Regular Tree Model Checking

Parosh Aziz Abdulla Bengt Jonsson Pritha Mahata

Julien d'Orso

Dept. of Computer Systems
P.O. Box 325
S-751 05 Uppsala, Sweden
{parosh,bengt,pritha,juldor}@docs.uu.se

Abstract

In this paper, we present an approach for algorithmic verification of infinite-state systems with a parameterized tree topology. Our work is a generalization of regular model checking, where we extend the work done with strings toward trees. States are represented by trees over a finite alphabet, and transition relations by regular, structure preserving relations on trees. We use an automata theoretic method to compute the transitive closure of such a transition relation. Although the method is incomplete, we present sufficient conditions to ensure termination.

We have implemented a prototype for our algorithm and show the result of its application on a number of examples.

Appendix A – Proofs of Lemmas

Appendix B – Experimental Data

1 Introduction

Model Checking [CES86, QS82] has become one of the most prominent techniques for algorithmic verification of reactive systems. An important challenge has recently been to extend the applicability of model checking to infinite-state systems. One source of infiniteness of state spaces is *parameterization*. In a parameterized system the description of the system is parameterized by the number of components inside the system. Such a system typically consists of a network of components; and we are interested in verifying the correctness of the system regardless of the number of components. A parameterized system represents a family with an infinite number of instances; namely an instance for each instantiation of the parameter. This means that the total state space of the entire system is the union of the state spaces of all the instances, and therefore this state space is infinite.

For several classes of properties, model checking can be performed using some form of reachability analysis. For example, it is well-known that safety properties can be stated by specifying a set of "bad states" which we do not want to occur in the execution of the system. Checking safety properties amounts then to checking reachability for a set of bad states. To perform reachability analysis in the context of infinite systems, we need to provide a symbolic representation of infinite sets of states. This representation should be "adequate" in the sense that it should allow the operations needed for implementing reachability analysis such as union, intersection, checking emptiness, computing pre- and post-images with respect to transitions, etc.

A framework which has recently been advocated for model checking of infinite-state systems is *regular model checking* (e.g. [KMM$^+$97, KMM$^+$01, WB98, BJNT00]).

In regular model checking, states are represented by strings over a finite alphabet, while sets of states are represented by regular sets. Regular relations specified by finite-state transducers are used to describe actions between states. Regular model checking has been used to verify several classes of protocols with linear or ring-formed topologies, such as mutual exclusion protocols and cache coherence protocols (e.g. [BJNT00, Mai01, PRZ01, APR$^+$01, FP01]).

On the other hand, there are several classes of systems which are beyond the capability of regular model checking, either because the behaviour of the system cannot be captured by a regular relation [FP01], or because the topology of the system is not linear. In this paper, we extend the work in [JN00, BJNT00] in order to obtain a model checking algorithm for tree-formed protocols. We propose *regular tree languages* as a symbolic representation of state spaces and *regular tree relations*, characterized by finite-state *tree transducers*, as a symbolic representation of the transition relation.

A major problem in model checking of parameterized systems is that the depth of the state space is in general not bounded. This means that standard iteration based symbolic reachability algorithms [BCMD92, McM93] are not guaranteed to terminate for these systems. Therefore, an important challenge is how to *accelerate* the standard algorithm in order to make it terminate more often on practical examples. One way to achieve that is to augment the algorithm by adding the effect of arbitrarily long sequences of actions. Since an action in our case is modelled by a regular tree relation, this amounts to computing the transitive closure of a regular tree relation. For instance, the effect of an action which sends a token upwards in a tree is that the token is propagated an arbitrary number of steps toward the root of the tree.

The main contribution of this paper is to show how to compute the transitive closure for a large class of actions. Starting from a tree transducer corresponding to an action, we compute a new (symbolic) transducer corresponding to the transitive closure. We also classify a class of actions for which the construction of the symbolic transducer always terminates. We do that by extending the notion of *local depth* [JN00] to trees. Intuitively, an action has local depth k if repeated applications of the action changes each node of the tree at most k times. We show that, for any action with a finite local depth, we can compute a finite-state tree transducer corresponding to the transitive closure of the action.

We have implemented a prototype for computing such a transitive closure and verification. We show the result of running our algorithm for verification of parameterized versions of a number of protocols: two token tree protocols, the PERCOLATE protocol [KMM$^+$97], and a tree arbiter described in [ABH$^+$97].

