Simulation-Based Iteration of Tree Transducers

Parosh Aziz Abdulla[1], Axel Legay[2,⋆], Julien d'Orso[3], and Ahmed Rezine[1]

[1] Dept. of Information Technology, Uppsala University,
P.O. Box 337, SE-751 05 Uppsala, Sweden
{parosh, rahmed}@it.uu.se

[2] Université de Liège, Institut Montefiore, B28,
4000 Liège, Belgium
legay@montefiore.ulg.ac.be

[3] IRCCyN - UMR CNRS 6597, BP 92 101,
44321 Nantes CEDEX 03, France
julien.dorso@irccyn.ec-nantes.fr

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. The central problem is to compute the transitive closure of a transducer. A main obstacle is that the set of reachable states is in general 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 several protocols.

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

⋆ This author is supported by a F.R.I.A grant.

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 [BJNT00, DLS01, BLW03, BHV04, BLW04].

This work aims at extending the paradigm of regular model checking to verify systems which operate on tree-like architectures. This includes several interesting protocols such as the Percolate Protocol ([KMM+01]) or the Tree-arbiter Protocol ([ABH+97]).

To verify such systems, we use the extension of regular model checking called *tree regular model checking*, which was introduced in [KMM+01, AJMd02, BT02]. 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 closure of a tree transducer. This problem was considered in [AJMd02, BT02]; however the proposed algorithms are most of the time 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. These simulations can be computed by efficient algorithms derived from those of Henzinger *et al.* ([HHK95]) for finite words.

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

We have implemented our algorithms in a tool which we have applied to a number of protocols including a Two-Way Token protocol, the Percolate Protocol ([KMM$^+$01]), a parametrized version of the Tree-arbiter Protocol ([ABH$^+$97]), 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 [DLS01, AJNd03, BLW03, BHV04, BLW04]. However, all current algorithms devoted to the computation of the transitive closure of a *tree* transducer are not efficient or not implementable. In [AJMd02], we presented a method for computing transitive closures of tree transducers. The method presented in [AJMd02] 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 [BT02] 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 [BT02] 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 [AJNd03], 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 [AJNd03] cannot be applied to tree transducers. In [DLS01], Dams, Lakhnech, and Steffen present an 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 [DLS01], the authors consider several definitions of simulations and bisimulations between 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 to 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.

Some proofs had to be omitted due to space constraints. A self-contained long version of this paper can be obtained from the authors.

2 Tree Automata

In this section, we introduce some preliminaries on trees and tree automata (more details can be found in [CDG+99]).

A *ranked alphabet* is a pair (Σ, ρ), where Σ is a finite set of symbols and ρ is a mapping from Σ to $\mathbb{N}$. For a symbol $f \in \Sigma$, we call $\rho(f)$ the *arity* of f. We let Σ_p 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 of the node. Sometimes, we abuse notation and use Σ to denote the ranked alphabet (Σ, ρ).

Following [CDG+99], 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 $b_1 b_2 ... b_{k-1}$. Also, nodes are labeled by symbols from Σ.

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 $r : 0 \le r < b_k$. 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.

Sets of trees are recognized using tree automata. 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, inclusion test, complementation, etc. In the sequel, we will omit the term bottom-up.

Definition 2. [**Tree Automata and Languages**]
A tree language is a set of trees.
A tree automaton [CDG+99, Tho90] 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, we assume Q and δ to be finite.

We say that A is *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 (that explains why it is called bottom-up), annotating states to the nodes of T. A transition rule of the form shown above 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.

Formally, a *run* r of A 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 $K = L(A)$.

We now define the notion of context. Intuitively, a context is a tree with "holes" instead of leaves. Formally, we consider a special symbol $\square \notin \Sigma$ with arity 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 $C[q_1, \ldots, q_k] \overset{r}{\Longrightarrow}_A q$ 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 3. [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. Our definition of a context coincides with the one of [BT03] where all leaves are holes. On the other hand, a context in [CDG+99] and [AJMd02] is a tree with a *single* hole.

3 Tree Relations and Transducers

In this section we introduce tree relations and transducers.

For a binary relation R, we use R^+ to denote the transitive closure of R.

