, we will use tree automata. In order to avoid confusion between the states of a system, and the states of an automaton representation, we will use the term *configuration* to denote a state of a system, or program.

Definition 12 **[Program]**
A program *is a triple* $P = (\Sigma, \phi_I, D)$ *where*

- Σ *is a ranked alphabet;*
- ϕ_I *is a set of initial configurations represented by a tree automaton over* Σ*;*
- D *is a transducer over* Σ *characterizing a transition relation* $R(D)$*.*

Example 13 *We consider a simple token protocol. Roughly speaking, the protocol consists of processes that are connected to form a binary tree. Each process stores a single bit which reflects whether the process has a token or not. In this system, the token can move from a leaf upward to the root in the following fashion: any process that currently has the token can release it to its parent neighbor. Initially, the system contains exactly one token, which can be located anywhere.*

In the regular model checking framework, we formalize this system by the program $P = (\Sigma, \phi_I, D)$ *where*

- $\Sigma = \{n, t, N, T\}$*, with* $\rho(n) = \rho(t) = 0$ *and* $\rho(N) = \rho(T) = 2$*;*
- ϕ_I *is the language of Example 5;*
- D *is the transducer of Example 11.*

□

In a similar manner to the case in which configurations are encoded by words, see [1,5], the problems we are going to consider are the following:

- *Computing the transitive closure of D:* The goal is to compute a new tree transducer D^+ representing the transitive closure of (the relation of) D, i.e., $R(D^+) = (R(D))^+$. Such a representation can be used for computing the reachability set of the program or for finding cycles between reachable program configurations.
- *Computing the reachable configurations:* The goal is to compute a tree automaton representing $R(D^+)(\phi_I)$. This set can be used for checking safety properties of the program as it is explained below.

Checking safety properties is often reduced to the reachability problem (see [15]) as follows. We are given a program $P = (\Sigma, \phi_I, D)$, and a safety property φ. We consider the set of so-called *bad configurations*, *Bad*. These are configurations which do not satisfy φ, and hence that the program should never reach. If the property φ is given as a regular set of program configurations, then the set of bad configurations is also a regular language (the complement operator preserves regularity), or equivalently an automaton. Then, the problem of checking whether the property φ holds in program P is reduced to checking that the set $R(D^+)(\phi_I) \cap Bad$ is empty. It is well-known how to perform intersection and emptiness testing for tree automata (see e.g. [13]). Hence the only remaining computation is the one of $R(D^+)(\phi_I)$.

In the sequel, we will first provide a technique for computing D^+. In Section 8, we will show the modifications needed for computing $R(D^+)(\phi_I)$ without computing the entire D^+.

5 Computing the Transitive Closure

In this section we introduce the notion of *history transducer*. With a transducer D we associate a *history transducer*, H, which corresponds to the transitive closure of D. Each state of H is a word of the form $q_1 \cdots q_k$, where $q_1, \ldots, q_k$ are states of D. For a word w, we let $w(i)$ denote the i-th symbol of w. Intuitively, for each $(T, T') \in R(D^+)$, the history transducer H encodes the successive runs of D needed to derive T' from T. The term "history transducer" reflects the fact that the transducer encodes the histories of all such derivations.

Definition 14 [**History Transducer**]
Consider a tree transducer $D = (Q, F, \delta)$ over a ranked alphabet Σ. The history (tree) transducer H for D is an (infinite) transducer (Q_H, F_H, δ_H), where $Q_H = Q^+$, $F_H = F^+$, and δ_H contains all rules of the form

$$(w_1, \ldots, w_p) \xrightarrow{(f,f')} w$$

such that there is $k \geq 1$ where the following conditions are satisfied

- $|w_1| = \cdots = |w_p| = |w| = k$;
- *there are* $f_1, f_2, \ldots, f_{k+1}$, *with* $f = f_1$, $f' = f_{k+1}$, *and* $(w_1(i) \ldots, w_p(i)) \overset{(f_i,f_{i+1})}{\longrightarrow} w(i)$ *belongs to* δ, *for each* $i : 1 \leq i \leq k$.

Observe that all the symbols $f_1, \ldots, f_{k+1}$ are of the same arity p. Also, notice that if $(T \times T') \overset{r}{\Longrightarrow}_H w$, then there is a $k \geq 1$ such that $|r(n)| = k$ for each $n \in (T \times T')$. In other words, any run of the history transducer assigns states (words) of the same length to the nodes. From the definition of H we derive the following lemma which states that H characterizes the transitive closure of the relation of D.

Lemma 15 *For a transducer* $D = (Q, F, \delta)$ *and its history transducer* $H = (Q_H, F_H, \delta_H)$, *we have that* $R(H) = R(D^+)$.

PROOF. (See also [9]).

We show inclusion in both direction.

$R(H) \subseteq R(D^+)$:

Consider $(T, T') \in R(H)$, and let r be an accepting run, i.e. $(T \times T') \overset{r}{\Longrightarrow}_H F_H$.

Let $k = |r(\epsilon)|$ be the size of the states encountered in the run r. For $i : 1 \leq i \leq k$, and for each node n in the structure of tree T, let $r_i(n) = (r(n))(i)$. We show that $r_1, \ldots, r_k$ are successive runs of D.

By definition of a run, for each node n with children $n_1, \ldots, n_p$, there is a rule: $(r(n_1), \ldots, r(n_p)) \overset{(f,f')}{\longrightarrow} r(n) \in \delta_H$.

By definition of H, there exist symbols $f_1, \ldots, f_{k+1}$ with $f = f_1$ and $f' = f_{k+1}$, such that $(r(n_1)(i), \ldots, r(n_p)(i)) \overset{(f_i,f_{i+1})}{\longrightarrow} r(n)(i) \in \delta$, for each $i : 1 \leq i \leq k$. We let $T_1, \ldots, T_{k+1}$ be (structurally equivalent) trees such that for node n, their labeling function is given by $\lambda_1(n) = f_1, \ldots, \lambda_{k+1}(n) = f_{k+1}$.

Observe that the rule $(r(n_1)(i), \ldots, r(n_p)(i)) \overset{(f_i,f_{i+1})}{\longrightarrow} r(n)(i)$ can be rewritten as: $(r_i(n_1), \ldots, r_i(n_p)) \overset{(f_i,f_{i+1})}{\longrightarrow} r_i(n)$. We conclude that each r_i is indeed a run of D. Notice that $T = T_1$ and $T' = T_{k+1}$. Furthermore, for each $i : 1 \leq i \leq k$, $T_i \times T_{i+1} \overset{r_i}{\Longrightarrow}_D F$, i.e. each pair (T_i, T_{i+1}) is accepted by D with run r_i.

Hence, we conclude that $(T, T') \in R(D^k) \subseteq R(D^+)$.

$R(H) \supseteq R(D^+)$:

Conversely, suppose that $(T, T') \in R(D^+)$.

Let k be the smallest integer such that $(T, T') \in R(D^k) \subseteq R(D^+)$.

By definition of composition, there exist structurally equivalent trees $T_1, \ldots, T_{k+1}$ with labeling functions $\lambda_1, \ldots, \lambda_{k+1}$ such that $T = T_1$, $T' = T_{k+1}$, and for each $i : 1 \le i \le k$, $(T_i, T_{i+1}) \in R(D)$. For each $T_i \times T_{i+1}$, let r_i be an accepting run of D. Finally, let r be the mapping $r(n) = r_1(n) \cdots r_k(n)$ for each node n.

Observe that for each node n with children $n_1, \ldots, n_p$, the following holds:

- $|r(n)| = |r(n_1)| = \ldots = |r(n_p)| = k$;
- for each $i : 1 \le i \le k$, $(r(n_1)(i), \ldots, r(n_p)(i)) \stackrel{(\lambda_i(n), \lambda_{i+1}(n))}{\longrightarrow} r(n)(i) \in \delta$.

Hence, there is a rule $(r(n_1), \ldots, r(n_p)) \stackrel{(\lambda_1(n), \lambda_{k+1}(n))}{\longrightarrow} r(n) \in \delta_H$, for each node n.

Thus, r is a run of H that accepts $T \times T'$. We conclude that $(T, T') \in R(H)$. $\square$

The problem with H is that it has infinitely many states. Therefore, we define an *equivalence* $\simeq$ on the states of H, and construct a new transducer where equivalent states are merged. This new transducer will hopefully only have a finite number of states.

Given an equivalence relation $\simeq$, the symbolic transducer $D_{\simeq}$ obtained by merging states of H according to $\simeq$ is defined as $(Q/\simeq, F/\simeq, \delta_{\simeq})$, where:

- $Q/\simeq$ is the set of equivalence classes of Q_H w.r.t. $\simeq$;
- $F/\simeq$ is the set of equivalence classes of F_H w.r.t. $\simeq$ (this will always be well-defined, see condition 5 of Sufficient Conditions 24, in Section 6.3);
- $\delta_{\simeq}$ contains rules of the form $(x_1, \ldots, x_n) \stackrel{f}{\longrightarrow}_{\simeq} x$ iff there are states $w_1 \in x_1, \ldots, w_n \in x_n, w \in x$ such that there is a rule $(w_1, \ldots, w_n) \stackrel{f}{\longrightarrow} w \in \delta_H$ of H.

Since H is infinite we cannot derive $D_{\simeq}$ by first computing H. Instead, we compute $D_{\simeq}$ on-the-fly, collapsing states which are equivalent according to $\simeq$. In other words, we perform the following *procedure* (which need not terminate in general).

- The procedure computes successive reflexive powers of D: $D^{\le 1}, D^{\le 2}, D^{\le 3}, \ldots$ (where $D^{\le i} = \bigcup_{n=1}^{n=i} D^n$), and collapses states[4] according to $\simeq$. We thus obtain $D_{\simeq}^{\le 1}, D_{\simeq}^{\le 2}, \ldots$

[4] The states of $D^{\le i}$ are by construction states of the history transducer.

- The procedure terminates when the relation $R(D^+)$ is accepted by $D_{\simeq}^{\leq i}$. This can be tested by checking if the language of $D_{\simeq}^{\leq i} \circ D$ is included in the language of $D_{\simeq}^{\leq i}$.

In the next section, we explain how we can make the above procedure sound, complete, and implementable.