Related work Regular model checking has been proposed by [KMM$^+$97, KMM$^+$01] and [WB98]. Several techniques have been proposed for accelerating reachability analysis for parameterized systems such as bisimulation [DLS01], widening [BJNT00, Tou01], automatic invariant generation [PRZ01, APR$^+$01], and transitive closure [ABJN99, JN00, BJNT00].

The paper [FP01] goes beyond regular languages using context-free languages as a symbolic representation. The paper [BMT01] proposes a subclass of regular languages closed under a larger set of operations than regular languages.

The difference between the above works and the work of this paper is that they all consider systems with linear topologies. Our work can be seen as a generalization of the techniques described in [JN00] for word transducers. The work in [KMM$^+$97] also considers tree-formed protocols. However, [KMM$^+$97] only considers transducers that represent the effect of a single application of an action rather than the transitive closure.

Outline In Section 2 and Section 3 we give some preliminaries on words and trees respectively. In Section 4 we introduce tree automata and tree transducers and use them to define regular tree languages and relations. Section 5 describes symbolic transducers and how they can be used to characterize the transitive closure of regular tree relations. In Section 6 we describe a saturation operation which helps to accelerate the construction of symbolic transducers. Section 7 describes the local depth property for actions. In Section 8, we show the result of our implementation on a number of examples. Finally, in Section 9, we give some conclusions and directions for future research.

2 Words

In this section, we recall some standard definitions and results for word languages. The concepts of *finite* automata and *regular languages* are defined as usual. For a word w, and $i : 1 \leq i \leq |w|$, we let $w(i)$ denote the i^{th} element of w. For words $w_1, \ldots, w_m$ of equal length k over an alphabet Σ, we let $w_1 \times \cdots \times w_m$ be the word w over Σ^m such that $w(i) = (w_1(i), \ldots, w_m(i))$ for $i : 1 \leq i \leq k$. An m-ary *relation* on the alphabet Σ is a set of tuples of the form $(w_1, \ldots, w_m)$, where $w_1, \ldots, w_m \in \Sigma^*$ and $|w_1| = \cdots = |w_m|$. We observe that a language K over Σ^m characterizes an m-ary relation $[K]$ on Σ in the sense that

$(w_1, \ldots, w_m) \in [K]$ if and only if $(w_1 \times \cdots \times w_m) \in K$. A relation R is *regular* if $R = [K]$ for some regular language K.

For relations R and R', we define $R \otimes R^{'}$ as the relation $(w_1, \ldots, w_k, w'_1, \cdots, w'_m)$ where $(w_1, \ldots, w_k) \in R$, $(w'_1, \ldots, w'_m) \in R'$, and $|w_1| = |w'_1|$. Notice that the third condition ensures $|w_1| = \cdots = |w_k| = |w'_1| = \cdots = |w'_m|$. For a relation R of arity m and $i : 1 \leq i \leq m$, we let $R|_i$ denote the relation $\{(w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_m) \mid (w_1, w_2, \ldots, w_m) \in R\}$. The operation is generalized in the obvious manner to $R|_I$, where I is a subset of $\{1, \ldots, m\}$.

It is straightforward to show that regular relations are closed under $\otimes$ and $|_I$.

Sometimes, we use the term *word language* instead of *language* to avoid confusion with *tree languages* (defined later). The same applies to other concepts, e.g. automata, relations, etc.

3 Trees

In this section, we introduce some preliminaries on trees and tree relations.

A *ranked alphabet* is a pair (Σ, ρ), where Σ is a finite set of symbols and ρ is a mapping from Σ to $\mathbb{N}$. 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 labelled 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).

Trees Following [CDG+99], the nodes in a tree are represented by strings over $\mathbb{N}$. More precisely, the empty string 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 labelled by symbols from Σ.

Formally, 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.

We let $n \in T$ indicate that $n \in S$, and let $f \in T$ denote that $\lambda(n) = f$ for some $n \in T$.

For a tree $T = (S, \lambda)$ and a node $n \in T$, the *subtree* of T rooted at n is the tree $T_n = (S_n, \lambda_n)$, where $S_n = \{b \mid nb \in S\}$ and $\lambda_n(b) = \lambda(nb)$.

Tree Relations We generalize the definition of a relation from words to trees.

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 $T_1 \cong \cdots \cong T_m$. In a similar manner to the case of words, a tree language K over $\Sigma^\bullet(m)$ characterizes an m-ary tree relation $[K]$ on $T(\Sigma)$. Notice that the condition of being structurally equivalent is a generalization of the condition of having the same length in the case of words (Section 2). Furthermore, in the case of words we worked with Σ^m (rather than $\Sigma^\bullet(m)$) since symbol arities were not relevant there.