For a ranked alphabet Σ and $m \geq 1$, we let $\Sigma^\bullet(m)$ be 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 *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 $T(\Sigma)$ as follows: $(T_1, \ldots, T_m) \in [K]$ iff $T_1 \times \cdots \times T_m \in K$.

We use tree automata also to characterize tree relations: an automaton A over $\Sigma^\bullet(m)$ characterizes an m-ary relation on $T(\Sigma)$, 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 as case, we denote this relation by $R(A)$.

Definition 4. [**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. Our definition of tree transducers is a restricted version of the one considered in [BT02] in the sense that we only consider transducers that do not modify the structure of the tree. In [BT02], such transducers are called relabeling transducers.

4 Tree Regular Model Checking

We use the following framework known as *tree regular model checking* [AJMd02, BT02, KMM+01]:

Definition 5. [**Program**]
A program *is a triple $P = (\Sigma, \phi_I, D)$ where*

- *Σ is a ranked alphabet, over which the program configurations are encoded as trees;*

– *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 $R(D)$.*

In a similar manner to the the case of words (see [BJNT00]), the problems we are going to consider are the following:

– *Computing the transitive closure:* The goal is to compute a new tree transducer D^+ representing the transitive closure 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 states:* 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.

We will first provide a technique for computing D^+. Then, we will show the modifications needed for computing $R(D^+)(\phi_I)$ without computing 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 $(T, T') \in D^+$, 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 6. [**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 $(w_1(i) \ldots, w_p(i)) \xrightarrow{(f_i, f_{i+1})} w(i)$ 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 (proved in [AJMd02]) which states that H characterizes the transitive closure of the relation of D.

Lemma 1. *For a transducer D and its history transducer H, we have that $R(H) = R(D^+)$.*

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 sufficient condition 5 of Theorem 1);
- $\delta_\simeq$ contains rules of the form $(x_1, \ldots, x_n) \xrightarrow{f}_\simeq x$ iff there are states $q_1 \in x_1, \ldots, q_n \in x_n, q \in x$ such that there is a rule $(q_1, \ldots, q_n) \xrightarrow{f} q$ 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: $D^{\leq 1}, D^{\leq 2}, D^{\leq 3}, \ldots$ (where $D^{\leq i} = \bigcup_{n=1}^{n=i} D^n$), and collapses states[4] according to $\simeq$. We thus obtain $D_\simeq^{\leq 1}, D_\simeq^{\leq 2}, \ldots$
- The procedure terminates when the relation R^+ is accepted by $D_\simeq^{\leq i}$. This can be tested by checking if the language $D_\simeq^{\leq i} \circ D$ is included in $D_\simeq^{\leq i}$.

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 good equivalence relations as follows: we define two simulation relations; namely a downward simulation relation $\preccurlyeq_{down}$ and an *upward simulation relation* $\preccurlyeq_{up}$, which together induce an equivalence relation $\simeq$. Then, we give sufficient conditions of the simulation relations which guarantee that the induced equivalence $\simeq$ is a good one.

6.1 Downward and Upward Simulation

We start by giving the definitions.

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

Definition 7. [**Downward Simulation**]
Let $A = (Q, F, \delta)$ be a tree automaton. A binary relation $\preccurlyeq_{down}$ is a downward simulation iff for any $n \geq 1$ and any symbol $f \in \Sigma_n$, for all states $q, q_1, \ldots, q_n, r$, the following holds:
Whenever $q \preccurlyeq_{down} r$ and $(q_1, \ldots, q_n) \xrightarrow{f} q$, then there exist states $r_1, \ldots, r_n$ such that $q_1 \preccurlyeq_{down} r_1, \ldots, q_n \preccurlyeq_{down} r_n$ and $(r_1, \ldots, r_n) \xrightarrow{f} r$.

Definition 8. [**Upward Simulation**]
Let $A = (Q, F, \delta)$ be a tree automaton. Given a downward simulation $\preccurlyeq_{down}$, a binary relation $\preccurlyeq_{up}$ 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 $(q_1, \ldots, q_n) \xrightarrow{f} q$, then 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 $(r_1, \ldots, r_n) \xrightarrow{f} r$.