6 Soundness, Completeness, and Computability

In this section, we describe how to derive equivalence relations on the states of the history transducer which can be used in the procedure given in Section 5. A *good* equivalence relation $\simeq$ satisfies the following two conditions:

- It is sound and complete, i.e., $R(D_{\simeq}) = R(H)$. This means that $D_{\simeq}$ characterizes the same relation as D^+.
- It is computable. This turns the procedure of Section 5 into an *implementable algorithm*, since it allows on-the-fly merging of equivalent states.

We provide a methodology for deriving such a good equivalence relations as follows:

(1) In Section 6.1, we define two simulation relations; namely a downward simulation relation $\preccurlyeq_{down}$ and an *upward simulation relation* $\preccurlyeq_{up}$.
(2) In Section 6.2, an upward and a downward simulation are put together to induce an equivalence relation $\simeq$.
(3) Next, in Section 6.3, we give sufficient conditions on the simulation relations which guarantee that the induced equivalence $\simeq$ is sound and complete.
(4) Finally, Section 6.4 deals with the computability of $\simeq$.

6.1 Downward and Upward Simulation

We start by giving the definitions.

Definition 16 [Downward Simulation]
Let $A = (Q, F, \delta)$ be a tree automaton over Σ. A binary relation $\preccurlyeq_{down}$ on the states of A is a downward simulation iff for any $n \geq 1$ and any symbol $f \in \Sigma_n$, for all states $q, q_1, \ldots, q_n, r \in Q$, the following holds:
Whenever $q \preccurlyeq_{down} r$ and $(q_1, \ldots, q_n) \xrightarrow{f} q \in \delta$, there exist states $r_1, \ldots, r_n \in Q$ such that $q_1 \preccurlyeq_{down} r_1, \ldots, q_n \preccurlyeq_{down} r_n$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$.

Definition 17 [**Upward Simulation**]
Let $A = (Q, F, \delta)$ be a tree automaton over Σ. Given a downward simulation $\preccurlyeq_{down}$, a binary relation $\preccurlyeq_{up}$ on the states of A is an upward simulation w.r.t. $\preccurlyeq_{down}$ iff for any $n \geq 1$ and any symbol $f \in \Sigma_n$, for all states $q, q_1, \ldots, q_i, \ldots, q_n, r_i \in Q$, the following holds:
Whenever $q_i \preccurlyeq_{up} r_i$ and $(q_1, \ldots, q_n) \xrightarrow{f} q \in \delta$, there exist states $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n, r \in Q$ such that $q \preccurlyeq_{up} r$ and $\forall j \neq i : q_j \preccurlyeq_{down} r_j$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$.

While the notion of a downward simulation is a straightforward extension of the word case, the notion of an upward simulation is not as obvious. This comes from the asymmetric nature of trees. If we follow the execution of a tree automaton downwards, it is easy to see that all respective children of two nodes related by simulation should continue to be related pairwise. If we now consider how a tree automaton executes when going upwards, we are confronted to the problem that the parent of the current node may have several children. The question is then how to characterize the behavior of such children. The answer lies in constraining their prefixes, i.e. using a downward simulation.

We state some elementary properties of the simulation relations

Lemma 18 *Let $A = (Q, F, \delta)$ be a tree automaton. Let $\preccurlyeq_{down}$ be a relation on the states of A which is a downward simulation. The reflexive closure and the transitive closure of $\preccurlyeq_{down}$ are both downward simulations. Furthermore, there is a unique maximal downward simulation.*

PROOF. We consider the three claims.

Reflexivity

Let $\preccurlyeq^1_{down} = \preccurlyeq_{down} \cup Id$. We show that $\preccurlyeq^1_{down}$ is also a downward simulation. Assume $(q_1, \ldots, q_n) \xrightarrow{f} q \in \delta$ and $q \preccurlyeq^1_{down} r$. We find states $r_1, \ldots, r_n \in Q$ such that $q_1 \preccurlyeq^1_{down} r_1, \ldots, q_n \preccurlyeq^1_{down} r_n$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ as follows:

- If $q = r$, then we choose $r_1 = q_1, \ldots, r_n = q_n$. Observe that since $\preccurlyeq^1_{down} \supseteq Id$, we have $q_1 \preccurlyeq^1_{down} r_1, \ldots, q_n \preccurlyeq^1_{down} r_n$. Thus, the claim holds.
- If $q \preccurlyeq_{down} r$, then we apply the hypothesis that $\preccurlyeq_{down}$ is a downward simulation, and conclude that there exist $r_1, \ldots, r_n \in Q$ such that $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ and $q_1 \preccurlyeq_{down} r_1, \ldots, q_n \preccurlyeq_{down} r_n$. From $\preccurlyeq^1_{down} \supseteq \preccurlyeq_{down}$, we conclude that the claim holds.

Transitivity

Let $\preccurlyeq^1_{down}$ be the transitive closure of $\preccurlyeq_{down}$. We show that $\preccurlyeq^1_{down}$ is also a downward simulation. Assume $(q_1, \ldots, q_n) \xrightarrow{f} q \in \delta$ and $q \preccurlyeq^1_{down} r$. We find states $r_1, \ldots, r_n \in Q$ such that $q_1 \preccurlyeq^1_{down} r_1, \ldots, q_n \preccurlyeq^1_{down} r_n$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ as follows:

- If $q \preccurlyeq_{down} r$, then the claim trivially holds.
- If there is $s \in Q$ such that $q \preccurlyeq_{down} s \preccurlyeq_{down} r$, then apply the hypothesis that $\preccurlyeq_{down}$ is a downward simulation with $q \preccurlyeq_{down} s$, and find states $s_1, \ldots, s_n \in Q$ with $(s_1, \ldots, s_n) \xrightarrow{f} s \in \delta$ and $q_1 \preccurlyeq_{down} s_1, \ldots, q_n \preccurlyeq_{down} s_n$. Now, we apply this a second step using $s \preccurlyeq_{down} r$, and find states $r_1, \ldots, r_n \in Q$ such that $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ and $s_1 \preccurlyeq_{down} r_1, \ldots, s_n \preccurlyeq_{down} r_n$. By transitivity, we get $q_1 \preccurlyeq^1_{down} r_1, \ldots, q_n \preccurlyeq^1_{down} r_n$. Thus, the claim holds.

Observe that in the second alternative above, we only treat the case of one step transitivity. Arbitrary transitivity follows by induction on the number of steps.

Uniqueness

Assume two maximal downward simulations $\preccurlyeq^1_{down}$ and $\preccurlyeq^2_{down}$. Let $\preccurlyeq_{down} = \preccurlyeq^1_{down} \cup \preccurlyeq^2_{down}$. We show that $\preccurlyeq_{down}$ is also a simulation. Assume $(q_1, \ldots, q_n) \xrightarrow{f} q \in \delta$ and $q \preccurlyeq_{down} r$. We find states $r_1, \ldots, r_n \in Q$ such that $q_1 \preccurlyeq_{down} r_1, \ldots, q_n \preccurlyeq_{down} r_n$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ as follows:

- If $q \preccurlyeq^1_{down} r$, then since $\preccurlyeq^1_{down}$ is a simulation, and $\preccurlyeq_{down} \supseteq \preccurlyeq^1_{down}$, the claim holds.
- If $q \preccurlyeq^2_{down} r$, then since $\preccurlyeq^2_{down}$ is a simulation, and $\preccurlyeq_{down} \supseteq \preccurlyeq^2_{down}$, the claim holds.

We have $\preccurlyeq_{down} \supseteq \preccurlyeq^1_{down}$ and $\preccurlyeq_{down} \supseteq \preccurlyeq^2_{down}$. Now, if we assume $\preccurlyeq^1_{down} \neq \preccurlyeq^2_{down}$, we get either $\preccurlyeq_{down} \supset \preccurlyeq^1_{down}$ or $\preccurlyeq_{down} \supset \preccurlyeq^2_{down}$. This violates the maximality of either $\preccurlyeq^1_{down}$ or $\preccurlyeq^2_{down}$. □

Lemma 19 *Let $A = (Q, F, \delta)$ be a tree automaton. Let $\preccurlyeq_{down}$ be a reflexive (transitive) downward simulation on the states of A. The reflexive (transitive) closure of an upward simulation w.r.t to $\preccurlyeq_{down}$ is also an upward simulation w.r.t $\preccurlyeq_{down}$. Furthermore there exists a unique maximal upward simulation w.r.t. any downward simulation.*

PROOF. We consider the three claims.

Reflexivity

Let $\preccurlyeq_{up}^1 = \preccurlyeq_{up} \cup Id$. We show that $\preccurlyeq_{up}^1$ is also an upward simulation. Assume $(q_1, \ldots, q_i, \ldots, q_n) \xrightarrow{f} q \in \delta$ and $q_i \preccurlyeq_{up}^1 r_i$. We find states $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n, r \in Q$ such that $q \preccurlyeq_{up}^1 r$ and $\forall j \neq i : q_j \preccurlyeq_{down} r_j$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ as follows:

- If $q_i \preccurlyeq_{up} r_i$, then since $\preccurlyeq_{up}$ is an upward simulation, and $\preccurlyeq_{up}^1 \supseteq \preccurlyeq_{up}$, the claim trivially holds.
- If $q_i = r_i$, then we choose $r = q$ and $r_j = q_j$ for each $j \neq i$. By reflexivity of $\preccurlyeq_{down}$, we have $q_j \preccurlyeq_{down} r_j$ for each $j \neq i$. Since $\preccurlyeq_{up}^1 \supset Id$, we also have $q \preccurlyeq_{up}^1 r$. Hence, the claim holds.