The operations of intersection $\cap$ and union $\cup$ are defined as usual. The operation $|_i$ and its generalization $|_I$ are defined in a similar manner to words. For regular tree relations R and R', we define $R \otimes R'$ as the set of tuples $\left(T_1, \cdots, T_m, T_1', \cdots, T_n'\right)$ such that $(T_1, \cdots, T_m) \in R$, $\left(T_1', \cdots, T_n'\right) \in R'$, and $T_1 \cong T_1'$. Observe that the third condition is again a generalization of the corresponding condition in the case of words. We use $\circ$ to denote the composition of two binary relations as usual. We use R^i to denote i compositions of the relation R and define $R^* = \cup_{i \geq 0} R^i$ and $R^+ = \cup_{i \geq 1} R^i$.

Example 3.1 Let $\Sigma = \{and, or, 1, 0\}$ with $\rho(and) = \rho(or) = 2$ and $\rho(1) = \rho(0) = 0$. Figure 1(a) shows an example of a tree in $T(\Sigma)$ and Figure 1(b) shows an example of a tree in $T(\Sigma^\bullet(2))$. We write the label of a node inside a circle, and the node itself above the circle. $\square$

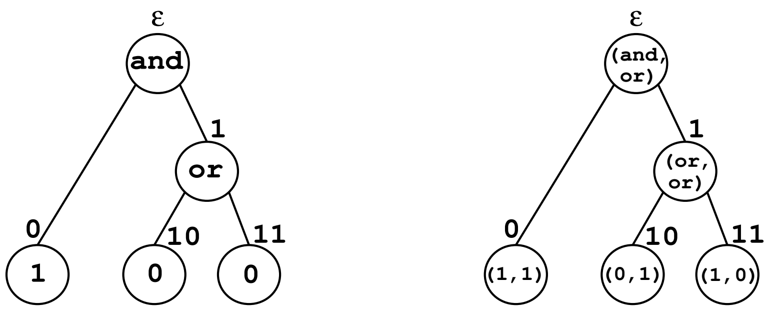

Figure 1: Examples of trees in $T(\Sigma)$ and $T(\Sigma^\bullet(2))$

4 Tree Automata

In this section, we introduce tree automata and use them to recognize regular tree languages and regular tree relations.

A *tree language* is a set of trees.

A *tree automaton* over a ranked alphabet Σ is a tuple $A = (Q, F, \delta)$, where Q is a finite 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.

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 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 $\lambda(n) = f$ (with $f \in \Sigma_p$), 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 $n_1, \ldots, n_k$:

$$\left((r(n_1), \ldots, r(n_k)) \xrightarrow{\lambda(n)} r(n) \right) \in \delta.$$

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 use tree automata also to characterize relations: An automaton A over $\Sigma^\bullet(m)$ characterizes an m-ary relation on $T(\Sigma)$, namely the relation $R = [L(A)]$. A relation R is said to be *regular* if there is a tree automaton A with $R = [L(A)]$. Sometimes, we denote R by $R(A)$.

In [CDG$^+$99], it is shown that regular tree languages are closed under the Boolean operations. Closedness under the operators $\otimes$ and $R|_I$ is straightforward. From the fact that $R \circ R' = ((R \otimes T(\Sigma)) \cap (T(\Sigma) \otimes R'))|_2$ we get the following

Lemma 4.1 Regularity is closed under composition.

Although Lemma 4.1 states that regularity is preserved by a finite number of applications of the $\circ$ operator, it is well-known that regularity is not preserved by $*$ even in the case of words (i.e. R^* need not be regular even if R is).

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

Example 4.2 Let B be a tree automaton over $\Sigma = \{0, 1, \mathit{and}, \mathit{or}\}$ (with $\rho(\mathit{and}) = \rho(\mathit{or}) = 2$ and $\rho(1) = \rho(0) = 0$), where $Q = \{q_0, q_1\}$, $F = \{q_1\}$ and δ:

$$\begin{array}{llll}
\xrightarrow{0} q_0 & \xrightarrow{1} q_1 & & \\
(q_0, q_0) \xrightarrow{or} q_0 & (q_0, q_1) \xrightarrow{or} q_1 & (q_1, q_0) \xrightarrow{or} q_1 & (q_1, q_1) \xrightarrow{or} q_1 \\
(q_0, q_0) \xrightarrow{and} q_0 & (q_0, q_1) \xrightarrow{and} q_0 & (q_1, q_0) \xrightarrow{and} q_0 & (q_1, q_1) \xrightarrow{and} q_1
\end{array}$$