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 2. *The reflexive closure and the transitive closure of a downward simulation $\preccurlyeq_{down}$ are both downward simulations. Furthermore, there is a unique maximal downward simulation.*

Lemma 3. *Let $\preccurlyeq_{down}$ be a reflexive (transitive) downward simulation. 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.*

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

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

Definition 9. [**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 10. [**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 4. *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.*

6.2 Sufficient Conditions for Soundness and Completeness

We give sufficient conditions for the 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 $\simeq$ induced by the two relations $\preceq$ and $\preccurlyeq_{down}$ satisfying the following *sufficient* conditions:

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 sufficient conditions, then the induced relation is indeed an equivalence.

Lemma 5. *Let $A = (Q, F, \delta)$ be a tree automaton. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ which satisfies the above sufficient conditions, as well as their induced relation $\simeq$. We have that $\simeq$ is an equivalence relation on states of A.*

The above Lemma holds since Conditions 1 through 3 imply directly that $\preccurlyeq_{down}$ and $\preceq$ satisfy the premises needed by Lemma 4.

Next, we state that such an equivalence relation is sound and precise.

Theorem 1. *Let $A = (Q, F, \delta)$ be a tree automaton. Consider two binary relations $\preccurlyeq_{down}$ and $\preceq$ satisfying the above sufficient conditions, 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, $L(A_{\simeq}) = L(A)$.*

Theorem 1 can be used to relate the languages of H and $D_{\simeq}$.

We are now ready to prove the soundness and the completeness of our on-the-fly algorithm (assuming a computable equivalence relation $\simeq$).

Theorem 2. *Consider two binary relations on the states of H $\preccurlyeq_{down}$ and $\preceq$, satisfying the hypothesis of Theorem 1. Let $\simeq$ be their induced equivalence relation. If the algorithm terminates at step i, then the transducer $D_{\simeq}^{\leq i}$ accepts the same relation as $D_{\simeq}$.*

6.3 Sufficient Condition for Computability

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

Definition 11. [**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 3. *Let $\preceq_1$ and $\preceq_2$ be both reflexive, transitive, effective and independent. Let $\simeq$ be their induced equivalence. Then for any state x of H, we can compute its equivalence class $[x]$ w.r.t. $\simeq$.*

The theorem follows by definition of $\simeq$, and effectiveness [5] of $\preceq_1$ and $\preceq_2$. $\square$

An equivalence relation that satisfies hypothesis of Theorem 1 and Theorem 3 can be used in the on-the-fly algorithm of Section 5 to compute the transitive closure of a tree transducer. The next step is to provide a concrete example of such an equivalence. Because we are not able to compute the infinite representation of H, the equivalence 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 1 and Theorem 3. We first introduce prefix- and suffix-copying states.

Definition 12. [**Prefix-Copying State**]
Given a transducer D, and a state 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 13. [**Suffix-Copying State**]
Given a transducer D, and a state 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 relation on the states of the history transducer.

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

Definition 14. [**Generated Relation**]
Given a set S of pairs of states of H, 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 $\preccurlyeq_{down}$ that satisfy the sufficient conditions for computability (Theorem 3) and conditions for exactness of abstraction (Lemma 6.2).

Definition 15. [**Simulation Relations**]

- *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 $\preccurlyeq_{up}^1$ be the maximal upward simulation computed on $D \cup D^2$. Then, we define $\preceq$ to be the relation generated by the maximal set $S \subseteq \preccurlyeq_{up}^1$ 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 $q \in Q_N$.

In the full version of the paper, we provide efficient algorithms for computing the simulations needed for Definition 15. Those algorithms are adapted from those provided by Henzinger *et al.* [HHK95] for the case of finite words.

Let us state that the simulations of Definition 15 satisfy the hypothesis needed by Theorems 1 and 3.

Lemma 6. *The following properties of $\preccurlyeq_{down}$ hold:*

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

Lemma 7. *The following properties of $\preceq$ hold:*

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

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

Lemma 8. *$\preceq$ and $\preccurlyeq_{down}$ are independent.*

Lemma 9. *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 $\xrightarrow{f} y$.*

We conclude that $\preceq$ and $\preccurlyeq_{down}$ satisfy the hypothesis of Theorem 1 and Theorem 3 and can thus be used by the on-the-fly procedure presented in Section 5.