Transitivity

Let $\preccurlyeq_{up}^1$ be the transitive closure of $\preccurlyeq_{up}$. We show that $\preccurlyeq_{up}^1$ is also an upward simulation. Assume $(q_1, \ldots, q_i, \ldots, q_n) \xrightarrow{f} q \in \delta$ and $q_i \preccurlyeq_{up}^1 r_i$. We find states $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n, r \in Q$ such that $q \preccurlyeq_{up}^1 r$ and $\forall j \neq i : q_j \preccurlyeq_{down} r_j$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ as follows:

- If $q_i \preccurlyeq_{up} r_i$, then since $\preccurlyeq_{up}$ is an upward simulation, and $\preccurlyeq_{up}^1 \supseteq \preccurlyeq_{up}$, the claim trivially holds.
- If there is s_i with $q_i \preccurlyeq_{up} s_i \preccurlyeq_{up} r_i$, then since $q_i \preccurlyeq_{up} s_i$, we apply the hypothesis that $\preccurlyeq_{up}$ is an upward simulation. We get states $s, s_1, \ldots, s_n \in Q$ with $(s_1, \ldots, s_i, \ldots, s_n) \xrightarrow{f} s \in \delta$ and $q \preccurlyeq_{up} s$ and for each $j \neq i$, $q_j \preccurlyeq_{down} s_j$. With $s_i \preccurlyeq_{up} r_i$, we use simulation a second time, and get states $r, r_1, \ldots, r_n \in Q$ with $(r_1, \ldots, r_i, \ldots, r_n) \xrightarrow{f} r \in \delta$ and $s \preccurlyeq_{up} r$ and for each $j \neq i$, $s_j \preccurlyeq_{down} r_j$. By transitivity of $\preccurlyeq_{down}$, we get for each $j \neq i$, $q_j \preccurlyeq_{down} r_j$. By transitivity of $\preccurlyeq_{up}^1$, we also get $q \preccurlyeq_{up}^1 r$. Hence, the claim holds.

Observe that in the second alternative above, we only treat the case of one step transitivity. Arbitrary transitivity follows by induction on the number of steps.

Uniqueness

Assume two maximal upward simulations $\preccurlyeq_{up}^1$ and $\preccurlyeq_{up}^2$. Let $\preccurlyeq_{up} = \preccurlyeq_{up}^1 \cup \preccurlyeq_{up}^2$. We show that $\preccurlyeq_{up}$ is also a simulation. Assume $(q_1, \ldots, q_i, \ldots, q_n) \xrightarrow{f} q \in \delta$ and $q_i \preccurlyeq_{up} r_i$. We find states $r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n, r \in Q$ such that $q \preccurlyeq_{up} r$ and $\forall j \neq i : q_j \preccurlyeq_{down} r_j$ and $(r_1, \ldots, r_n) \xrightarrow{f} r \in \delta$ as follows:

- If $q_i \preccurlyeq_{up}^1 r_i$, then since $\preccurlyeq_{up}^1$ is a simulation, and $\preccurlyeq_{up} \supseteq \preccurlyeq_{up}^1$, the claim holds.
- If $q_i \preccurlyeq_{up}^2 r_i$, then since $\preccurlyeq_{up}^2$ is a simulation, and $\preccurlyeq_{up} \supseteq \preccurlyeq_{up}^2$, the claim holds.

We have $\preccurlyeq_{up} \supseteq \preccurlyeq^1_{up}$ and $\preccurlyeq_{up} \supseteq \preccurlyeq^2_{up}$. Now, if we assume $\preccurlyeq^1_{up} \neq \preccurlyeq^2_{up}$, we get either $\preccurlyeq_{up} \supset \preccurlyeq^1_{up}$ or $\preccurlyeq_{up} \supset \preccurlyeq^2_{up}$. This violates the maximality of either $\preccurlyeq^1_{up}$ or $\preccurlyeq^2_{up}$. $\square$

Observe that both for downward simulations, and upward simulations, maximality implies transitivity and reflexivity.

6.2 Induced Equivalence Relation

We now define an equivalence relation derived from two binary relations.

Definition 20 [**Independence**]
Two binary relations $\preceq_1$ and $\preceq_2$ are said to be independent *iff whenever $q \preceq_1 r$ and $q \preceq_2 r'$, there exists s such that $r \preceq_2 s$ and $r' \preceq_1 s$.*

Definition 21 [**Induced Relation**]
The relation $\simeq$ induced *by two binary relations $\preceq_1$ and $\preceq_2$ is defined as:*

$$\preceq_1 \circ \preceq_2^{-1} \cap \preceq_2 \circ \preceq_1^{-1} .$$

The following Lemma gives sufficient conditions for two relations to induce an equivalence relation.

Lemma 22 *Let $\preceq_1$ and $\preceq_2$ be two binary relations. If $\preceq_1$ and $\preceq_2$ are reflexive, transitive, and independent, then their induced relation $\simeq$ is an* equivalence relation.

PROOF.

(1) If $\preceq_1$ and $\preceq_2$ are reflexive, then for any q we have $q \preceq_1 q \preceq_2^{-1} q$ and $q \preceq_2 q \preceq_1^{-1} q$. Thus, we have $q \simeq q$.
(2) If $q \simeq r$, then $q \preceq_1 \circ \preceq_2^{-1} r$ and $q \preceq_2 \circ \preceq_1^{-1} r$. We can rewrite this $r \preceq_1 \circ \preceq_2^{-1} q$ and $r \preceq_2 \circ \preceq_1^{-1} q$. Hence, $r \simeq q$.
(3) Assume $q \simeq r \simeq s$. Then by definition of $\simeq$, we can find t', t'' such that $q \preceq_1 t' \preceq_2^{-1} r$ and $r \preceq_1 t'' \preceq_2^{-1} s$. Since $\preceq_1$ and $\preceq_2$ are independent, there is t such that $t' \preceq_1 t$ and $t'' \preceq_2 t$. By transitivity, we get $q \preceq_1 t \preceq_2^{-1} s$. Hence, $q \preceq_1 \circ \preceq_2^{-1} s$. Similarly, we can get $q \preceq_2 \circ \preceq_1^{-1} s$, and finally conclude that $q \simeq s$

$\square$

To conclude, we state a property of $\simeq$ which follows from independence.

Lemma 23 *Let $\preceq_1$ and $\preceq_2$ be both reflexive and transitive, and $\simeq$ be their induced relation. Furthermore, $\preceq_1$ and $\preceq_2$ are independent. Whenever $x \simeq y$ and $x \preceq_1 z$, there exists t such that $y \preceq_1 t$ and $z \preceq_2 t$.*

PROOF. Assume $x \simeq y$ and $x \preceq_1 z$. By definition of $\simeq$, we know that there is u with $x \preceq_2 u$ and $y \preceq_1 u$. We apply the definition of independence to x, u, z, and conclude that there is a state t such that $z \preceq_2 t$ and $u \preceq_1 t$. By transitivity of $\preceq_1$, we have $y \preceq_1 t$. □

6.3 Sufficient Conditions for Soundness and Completeness

We give sufficient conditions for two simulation relations to induce a sound and complete equivalence relation on states of a tree automaton.

We assume a tree automaton $A = (Q, F, \delta)$. We now define a relation $\simeq$ on the states of A, induced by the two relations $\preceq$ and $\preccurlyeq_{down}$ both on the states of A, satisfying the following sufficient conditions:

Sufficient Conditions 24 *(1) $\preccurlyeq_{down}$ is a downward simulation;*
(2) $\preceq$ is a reflexive and transitive relation included in $\preccurlyeq_{up}$ which is an upward simulation w.r.t. $\preccurlyeq_{down}$;
(3) $\preccurlyeq_{down}$ and $\preceq$ are independent;
(4) whenever $x \in F$ and $x \preccurlyeq_{up} y$, then $y \in F$;
(5) F is a union of equivalence classes w.r.t. $\simeq$;
(6) whenever $\xrightarrow{f} x$ and $x \preccurlyeq_{down} y$, then $\xrightarrow{f} y$.

□

We first obtain the following Lemma which shows that if the simulations satisfy the above Sufficient Conditions, then the induced relation is indeed an equivalence.

Lemma 25 *Let $A = (Q, F, \delta)$ be a tree automaton. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ on the states of A, which satisfy Sufficient Conditions 24, as well as their induced relation $\simeq$. Then $\simeq$ is an equivalence relation on states of A.*

PROOF. The claim holds since Conditions 1 through 3 of Sufficient Conditions 24 above imply directly that $\preccurlyeq_{down}$ and $\preceq$ satisfy the hypothesis needed by Lemma 22. □

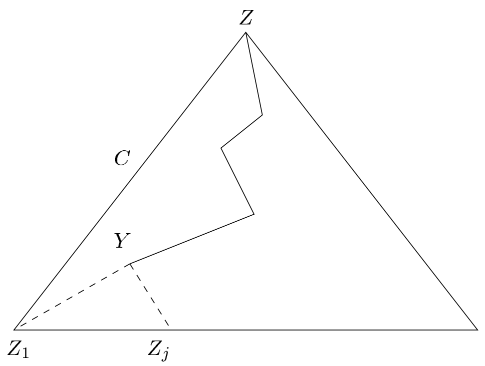

Fig. 3. Lemma 26: a path within context C.

We then prove that an equivalence relation satisfying Sufficient Conditions 24 is sound. This result is stated in Theorem 27. Lemma 26 is an intermediate result.

We first show that the tree automaton $A_{\simeq}$ has the same traces as A.

Lemma 26 *Let $A = (Q, F, \delta)$ be a tree automaton. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ on the states of A, satisfying Sufficient Conditions 24, and let $\simeq$ be their induced relation. Let $A_{\simeq} = (Q/\simeq, F/\simeq, \delta_{\simeq})$ be the automaton obtained by merging the states of A according to $\simeq$. For any states $Z_1, \ldots, Z_k, Z$ of $A_{\simeq}$ and context C,*
if $C[Z_1, \ldots, Z_k] \Longrightarrow_{\simeq} Z$, then there exist states $z_1, \ldots, z_k, z$ and states $t_1, \ldots, t_k, t$ of A such that $C[t_1, \ldots, t_k] \Longrightarrow t$ and $z_1 \in Z_1, \ldots, z_k \in Z_k, z \in Z$ and $z_1 \preccurlyeq_{down} t_1, \ldots, z_k \preccurlyeq_{down} t_k, z \preccurlyeq_{up} t$.

PROOF. The claim is shown by induction on the structure of C.

Base case: C contains only a hole. We choose a $z \in Z$. By reflexivity of $\preccurlyeq_{down}$ and $\preccurlyeq_{up}$, the claim obviously holds.

Induction case: C is not just a hole. Consider a run r of $A_{\simeq}$ on $C = (S_C, \lambda_C)$ satisfying
$C[Z_1, \ldots, Z_k] \Longrightarrow_{\simeq} Z$. Let $n_1, \ldots, n_j$ be the left-most leaves of C with a common parent. Let n be the parent of $n_1, \ldots, n_j$. Note that $Z_1 = r(n_1), \ldots, Z_j = r(n_j)$. Let $Y = r(n)$. This situation is illustrated in Figure 3.

We let C' be the context C, with the leaves $n_1, \ldots, n_j$ deleted. In other words $C' = (S'_C, \lambda'_C)$ where $S'_C = S_C - \{n_1, \ldots, n_j\}$, $\lambda'_C(n') = \lambda_C(n')$ if $n' \in S_C - \{n, n_1, \ldots, n_j\}$, and $\lambda'_C(n) = \Box$. Since C' is smaller than C, we can apply the induction hypothesis. Let $u, z_{j+1}, \ldots, z_k, y$ and $v, t'_{j+1}, \ldots, t'_k, t'$ be states of A such that $C'\left[v, t'_{j+1}, \ldots, t'_k\right] \Longrightarrow t'$ and $u \in Y, z_{j+1} \in Z_{j+1}, \ldots, z_k \in Z_k, y \in Z$ and $u \preccurlyeq_{down} v, z_{j+1} \preccurlyeq_{down} t'_{j+1}, \ldots, z_k \preccurlyeq_{down} t'_k, y \preccurlyeq_{up} t'$.

By definition of $A_{\simeq}$, there are states $z \in Y, z_1 \in Z_1, \ldots, z_j \in Z_j$ such that $(z_1, \ldots, z_j) \xrightarrow{f} z$ for some f.

We now use Lemma 23 with premise $u \simeq z$ and $u \preccurlyeq_{down} v$. We thus find state w such that $z \preccurlyeq_{down} w$ and $v \preceq w$. Note that this implies $v \preccurlyeq_{up} w$.

By definition of a downward simulation, and premises $z \preccurlyeq_{down} w$ and $(z_1, \ldots, z_j) \xrightarrow{f} z$, we find states $t_1, \ldots, t_j$ with $z_1 \preccurlyeq_{down} t_1, \ldots, z_j \preccurlyeq_{down} t_j$ and $(t_1, \ldots, t_j) \xrightarrow{f} w$.

By definition of an upward simulation and premises $v \preccurlyeq_{up} w$ and $C' \left[v, t'_{j+1}, \ldots, t'_k\right] \Longrightarrow t'$, we find states $t, t_{j+1}, \ldots, t_k$ with $t' \preccurlyeq_{up} t$, $t'_{j+1} \preccurlyeq_{down} t_{j+1}, \ldots, t'_k \preccurlyeq_{down} t_k$ and $C' [w, t_{j+1}, \ldots, t_k] \Longrightarrow t$.

The claim thus holds. □

We are now ready to prove the soundness of merging with $\simeq$.

Theorem 27 *Let $A = (Q, F, \delta)$ be a tree automaton. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ on the states of A, satisfying Sufficient Conditions 24, and let $\simeq$ be their induced relation. Let $A_{\simeq} = (Q/\simeq, F/\simeq, \delta_{\simeq})$ be the automaton obtained by merging the states of A according to $\simeq$. Then, $L(A_{\simeq}) = L(A)$.*

PROOF. Since $A_{\simeq}$ is a collapsed version of A, we trivially have $L(A_{\simeq}) \supseteq L(A)$.

Conversely, let T be a tree accepted by $A_{\simeq}$. We construct a context C by replacing all leaves in T by holes. We apply the construction of Lemma 26 to context C. We now have a run of A on C. Conditions 4 and 5 of Sufficient Conditions 24 ensure that this run is accepting. Condition 6 of Sufficient Conditions 24 ensures that we can extend the run on C to a run on T. □

Theorem 27 can be used to relate the languages of H and $D_{\simeq}$ (recall that $D_{\simeq}$ was defined in Section 5).

We are now ready to prove the soundness and the completeness of the procedure of Section 5 (assuming a computable equivalence relation $\simeq$).

Theorem 28 *Consider a transducer $D = (Q, F, \delta)$ and its associated history transducer $H = (Q_H, F_H, \delta_H)$. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ on the states of H which satisfy the hypothesis of Theorem 27. Let $\simeq$ be their induced equivalence relation. If the procedure of Section 5 terminates at step i, then the transducer $D_{\simeq}^{\leq i}$ accepts the same relation as $D_{\simeq}$.*

PROOF. We can easily see that by construction, $D_{\simeq}^{\leq i}$ is a sub-automaton of $D_{\simeq}$.

Conversely, let (T_1, T_2) be a pair accepted by $D_{\simeq}$. We use Theorem 27, and let r be the corresponding run in H. Let $w_0, w_1, \ldots, w_n$ be the states in r. Let k be the length $k = |w_0| = |w_1| = \ldots = |w_n|$. Note that (T_1, T_2) is accepted by $D^{\leq k}$.

- If $k \leq i$, then by construction, states $[w_0], [w_1], \ldots, [w_n]$ are in $D_{\simeq}^{\leq i}$, and there is an accepting run in $D_{\simeq}^{\leq i}$ for the pair (T_1, T_2).
- If $k > i$, then we let T be such that (T_1, T) is recognized by $D^{\leq i}$ and (T, T_2) is recognized by $D^{\leq k-i}$. By the reasoning above, we know that (T_1, T) is recognized by $D_{\simeq}^{\leq i}$.

  Hence, we can write $(T_1, T_2) \in R(D_{\simeq}^{\leq i} \circ D^{\leq k-i})$. Using ($k-i$ times) the termination condition $R(D_{\simeq}^{\leq i} \circ D) \subseteq R(D_{\simeq}^{\leq i})$, we get that (T_1, T_2) is recognized by $D_{\simeq}^{\leq i}$.

□

6.4 Sufficient Condition for Computability

The next step is to give conditions on the simulations which ensure that the induced equivalence relation is computable.

Definition 29 [**Effective relation**]
A relation $\preceq$ is said to be effective if the image of a regular set w.r.t. $\preceq$ and w.r.t. $\preceq^{-1}$ is regular and computable.

Effective relations induce an equivalence relation which is also computable.

Theorem 30 *Let $D = (Q, F, \delta)$ be a transducer and $H = (Q_H, F_H, \delta_H)$ its associated history transducer. Let $\preceq_1$ and $\preceq_2$ be relations on the states of H that are both reflexive, transitive, effective and independent. Let $\simeq$ be their induced equivalence. Then for any state $w \in Q_H$, we can compute its equivalence class $[w]$ w.r.t. $\simeq$.*

PROOF. The claim follows by definition of $\simeq$, and effectiveness [5] of $\preceq_1$ and $\preceq_2$. □

Using relations on the states of H that satisfy the premises of Theorem 30 naturally turns the procedure of Section 5 into an *algorithm*. If the relations

[5] A state w of the history transducer is a word. The set $\{w\}$ is regular.

used also satisfy the premises of Theorem 27, then the on-the-fly algorithm of Section 5 computes (when it terminates) the transitive closure of a tree transducer. The next step is to provide a concrete example of such relations. Because we are not able to compute the infinite representation of H, the relations will be directly computed from the powers of D provided by the on-the-fly algorithm.

7 Good Equivalence Relation

In this section, we provide concrete relations satisfying Theorem 27 and Theorem 30. We first introduce prefix- and suffix-copying states.

Definition 31 [**Prefix-Copying State**]
Given a transducer $D = (Q, F, \delta)$, and a state $q \in Q$, we say that q is a prefix-copying state if for any tree $T = (S, \lambda) \in \mathrm{pref}(q)$, then for any node $n \in S$, $\lambda(n) = (f, f)$ for some symbol $f \in \Sigma$.

Definition 32 [**Suffix-Copying State**]
Given a transducer $D = (Q, F, \delta)$, and a state $q \in Q$, we say that q is a suffix-copying state if for any context $C = (S_C, \lambda_C) \in \mathrm{suff}(q)$, then for any node $n \in S_C$ with $\lambda_C(n) \neq \square$, we have $\lambda_C(n) = (f, f)$ for some symbol $f \in \Sigma$.

We let Q_{pref} (resp. Q_{suff}) denote the set of prefix-copying states (resp. the set of suffix-copying states) of D and we assume that $Q_{pref} \cap Q_{suff} = \emptyset$. We let $Q_N = Q - Q_{pref} \cup Q_{suff}$.

We now define relations by the means of rewriting rules on the states of the history transducer.

Definition 33 [**Generated Relation**]
Let $D = (Q, F, \delta)$ be a tree transducer, and $H = (Q_H, F_H, \delta_H)$ its associated history transducer. Given a set $S \subseteq Q_H \times Q_H = Q^* \times Q^*$, we define the relation $\mapsto$ generated *by S to be the smallest reflexive and transitive relation such that $\mapsto$ contains S, and $\mapsto$ is a congruence w.r.t. concatenation (i.e. if $x \mapsto y$, then for any w_1, w_2, we have $w_1 \cdot x \cdot w_2 \mapsto w_1 \cdot y \cdot w_2$).*

Next, we find relations $\preceq$ and $\preccurlyeq_{down}$ on the states of H that satisfy the sufficient conditions for computability (Theorem 30) and conditions for exactness of abstraction (Theorem 27).

Definition 34 [**Simulation Relations**]
Let $D = (Q, F, \delta)$ be a tree transducer, and $H = (Q_H, F_H, \delta_H)$ its associated history transducer. Let Q_{pref} (resp. Q_{suff}) denote the set of prefix-copying

states (resp. the set of suffix-copying states) of D, assuming $Q_{pref} \cap Q_{suff} = \emptyset$. We let $Q_N = Q - Q_{pref} \cup Q_{suff}$.

- *We define $\preccurlyeq_{down}$ to be the downward simulation generated by all pairs of the form $(q_{pref} \cdot q_{pref}, q_{pref})$ and $(q_{pref}, q_{pref} \cdot q_{pref})$, where $q_{pref} \in Q_{pref}$.*
- *Let $\preccurlyeq^1_{up}$ be the maximal upward simulation computed on $D \cup D^2$. We define $\preceq$ to be the relation generated by the maximal set $S \subseteq \preccurlyeq^1_{up}$ such that*
  · $(q_{suff} \cdot q_{suff}, q_{suff}) \in S$ *iff* $(q_{suff}, q_{suff} \cdot q_{suff}) \in S$,
  · $(q \cdot q_{suff}, q) \in S$ *iff* $(q, q \cdot q_{suff}) \in S$,
  · $(q_{suff} \cdot q, q) \in S$ *iff* $(q, q_{suff} \cdot q) \in S$,
  where $q_{suff} \in Q_{suff}$, and $q \in Q_N$.

Algorithms for computing the simulations needed for Definition 34 can be found in Appendix B. These algorithms are adapted from those provided by Henzinger *et al.* [16] for the case of finite words.