8 Computing Reachable Configurations

We now sketch the modifications needed to compute $R(D^+)(\phi_I)$ without computing D^+. When checking for safety properties, such a computation is known to be sufficient. Computing $R(D^+)(\phi_I)$ rather than D^+, can simply be done by lightly modifying the definition of the history transducer. Assume that we have constructed a tree automaton A_{ϕ_I} for ϕ_I, we replace the transducer run in the first "row" of the history transducer by a transducer that only accept trees from A_{ϕ_I} in input. Such a transducer can easily by constructed. Let D be the transducer representing the transition of the system, the restricted transducer is obtained by taking the intersection between D and $A_{\phi_I} \times T(\Sigma)$ where Σ is the ranked alphabet of the system. Computing $R(D^+)(\phi_I)$ is often less expensive than computing D^+ because it only considers reachable sets of states (see Section 9 for a time comparison). 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 www.regularmodelchecking.com).

In Table 1 we report the result of running our implementation on a number of parametrized protocols for which we have computed the set of reachable states as well as the transitive closure of their transition relation. A full description of the protocols is given in the full version of the paper.

In our previous work [AJMd02], we were able to handle the first three protocols of the table (computation times were very long, however).

The technique of [BT02] was manually applied to compute the set of reachable states of the tree-arbiter protocol (and of smaller examples). But, 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. In our approach, all computations are entirely automatic.

Observe that we are not able to compute the transitive closure of the transition relation of the tree-arbiter protocol (in fact, we do not know if it is regular

Table 1. Results

Relation	$\lvert D\rvert$	$\lvert D^+\rvert$	max size	$\lvert D^+(\phi_I)\rvert$	max size
Simple Token Protocol	3	4	15	3	17
Two-Way Token Protocol	4	6	28	3	26
Percolate Protocol	4	6	40	3	21
Tree-arbiter Protocol	8	-	-	10	246
Leader Election Protocol	6	9	105	10	150

or not). However, we are already 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.

This 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.

We believe that substantial efficiency improvement can be achieved by considering more general equivalence relations than the one defined in Section 7, and by refining our algorithms for computing simulation relations.

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. 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 [BT03].

Finally, 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 [BT02] (the detection of an increment can be done by isolating part of the automaton with the help of (bi)simulations).

References

[ABH+97] R. Alur, R.K. Brayton, T.A. Henzinger, S. Qadeer, and S.K. Rajamani. Partial-order reduction in symbolic state space exploration. In O. Grumberg, editor, *Proc. 9th Int. Conf. on Computer Aided Verification*, volume 1254, pages 340–351, Haifa, Israel, 1997. Springer Verlag.

[AJMd02] P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In *Proc. 14th Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[AJNd03] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Algorithmic improvements in regular model checking. In *Proc. 15th Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 236–248, 2003.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV04*, Lecture Notes in Computer Science, Boston, July 2004. Springer-Verlag.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla, editors, *Proc. 12th Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer Verlag, 2000.

[BLW03] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *Proc.* 15th *Int. Conf. on Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235, 2003.

[BLW04] B. Boigelot, A. Legay, and P. Wolper. Omega regular model checking. In *Proc. TACAS '04,* 10th *Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, 2004.

[BT02] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc.* 14th *Int. Conf. on Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, 2002.

[BT03] A. Bouajjani and T. Touili. Reachability analysis of process rewrite systems. In *Proc. Int. Conf. on Foundations of Software Technology and Theoritical Computer Science (FSTTCS'03)*, Lecture Notes in Computer Science, 2003.

[CDG$^+$99] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications.* not yet published, October 1999.

[DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, 2001.

[HHK95] M. Henzinger, T. Henzinger, and P. Kopke. Computing simulations on finite and infinite graphs. In *Proc.* 36th *Annual Symp. Foundations of Computer Science*, pages 453–463, 1995.

[KMM$^+$01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256:93–112, 2001.

[Tho90] W. Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Methods and Semantics*, pages 133–192, 1990.