Let us state that the simulations of Definition 34 satisfy the Sufficient Conditions 24, and hence satisfy the premises of Theorem 30 and Theorem 27.

Lemma 35 *Let $\preccurlyeq_{down}$ as defined in Definition 34. The following properties of $\preccurlyeq_{down}$ hold:*

(1) $\preccurlyeq_{down}$ is a downward simulation;
(2) $\preccurlyeq_{down}$ is effective.

PROOF.

(1) Let $x \cdot q_{pref} \cdot y$ be a state of H. Any transition rule leading to that state will be of the form:

$$\left(x^1 \cdot q^1_{pref} \cdot y^1, \ldots, x^n \cdot q^n_{pref} \cdot y^n\right) \xrightarrow{f}_H x \cdot q_{pref} \cdot y$$

Suppose $x \cdot q_{pref} \cdot y \preccurlyeq_{down} z$. Then by definition, we know that z is of the form $x \cdot q_{pref} \cdot q_{pref} \cdot y$. Observe that for each $i : 1 \leq i \leq n$, we have $x^i \cdot q^i_{pref} \cdot y^i \preccurlyeq_{down} x^i \cdot q^i_{pref} \cdot q^i_{pref} \cdot y^i$. We also have a rule:

$$\left(x^1 \cdot q^1_{pref} \cdot q^1_{pref} \cdot y^1, \ldots, x^n \cdot q^n_{pref} \cdot q^n_{pref} \cdot y^n\right) \xrightarrow{f}_H x \cdot q_{pref} \cdot q_{pref} \cdot y$$

Conversely, we consider the state $x \cdot q_{pref} \cdot q_{pref} \cdot y$ of H. We notice that since D is deterministic, it follows that a state of form $q^1_{pref} \cdot q^2_{pref}$ is not reachable in H unless $q^1_{pref} = q^2_{pref}$. We can thus ignore states in which $q^1_{pref} \neq q^2_{pref}$. Then, any transition rule leading to state $x \cdot q_{pref} \cdot q_{pref} \cdot y$ will be of the form:

$$\left(x^1 \cdot q^1_{pref} \cdot q^1_{pref} \cdot y^1, \ldots, x^n \cdot q^n_{pref} \cdot q^n_{pref} \cdot y^n\right) \xrightarrow{f}_H x \cdot q_{pref} \cdot q_{pref} \cdot y$$

Suppose $x \cdot q_{pref} \cdot q_{pref} \cdot y \preccurlyeq_{down} z$. Then we have z of the form $x \cdot q_{pref} \cdot y$. Observe that we also have for each i: $x^i \cdot q^i_{pref} \cdot q^i_{pref} \cdot y' \preccurlyeq_{down} x^i \cdot q^i_{pref} \cdot y^i$. We also have a rule:

$$\left(x^1 \cdot q^1_{pref} \cdot y^1, \ldots, x^n \cdot q^n_{pref} \cdot y^n\right) \xrightarrow{f}_H x \cdot q_{pref} \cdot y$$

(2) If we consider a regular set of states of H given by a word automaton, then its image w.r.t. $\preccurlyeq_{down}$ or $\preccurlyeq_{down}^{-1}$ can be expressed by adding edges to this automaton: for each transition $x \xrightarrow{q_{pref}} y$, add an edge $y \xrightarrow{q_{pref}} y$; similarly, for each consecutive edges $x \xrightarrow{q_{pref}} y$ and $y \xrightarrow{q_{pref}} z$, add an edge $x \xrightarrow{q_{pref}} z$.

□

Lemma 36 *Let $\preceq$ as defined in Definition 34. The following properties of $\preceq$ hold:*

(1) $\preceq$ is included in an upward simulation;
(2) $\preceq$ is effective.

PROOF.

(1) We know that $\preccurlyeq^1_{up}$ is an upward simulation. If we let S be the relation generated by $\preccurlyeq^1_{up}$, then S is also an upward simulation. Furthermore, we have $\preceq \subseteq S$.
(2) If we consider a regular set of states of H given by a word automaton, then its image w.r.t. $\preceq$ or $\preceq^{-1}$ can be expressed by adding edges to this automaton:
- for each transition $x \xrightarrow{q_{suff}} y$, add an edge $y \xrightarrow{q_{suff}} y$; similarly, for each consecutive edges $x \xrightarrow{q_{suff}} y$ and $y \xrightarrow{q_{suff}} z$, add an edge $x \xrightarrow{q_{suff}} z$.
- if there is a pair $(q \cdot q_{suff}, q) \in S$ in Definition 34, then for each transition $x \xrightarrow{q} y$, we add an edge $y \xrightarrow{q_{suff}} y$; similarly, for two consecutive edges $x \xrightarrow{q} y$ and $y \xrightarrow{q_{suff}} z$, add an edge $x \xrightarrow{q} z$.
- if there is a pair $(q_{suff} \cdot q, q) \in S$ in Definition 34, then for each transition $x \xrightarrow{q} y$, we add an edge $x \xrightarrow{q_{suff}} x$; similarly, for two consecutive edges $x \xrightarrow{q_{suff}} y$ and $y \xrightarrow{q} z$, add an edge $x \xrightarrow{q} z$.

□

We now state that $\preceq$ and $\preccurlyeq_{down}$ are independent.

Lemma 37 *Let $\preccurlyeq_{down}$ and $\preceq$ as defined in Definition 34. $\preceq$ and $\preccurlyeq_{down}$ are independent.*

PROOF. Assume $x \preceq y$ and $x \preccurlyeq_{down} z$. Then $x = x_1 \cdot q_{pref} \cdot x_2$ and $z = x_1 \cdot z' \cdot x_2$, with $z' \in \{\epsilon, q_{pref} \cdot q_{pref}\}$. Since the left-hand side of each pair generating $\preceq$ does not contain any prefix-copying state, we conclude that $y = y_1 \cdot q_{pref} \cdot y_2$, where either $x_1 \preceq y_1$ and $x_2 = y_2$, or $x_1 = y_1$ and $x_2 \preceq y_2$. In either case, we have $z = x_1 \cdot z' \cdot x_2 \preceq y_1 \cdot z' \cdot y_2$. Furthermore, we also have $y = y_1 \cdot q_{pref} \cdot y_2 \preccurlyeq_{down} y_1 \cdot z' \cdot y_2$.

We have shown independence of *single steps* of $\preceq$ and $\preccurlyeq_{down}$. This is sufficient for proving that independence also holds for the transitive closure w.r.t. concatenation. Hence the claim holds. $\square$

The properties proved above are enough to ensure partial soundness (as stated in Lemma 26). We now state the remaining properties needed to comply with Theorem 27.

Lemma 38 *Let $\preccurlyeq_{down}$ and $\preceq$ as defined in Definition 34. The following holds:*

- *whenever $x \in F_H$ and $x \preccurlyeq_{up} y$, then $y \in F_H$;*
- *F_H is a union of equivalence classes w.r.t. $\simeq$;*
- *whenever $\xrightarrow{f} x$ and $x \preccurlyeq_{down} y$, then $\xrightarrow{f} y$;*

PROOF. We observe that all states in F are either in Q_{suff} or in Q_N. Therefore, the first and second claim hold.

The third claim holds since $x \preccurlyeq_{down} y$ only involves prefix-copying states. For a prefix-copying state q_{pref}, an arity 0 rule will be of the form $\xrightarrow{(f,f)} q_{pref}$, which means that the claim holds. $\square$

We conclude that $\preceq$ and $\preccurlyeq_{down}$ satisfy the hypothesis of Theorem 27 and Theorem 30 and can thus be used by the on-the-fly procedure presented in Section 5.

Consider the relations $\preceq$ and $\preccurlyeq_{down}$ of Definition 34. They induce an equivalence relation $\simeq$ used to merge states of the history transducer H. To get an intuition of the effect of merging states according to $\simeq$, we look at states considered equivalent:

- For any prefix-copying state $q_{pref} \in Q$, and any words x, y, all states in the set $x \cdot q_{pref}^{+} \cdot y$ will be equivalent.
- For any suffix-copying state $q_{suff} \in Q$, and any words x, y, all states in the set $x \cdot q_{suff}^{+} \cdot y$ will be equivalent.
- For any states q_{suff}, q such that $(q \cdot q_{suff}, q) \in S$ according to Definition 34, and for any words x, y, all states in the set $x \cdot q \cdot q_{suff}^{*} \cdot y$ will be equivalent.

- For any states q_{suff}, q such that $(q_{suff} \cdot q, q) \in S$ according to Definition 34, and for any words x, y, all states in the set $x \cdot q_{suff}^* \cdot q \cdot y$ will be equivalent.

8 Computing Reachable Configurations

In this section we describe the modifications needed to compute $R(D^+)(\phi_I)$ without computing D^+. For checking safety properties, such a computation is sufficient (see [17]). Computing $R(D^+)(\phi_I)$ rather than D^+ can be done by slightly modifying the definition of the history transducer associated with D^+.

Let $D = (Q, F, \delta)$. Assume that we have constructed a tree automaton $A_{\phi_I} = (Q_{\phi_I}, F_{\phi_I}, \delta_{\phi_I})$ for ϕ_I. Then, we define our new history transducer to be $H(\phi_I) = (Q_H, F_H, \delta_H)$, where

- $Q_H = Q_{\phi_I} \times Q^+$,
- $F_H = F_{\phi_I} \times F^+$,
- and δ_H contains all rules of the form

$$(w_1, \ldots, w_p) \xrightarrow{(f,f')} w$$

such that there is $k \geq 1$ where the following two conditions are satisfied
(1) $|w_1| = \cdots = |w_p| = |w| = k + 1$;
(2) there are $f_1, f_2, \ldots, f_{k+1}$, with $f = f_1$, $f' = f_{k+1}$, and
$(w_1(1) \ldots, w_p(1)) \xrightarrow{f_1} w(1) \in \delta_{\phi_I}$, and
$(w_1(i+1) \ldots, w_p(i+1)) \xrightarrow{(f_i, f_{i+1})} w(i+1) \in \delta$, for each $i : 1 \leq i \leq k$.

Intuitively, we just add a composition with the automaton representing the set of initial configurations, so that we effectively compute $(\bigcup_{i=1}^{\infty} (T(\Sigma) \times A_{\phi_I}) \circ D^i) |_1$.

Computing $R(D^+)(\phi_I)$ is often less expensive than computing D^+ because it only considers reachable sets of states. Moreover, as for the word case, there exist situations for which $R(D^+)(\phi_I)$ is regular while D^+ is not. We have an example for which our technique can compute $R(D^+)(\phi_I)$ but cannot compute D^+.

9 Experimental Results

The techniques presented in this paper have been applied on several case studies using a prototype implementation that relies in part on the regular model checking tool (see [18]). For each example, we compute the set of reachable states, as well as the transitive closure of the transition relation. In Table 1, we report the results. For each automaton, we give its size in terms of states (column labeled *st*), and of transitions (column labeled *tr*). The columns labelled *max* indicate the size of the largest automaton encountered while computing the result of the precedent column.

Note that since the protocols are all parameterized with respect to the number of participants, verification of such protocols has to take into account all possible instances (number of participants, how these participants connect to each other, etc.). Below is a short description of each example. Details of the encoding of the protocols in the tree regular model checking framework can be found in Appendix A.

Example	$\lvert R\rvert$		$\lvert\phi_I\rvert$		$\lvert R^+\rvert$		max	$\lvert R^+(\phi_I)\rvert$		max
	st	tr	st	tr	st	tr	st	st	tr	st
Simple Token Protocol	3	8	2	5	4	16	15	3	8	17
Two-Way Token Protocol	4	12	2	6	6	31	28	3	8	26
Percolate Protocol	4	13	2	3	6	73	40	3	6	21
Tree-arbiter Protocol	8	59	2	3	-	-	-	10	109	246
Leader Election Protocol	6	38	2	6	9	138	105	10	136	150

Table 1
Experimental results.

Simple Token Protocol See Example 13 of Section 4 for the description and the encoding.

Two-Way Token Protocol This example is an extension of the *Simple Token Protocol* above. The difference is that the token can move both upwards and downwards, in contrast to the Simple Token Protocol in which the token only moves upwards.

Percolate Protocol This protocol simulates the way results propagate in a set of `OR` gates organized in a tree. At the leaves, we have a series of binary inputs. The nodes of the tree act as the `OR` gates. At first, every gate has its output set to *unknown*. At each step of the protocol, a gate for which each input is determined can set its output to a definite value. The process iterates until the root gate has a definite value.

Tree-arbiter Protocol (See e.g. [8]) The tree arbiter is an asynchronous circuit that solves the mutual exclusion problem by building a tree of arbiter cells. The circuit works by performing elimination rounds: an arbiter cell arbitrates between its two children. The leaves of the tree are processors which may want to access asynchronously a shared resource. The n processors at the lowest level are arbitrated by $\frac{n}{2}$ cells. The winners of that level are arbitrated by the next level, and so forth. If both children of a cell are requesting the resource, then the cell choses either of them non-deterministically. The requests propagate upwards until the root is reached. The root cell grants the resource to at most one child, and the grant propagates downward to one of the processors. When the processor is finished with the resource, it sends a release request that propagates upwards, until it encounters a cell that is aware of a request. That cell can either grant the resource to one of the requesting nodes, or continue to propagate the release signal.

Leader Election Protocol A set of processes, denoted by the leaves, want to elect a leader. Each of them decides first whether to be a candidate or not. The election process proceeds in two phases. The first phase consists of the internal nodes polling their children nodes to see if at least one of them is candidate. In such a case, the internal node becomes a candidate as well. The second phase is the actual election procedure. The root chooses (elects) one candidate non-deterministically among its children. An internal node that has been elected, elects in turn one of its children that declared itself candidate.

In our previous work [9], we were able to handle the first three protocols of the table. However, for those protocols, we were only able to compute the transitive closure for individual *actions* representing one class of statements in the protocol, sometimes with manual intervention. Here we compute automatically the transitive closure of the tree transducers representing the *entire* transition relations of the protocols.

In order to compute the set of reachable states, we have used the technique presented in Section 8. In [9], the reachability computation was done by first computing the transitive closure for each individual action, and then applying a classical forward reachability algorithm using these results. However, such an approach requires manual intervention: to make the reachability analysis terminate, it is often necessary to combine actions in a certain order, or even to accelerate combinations of individual actions.

Observe that we are not able to compute the transitive closure of the transition relation of the tree-arbiter protocol (it is not known whether the closure is even regular). However, we are still able to compute transitive closure of individual actions for this protocol as well as the reachable set of states with the technique of Section 8.

10 Conclusions and Future Work

In this paper, we have presented a technique for computing the transitive closure of a tree transducer. The technique is based on the definition and the computation of a good equivalence relation which is used to collapse the states of the transitive closure of the tree transducer. Our technique has been implemented and successfully tested on a number of protocols, several of which are beyond the capabilities of existing tree regular model checking techniques.

The restriction to structure-preserving tree transducers might be seen as a weakness of our approach. However, structure-preserving tree transducers can model the relation of many interesting parametrized network protocols. In the future, we plan to investigate the case of non structure-preserving tree transducers. One possible solution would be to use *padding* to simulate a structure-preserving behavior. The technique of padding works by adding extra symbols to the alphabet to denote positions in the tree that are empty, and can be ignored. Using such symbols, transducer rules which change the structure of a tree can be rewritten as structure preserving rules. This would allow us to extend our method to work on such systems as Process Rewrite Systems (PRS). PRS are useful when modeling systems with a dynamic behavior [14,19].

It would also be interesting to see if one can extend our simulations, as well as the algorithms for computing them, in order to efficiently implement the technique presented in [10]. It would also be of interest to combine our simulation relations with other regular model checking techniques such as abstraction [4,20], or learning [21,22].

Finally, we intend to extend our framework to check for liveness properties on tree-like architecture systems as it is done for linear topologies in [23,24].

References

[1] A. Bouajjani, B. Jonsson, M. Nilsson, T. Touili, Regular model checking, in: Emerson, Sistla (Eds.), Proc. 12^{th} Int. Conf. on Computer Aided Verification, Vol. 1855 of Lecture Notes in Computer Science, Springer Verlag, 2000, pp. 403–418.

[2] D. Dams, Y. Lakhnech, M. Steffen, Iterating transducers, in: G. Berry, H. Comon, A. Finkel (Eds.), Computer Aided Verification, Vol. 2102 of Lecture Notes in Computer Science, 2001, pp. 286–297.

[3] B. Boigelot, A. Legay, P. Wolper, Iterating transducers in the large, in: Proc. 15^{th} Int. Conf. on Computer Aided Verification, Vol. 2725 of Lecture Notes in Computer Science, 2003, pp. 223–235.

[4] A. Bouajjani, P. Habermehl, T. Vojnar, Abstract regular model checking, in: CAV04, Lecture Notes in Computer Science, Springer-Verlag, Boston, 2004, pp. 372–386.

[5] B. Boigelot, A. Legay, P. Wolper, Omega regular model checking, in: Proc. TACAS '04, 10^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, 2004, pp. 561–575.

[6] D. Fisman, A. Pnueli, Beyond regular model checking, in: FSTTCS01, Lecture Notes in Computer Science, 2001, pp. 156–170.

[7] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shahar, Symbolic model checking with rich assertional languages, Theoretical Computer Science 256 (2001) 93–112.

[8] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, S. Rajamani, Partial-order reduction in symbolic state space exploration, in: O. Grumberg (Ed.), Proc. 9^{th} Int. Conf. on Computer Aided Verification, Vol. 1254, Springer Verlag, Haifa, Israel, 1997, pp. 340–351.

[9] P. A. Abdulla, B. Jonsson, P. Mahata, J. d'Orso, Regular tree model checking, in: Proc. 14^{th} Int. Conf. on Computer Aided Verification, Vol. 2404 of Lecture Notes in Computer Science, 2002, pp. 555–568.

[10] A. Bouajjani, T. Touili, Extrapolating Tree Transformations, in: Proc. 14^{th} Int. Conf. on Computer Aided Verification, Vol. 2404 of Lecture Notes in Computer Science, 2002, pp. 539–554.

[11] P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, Algorithmic improvements in regular model checking., in: Proc. 15^{th} Int. Conf. on Computer Aided Verification, Vol. 2725 of Lecture Notes in Computer Science, 2003, pp. 236–248.

[12] W. Thomas, Automata on infinite objects, in: Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics, 1990, pp. 133–192.

[13] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree Automata Techniques and Applications, not yet published, 1999.

[14] A. Bouajjani, T. Touili, Reachability analysis of process rewrite systems, in: FSTTCS03, Lecture Notes in Computer Science, 2003, pp. 73–87.

[15] M. Y. Vardi, P. Wolper, Automata-theoretic techniques for modal logics of programs, Journal of Computer and System Science 32 (2) (1986) 183–221.

[16] M. Henzinger, T. Henzinger, P. Kopke., Computing simulations on finite and infinite graphs, in: Proc. 36^{th} Annual Symp. Foundations of Computer Science, 1995, pp. 453–463.

[17] M. Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, in: Proc. LICS '86, 1^{st} IEEE Int. Symp. on Logic in Computer Science, 1986, pp. 332–344.

[18] M. Nilsson, Regular model checking, website (2002).
URL `http://www.regularmodelchecking.com`

[19] P. Kretinsky, V. Rehak, J. Strejcek, Extended process rewrite systems: Expressiveness and reachability, in: CONCUR04, Vol. 3170 of Lecture Notes in Computer Science, Springer-Verlag, London, 2004, pp. 355–370.

[20] A. Bouajjani, P. Habermehl, P. Moro, T. Vojnar, Verifying programs with dynamic 1-selector-linked structures in regular model checking, in: TACAS05, Lecture Notes in Computer Science, Springer, 2005, pp. 13–29.

[21] A. Vardhan, K.Sen, M. Viswanathan, G. Agha, Actively learning to verify safety for fifo automata, in: FSTTCS04, Lecture Notes in Computer Science, 2004, pp. 494–505.

[22] P. Habermehl, T. Vojnar, Regular model checking using inference of regular languages, in: Proc. of 6th International Workshop on Verification of Infinite-State Systems—Infinity'04, 2004, pp. 61–72.

[23] P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, M. Saksena, Regular model checking for s1s + ltl, in: CAV04, Lecture Notes in Computer Science, Springer-Verlag, Boston, 2004, pp. 348–360.

[24] A. Bouajjani, A. Legay, P. Wolper, Handling liveness properties in (omega-)regular model checking, in: Proc. of 6th International Workshop on Verification of Infinite-State Systems—Infinity'04, 2004, pp. 37–48.

A Detailed Descriptions of the Examples

In this section, we describe the encoding of the protocols used in our experiments.

A.1 *Simple Token Protocol*

See Example 13 of Section 4 for the description of the protocol and its encoding in the tree regular model checking framework.

A.2 *Two Way Token Protocol*

The protocol is encoded using the same alphabet as the *Simple Token Protocol* above. The states we use are $\{q_0, q_1, q_2, q_3\}$. Their intuitive meaning is as follows:

- q_0: the node is idle, i.e., the token is not in the node, nor in the subtree below the node;
- q_1: the node is releasing the token to the node above it;
- q_2: the token is either in the node or in a subtree below the node (this state is accepting);
- q_3: the node is receiving the token from the node above it.

The transition relation is given by:

$$
\begin{array}{rlrlrl}
 & \xrightarrow{(n,n)} q_0 & & \xrightarrow{(t,n)} q_1 & & \xrightarrow{(n,t)} q_3 \\
(q_0, q_0) & \xrightarrow{(T,N)} q_1 & (q_0, q_1) & \xrightarrow{(N,T)} q_2 & (q_1, q_0) & \xrightarrow{(N,T)} q_2 \\
(q_0, q_0) & \xrightarrow{(N,N)} q_0 & (q_0, q_2) & \xrightarrow{(N,N)} q_2 & (q_2, q_0) & \xrightarrow{(N,N)} q_2 \\
(q_3, q_0) & \xrightarrow{(T,N)} q_2 & (q_0, q_3) & \xrightarrow{(T,N)} q_2 & (q_0, q_0) & \xrightarrow{(N,T)} q_3
\end{array}
$$

We consider as initial configurations all trees with just one token. These configurations are the ones accepted by the following tree automaton, where q_5 is the only accepting state:

$$
\begin{array}{rlrlrl}
 & \xrightarrow{n} q_4 & & \xrightarrow{t} q_5 & (q_4, q_4) & \xrightarrow{T} q_5 \\
(q_4, q_4) & \xrightarrow{N} q_4 & (q_4, q_5) & \xrightarrow{N} q_5 & (q_5, q_4) & \xrightarrow{N} q_5
\end{array}
$$

At first, every gate has its output set to *unknown*. At each step of the protocol, a gate for which each input is determined can set its output to a definite value. The process iterates until the root gate has a definite value. Each process has a local variable with values {0,1} for the leaf nodes and {U,0,1} for the internal nodes [6], (U is interpreted as "undefined yet"). The system percolates the disjunction of values in the leaves up to the root.

The states we use are $Q = \{q_0, q_1, q_u, q_d\}$. Intuitively, these states correspond to the following

- q_0: All nodes below (and including) the current node are labeled with 0; Do not make any change to them;
- q_1: The current node and at least one node below is labeled with 1. No node below is labeled with U. Do not change the nodes below;
- q_u: All nodes above (and including) the current node have not yet been changed (they are still undefined);
- q_d: A single change has occurred in the current node or below (accepting state).

The transition relation δ is given below (we use q_m to denote any member of $\{q_u, q_1, q_0\}$)

$$\begin{array}{rrr}
\xrightarrow{(0,0)} q_0 & \xrightarrow{(1,1)} q_1 & (q_0, q_0) \xrightarrow{(0,0)} q_0 \\
(q_0, q_1) \xrightarrow{(1,1)} q_1 & (q_1, q_0) \xrightarrow{(1,1)} q_1 & (q_1, q_1) \xrightarrow{(1,1)} q_1 \\
(q_m, q_m) \xrightarrow{(U,U)} q_u & (q_m, q_d) \xrightarrow{(U,U)} q_d & (q_d, q_m) \xrightarrow{(U,U)} q_d \\
(q_0, q_0) \xrightarrow{(U,0)} q_d & (q_0, q_1) \xrightarrow{(U,1)} q_d & (q_1, q_0) \xrightarrow{(U,1)} q_d \\
(q_1, q_1) \xrightarrow{(U,1)} q_d & &
\end{array}$$

The set of initial configurations consists of all binary trees whose leaves are labelled with 0 or 1, while the rest of the nodes are labelled with U. This set is characterized with the following tree automaton

$$\begin{array}{rr}
\xrightarrow{0} q_i & \xrightarrow{1} q_i \\
(q_i, q_i) \xrightarrow{U} q_{iu} & (q_{iu}, q_{iu}) \xrightarrow{U} q_{iu}
\end{array}$$

[6] To simplify the notation, we do not distinguish between the nullary and binary versions of the symbols 0 and 1.

where the accepting state is q_{iu}.

A.4 The Tree-Arbiter Protocol

In our model of the protocol, any process can be labeled as follows:

idle: the process does not do anything;

requesting: the process wants to access the shared resource;

token: the process has been granted the shared resource.

Furthermore, an interior process can be labeled as follows:

idle: the process together with all the process below are idle;

below: the token is somewhere in one subtree below this node (but not in the node itself).

The alphabet we use is $\{i, r, t, b\}$ for respectively *idle*, *requesting*, *token*, and *below*.

When a leaf is in state *requesting*, the request is propagated upwards until it encounters a node which is aware of the presence of the token (i.e. a node that either owns the token or has a descendant which owns the token). If a node has the token it can always pass it upwards, or pass it downwards to a child which is requesting. Each time the token moves a step, the propagation moves a step or there is no move; the request and the token cannot propagate at the same time.

We now describe the transition relation. The states are

$$\{q_i, q_r, q_t, q_{req}, q_{rel}, q_{grant}, q_m, q_{rt}\}.$$

Intuitively, these states have the following meaning:

q_i: Every node up to the current one is idle;

q_r: Every node up to the current node are either idle or requesting, with at least one requesting. there was no move of the propagation below;

q_t: The token is either in this node or below; token has not moved (this is an accepting state);

q_{req}: The current node is requesting the token for itself or on behalf of a child : The request is being propagated;

q_{rel}: The token is moving upwards from the current node;

q_{grant}: The token is moving downwards to the current node;

q_m: The token is in this node or below; the token has moved (this is an accepting state);

q_{rt}: The token is either in this node or below it, i.e, nothing happens above the current node (this is an accepting state).

Using these states, the transition relation is given by:

$$
\begin{array}{rlrlrl}
 & \xrightarrow{(i,i)} q_i & & \xrightarrow{(i,r)} q_{req} & & \xrightarrow{(r,r)} q_r \\
 & \xrightarrow{(r,t)} q_{grant} & & \xrightarrow{(t,t)} q_t & & \xrightarrow{(t,i)} q_{rel} \\
(q_i, q_i) & \xrightarrow{(i,i)} q_i & (q_r, q_i) & \xrightarrow{(i,i)} q_i & (q_i, q_r) & \xrightarrow{(i,i)} q_i \\
(q_r, q_r) & \xrightarrow{(i,i)} q_r & (q_{req}, q_i) & \xrightarrow{(i,i)} q_{req} & (q_i, q_{req}) & \xrightarrow{(i,i)} q_{req} \\
(q_{req}, q_r) & \xrightarrow{(i,i)} q_{req} & (q_r, q_{req}) & \xrightarrow{(i,i)} q_{req} & (q_r, q_r) & \xrightarrow{(i,r)} q_{req} \\
(q_r, q_i) & \xrightarrow{(i,r)} q_{req} & (q_i, q_r) & \xrightarrow{(i,r)} q_{req} & (q_r, q_r) & \xrightarrow{(r,r)} q_r \\
(q_r, q_i) & \xrightarrow{(r,r)} q_r & (q_i, q_r) & \xrightarrow{(r,r)} q_r & (q_{req}, q_r) & \xrightarrow{(r,r)} q_{req} \\
(q_r, q_{req}) & \xrightarrow{(r,r)} q_{req} & (q_r, q_i) & \xrightarrow{(r,t)} q_{grant} & (q_i, q_r) & \xrightarrow{(r,t)} q_{grant} \\
(q_r, q_r) & \xrightarrow{(r,t)} q_{grant} & (q_i, q_i) & \xrightarrow{(t,t)} q_t & (q_i, q_r) & \xrightarrow{(t,t)} q_t \\
(q_r, q_i) & \xrightarrow{(t,t)} q_t & (q_r, q_r) & \xrightarrow{(t,t)} q_t & (q_i, q_{req}) & \xrightarrow{(t,t)} q_{rt} \\
(q_{req}, q_i) & \xrightarrow{(t,t)} q_{rt} & (q_r, q_{req}) & \xrightarrow{(t,t)} q_{rt} & (q_{req}, q_r) & \xrightarrow{(t,t)} q_{rt} \\
(q_i, q_{grant}) & \xrightarrow{(t,b)} q_m & (q_{grant}, q_i) & \xrightarrow{(t,b)} q_m & (q_{grant}, q_r) & \xrightarrow{(t,b)} q_m \\
(q_r, q_{grant}) & \xrightarrow{(t,b)} q_m & (q_t, q_i) & \xrightarrow{(b,b)} q_t & (q_t, q_r) & \xrightarrow{(b,b)} q_t \\
(q_r, q_t) & \xrightarrow{(b,b)} q_t & (q_i, q_t) & \xrightarrow{(b,b)} q_t & (q_m, q_r) & \xrightarrow{(b,b)} q_m \\
(q_m, q_i) & \xrightarrow{(b,b)} q_m & (q_r, q_m) & \xrightarrow{(b,b)} q_m & (q_i, q_m) & \xrightarrow{(b,b)} q_m \\
(q_t, q_{req}) & \xrightarrow{(b,b)} q_{rt} & (q_{req}, q_t) & \xrightarrow{(b,b)} q_{rt} & (q_r, q_{rt}) & \xrightarrow{(b,b)} q_{rt} \\
(q_{rt}, q_r) & \xrightarrow{(b,b)} q_{rt} & (q_i, q_{rt}) & \xrightarrow{(b,b)} q_{rt} & (q_{rt}, q_i) & \xrightarrow{(b,b)} q_{rt} \\
(q_i, q_{rel}) & \xrightarrow{(b,t)} q_m & (q_{rel}, q_i) & \xrightarrow{(b,t)} q_m & (q_r, q_{rel}) & \xrightarrow{(b,t)} q_m \\
(q_{rel}, q_r) & \xrightarrow{(b,t)} q_m & (q_i, q_i) & \xrightarrow{(t,i)} q_{rel} & (q_i, q_r) & \xrightarrow{(t,i)} q_{rel} \\
(q_r, q_i) & \xrightarrow{(t,i)} q_{rel} & (q_r, q_r) & \xrightarrow{(t,i)} q_{rel} & &
\end{array}
$$

The set of initial configurations consists of binary trees where all the nodes are idle except the root which holds the token. This set is characterized with the following tree automaton

$$\xrightarrow{i} qi_i \quad (qi_i, qi_i) \xrightarrow{i} qi_i \quad (qi_i, qi_i) \xrightarrow{t} qi_t$$

where qi_t is the accepting state.

A.5 The Leader Election Protocol

In the protocol, the node is said to be elected if it has changed from candidate to elected. We use also undefined when the node has not been defined.

There are six states:

- q_c: There is at least a candidate in the tree below (this is an accepting state);
- q_n: No candidates below;
- q_{el}: The candidate to be elected is below (this is an accepting state);
- q_u: Undefined yet;
- q_{jel}: Just elected;
- q_{ch}: Something changed below.

The transition relation is given below:

$$\begin{array}{rrr}
\xrightarrow{(c,c)} q_c & \xrightarrow{(c,el)} q_{jel} & \xrightarrow{(n,n)} q_n \\
\xrightarrow{(el,el)} q_{el} & (q_c, q_c) \xrightarrow{(c,c)} q_c & (q_c, q_n) \xrightarrow{(c,c)} q_c \\
(q_n, q_c) \xrightarrow{(c,c)} q_c & (q_c, q_n) \xrightarrow{(c,el)} q_{jel} & (q_c, q_c) \xrightarrow{(c,el)} q_{jel} \\
(q_n, q_c) \xrightarrow{(c,el)} q_{jel} & (q_n, q_n) \xrightarrow{(n,n)} q_n & (q_u, q_u) \xrightarrow{(u,u)} q_u \\
(q_n, q_u) \xrightarrow{(u,u)} q_u & (q_u, q_n) \xrightarrow{(u,u)} q_u & (q_u, q_c) \xrightarrow{(u,u)} q_u \\
(q_c, q_u) \xrightarrow{(u,u)} q_u & (q_c, q_c) \xrightarrow{(u,u)} q_u & (q_n, q_n) \xrightarrow{(u,u)} q_u \\
(q_n, q_c) \xrightarrow{(u,u)} q_u & (q_c, q_n) \xrightarrow{(u,u)} q_u & (q_{ch}, q_u) \xrightarrow{(u,u)} q_{ch} \\
(q_{ch}, q_n) \xrightarrow{(u,u)} q_{ch} & (q_{ch}, q_c) \xrightarrow{(u,u)} q_{ch} & (q_u, q_{ch}) \xrightarrow{(u,u)} q_{ch} \\
(q_n, q_{ch}) \xrightarrow{(u,u)} q_{ch} & (q_c, q_{ch}) \xrightarrow{(u,u)} q_{ch} & (q_c, q_c) \xrightarrow{(u,c)} q_{ch}
\end{array}$$

$$
\begin{array}{llll}
(q_n, q_c) \xrightarrow{(u,c)} q_{ch} & (q_c, q_n) \xrightarrow{(u,c)} q_{ch} & (q_n, q_n) \xrightarrow{(u,n)} q_{ch} \\
(q_n, q_{el}) \xrightarrow{(el,el)} q_{el} & (q_c, q_{el}) \xrightarrow{(el,el)} q_{el} & (q_{el}, q_n) \xrightarrow{(el,el)} q_{el} \\
(q_{el}, q_c) \xrightarrow{(el,el)} q_{el} & (q_n, q_{jel}) \xrightarrow{(el,el)} q_{el} & (q_c, q_{jel}) \xrightarrow{(el,el)} q_{el} \\
(q_{jel}, q_n) \xrightarrow{(el,el)} q_{el} & (q_{jel}, q_c) \xrightarrow{(el,el)} q_{el} &
\end{array}
$$

The set of initial configurations is given by the following tree automaton, where q_0 is the only accepting state:

$$
\begin{array}{lll}
\xrightarrow{c} q_0 & \xrightarrow{n} q_1 & (q_0, q_0) \xrightarrow{u} q_0 \\
(q_0, q_1) \xrightarrow{u} q_0 & (q_1, q_0) \xrightarrow{u} q_0 & (q_1, q_1) \xrightarrow{u} q_1
\end{array}
$$

B Simulation Algorithms

In the present section, we give algorithms for compute downwrad and upward simulations on states of a finite tree automaton. The algorithms are extensions of the one presented in [16] for word automata.

Definition 39 *Let $A = (Q, F, \delta)$. For $p \in Q$, we let $pre_f(p)$ be the set $\left\{(r_1, \ldots, r_n) \mid (r_1, \ldots, r_n) \xrightarrow{f} p\right\}$. We let $pre_{f,i}(p)$ be the set $\left\{r \mid \exists r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n.\ (r_1, \ldots, r_{i-1}, r, r_{i+1}, \ldots, r_n) \xrightarrow{f} p\right\}$. For $P \subseteq Q$, we let $pre_f(P)$ to be the set $\bigcup_{p \in P} pre_f(p)$. We extend $pre_{f,i}$ to sets of states in a similar manner. Also, we define $post_f$ and $post_{f,i}$ in a similar manner.*

B.1 Computing Downward Simulations

We present an algorithm for computing the maximal downward simulation on states of a tree automaton. The result will be stored in the variable *sim* after termination.

Input: a tree automaton $A = (Q, F, \delta)$

Variables: sim, oldsim $\subseteq Q \times Q$

- **INITIALIZATION**
  for each $q \in Q$ **do**
    $oldsim(q){:=}Q$;
    $sim(q){:=}\{r \in Q | \forall f \in \Sigma.\ pre_f(q) \neq \emptyset \Rightarrow pre_f(r) \neq \emptyset\}$;

od;

- **COMPUTATION**

  While $\exists q \in Q$ such that $sim(q) \neq oldsim(q)$ **do**

  {**Invariant 1:** $\forall q \in Q$, we have $sim(q) \subseteq oldsim(q)$}

  {**Invariant 2:** $\forall q, q_1, \ldots, q_n, r$, if $(q_1, \ldots, q_n) \xrightarrow{f} q$ and $r \in sim(q)$, then $\exists r_1 \in oldsim(q_1), \ldots, \exists r_n \in oldsim(q_n)$ such that $(r_1, \ldots, r_n) \xrightarrow{f} r$}

  For each symbol f **do**

  For each i:$1 \leq i \leq rank(f)$ **do**

  $remove$:=$post_{f,i}(oldsim(q)) - post_{f,i}(sim(q))$;

  For each $r \in post_{f,i}(q)$ **do**

  $sim(r)$:=$sim(r) - remove$;

  od

  od

  od

  $oldsim(q)$:=$sim(q)$;

  od

Observe that when the algorithm terminates, we get $sim = oldsim$, and hence invariant 2 implies that sim is a downward simulation.

The simulation returned by the algorithm is maximal: whenever a pair is removed in our algorithm, then this pair doesn't belong to any downward simulation. A state r is removed from $sim(q)$ when it is present in $remove$. This may have occurred because information was propagated, but ultimately, this is done because some pair was in *oldsim* but not in sim after the initialization procedure. This means that if we follow the path through which pairs were put in the $remove$ sets, we can construct an execution of length $k+1$, showing that q is simulated by r up to depth k but not depth $k+1$. We thus conclude that there is no simulation relating q and r.

B.2 Computing Upward Simulations

We present an algorithm for computing the maximal upward simulation between states of a tree automaton.

Input: a tree automaton $A = (Q, F, \delta)$ and a downward simulation $\preccurlyeq_{down}$

Variables: sim, oldsim $\subseteq Q \times Q$

- **INITIALIZATION**

  for each $q \in Q$ **do**

  $oldsim(q)$:=Q;

$sim(q)$:=$\{r \in Q | \forall f, \forall i : 1 \leq i \leq rank(f)$, if $\exists q_1, \ldots, q_{i-1}, q_{i+1}, \ldots, q_n$ s.t $post_f(q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) \neq \emptyset$, then $\exists r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n$ s.t $post_f(r_1, \ldots, r_{i-1}, r, r_{i+1}, \ldots, r_n) \neq \emptyset$ and $\forall j \neq i. q_j \preccurlyeq_{down} r_j\}$;

od;

- **COMPUTATION**

While $\exists q \in Q$ such that $sim(q) \neq oldsim(q)$ **do**

{**Invariant 1:** $\forall q \in Q$, we have $sim(q) \subseteq oldsim(q)$}

{**Invariant 2:** $\forall q, q_1, \ldots, q_n$ and r_i, if $(q_1, \ldots, q_n) \xrightarrow{f} q$ and $r_i \in sim(q_i)$ then $\exists r, r_1, \ldots, r_{i-1}, r_{i+1}, \ldots, r_n$ s.t. $q_1 \preccurlyeq_{down} r_1, \ldots, q_{i-1} \preccurlyeq_{down} r_{i-1}$, $q_{i+1} \preccurlyeq_{down} r_{i+1}, \ldots, q_n \preccurlyeq_{down} r_n$ and $(r_1, \ldots, r_{i-1}, r, r_{i+1}, \ldots, r_n) \xrightarrow{f} r$ and $r \in oldsim(q)$}

For each symbol f **do**

  For each $i : 1 \leq i \leq rank(f)$ **do**

    $remove$:=$pre_{f,i}(oldsim(q)) - pre_{f,i}(sim(q))$;

    For each $r \in pre_{f,i}(q)$ **do**

      $sim(r)$:=$sim(r) - remove$;

    od

  od

od

$oldsim(q)$:=$sim(q)$;

od;

Correctness of this algorithm is similar to the case of the downward simulation above. The only notable difference is in the use of $\preccurlyeq_{down}$ to further constrain the pairs we consider as potential candidates in the initialization step. This is due to the definition of an upward simulation, which depends on the given downward simulation. When we compare two states q_i and r_i, with respect to a transition rule $(q_1, \ldots, q_n) \xrightarrow{f} q$, we only consider other rules $(r_1, \ldots, r_n) \xrightarrow{f} r$ for which it holds that $\forall j \neq i. q_j \preccurlyeq_{down} r_j$. Hence, we don't need to redo that check later in the algorithm.

Remark In [16], less straightforward but more efficient variants of the above algorithms are presented for the word case. The algorithms presented here can be modified in a similar manner.