B recognizes the tree language which is the set of true Boolean expressions over Σ. The tree in Figure 1(a) is recognized by B. $\square$

Example 4.3 Token Tree Protocol As a running example in this paper, we consider a tree transducer modelling the behaviour of a simple token tree protocol. The system consists of processes that are connected in a binary tree-like fashion. Each process stores a single bit which reflects whether the process has a token or not. The token tree passes a token from a leaf to the root. We represent the system by a tree transducer over an alphabet consisting of $t, n \in \Sigma_0$ representing processes at the leaves, and $N, T \in \Sigma_2$ representing processes at the inner nodes of the tree. Processes labeled by $\{n, N\}$ are those which do not have a token, while those labeled by $\{T, t\}$ are those which do have the token. The set of states is $\{q_0, q_1, q_2\}$ where q_2 is the (single) final state. The transition relation is given by:

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

Intuitively, the states correspond to the following

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 in the tree;

q_2 the token is either in the node or in a subtree below the node.

□

5 Symbolic Transducers

In this section we show how to compute the transitive closure of regular tree relations. More precisely, given a tree transducer D, we generate a new infinite tree transducer H, called the *history transducer* of D, such that $R(H) = (R(D))^*$. We also introduce *symbolic transducers* which are compact representations of history transducers.

History Transducers With a transducer D we associate a *history transducer* which corresponds to the reflexive transitive closure of $R(D)$. Each state of H is a word of the form $q_1 \cdots q_k$ where $q_1, \ldots, q_k$ are states in D. 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.

Formally, 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 $k \geq 0$ 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 $(w_1(i) \ldots, w_p(i)) \stackrel{(f_i, f_{i+1})}{\longrightarrow} w(i)$ 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. We also notice that if $(T \times T') \stackrel{r}{\Longrightarrow}_H w$, then there is a $k \geq 0$ 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 reflexive transitive closure of $R(D)$.

Lemma 5.1 For a transducer D and its history transducer H, we have $R(H) = (R(D))^*$.

Symbolic Transducers For a transducer D, the symbolic transducer S of D is a compact representation of the history transducer H of D. More precisely, each state of S is a (word) regular expression over the states of D. A state ϕ in S represents all states in the history transducer which are (words) belonging to the language of ϕ.

Formally, we assume a transducer $D = (Q, F, \delta)$ and the corresponding history transducer $H = (Q_H, F_H, \delta_H)$. To define symbolic transducers, we first need the following definition.

For regular expressions $\phi_1, \ldots, \phi_p$ and symbols f, f', define $(\phi_1, \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow}$ to be the set

$$\left\{ w | \ \exists w_1 \in \phi_1. \ldots. \exists w_p \in \phi_p. \left((w_1, \ldots, w_p) \stackrel{(f,f')}{\longrightarrow} w \right) \in \delta_H \right\}$$

Lemma 5.2 For regular expressions $\phi_1, \ldots, \phi_p$ and symbols f, f', the set $(\phi_1, \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow}$ is effectively regular.

Consider a tree transducer $D = (Q, F, \delta)$ over a ranked alphabet Σ. We define the *symbolic (tree) transducer* S for D to be the (possibly infinite-state) transducer (Q_S, F_S, δ_S), where Q_S is a set of regular expressions over Q, F_S is a set of regular expressions over Q, and δ_S contains a set of rules each of the form

$$(\phi_1, \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow} \phi$$

The transducer S is derived from D according to Algorithm 2 (see Figure 2). Observe that, by Lemma 5.2, the regular expression ϕ at line 4 of the code is always computable.

The following lemmas state the relationship between symbolic and history transducers.

Lemma 5.3 Consider a transducer D and its corresponding history and symbolic transducers H and S. For trees T and T'

- if $(T \times T') \Longrightarrow_S \phi$ then $(T \times T') \Longrightarrow_H w$ for each $w \in \phi$.
- if $(T \times T') \Longrightarrow_H w$ then $(T \times T') \Longrightarrow_S \phi$ for some ϕ with $w \in \phi$.

Input: Tree Transducer $D = (Q, F, \delta)$
Output: Symbolic Transducer $S = (Q_S, F_S, \delta_S)$
begin
1. $Q_S = \emptyset$, $F_S = \emptyset$, $\delta_S = \emptyset$,
2. **repeat**
   3. **for** each p, $f, f' \in \Sigma_p$,and $\phi_1, \ldots, \phi_p \in Q_S$ **do**
      4. $\phi \; := \; \left((\phi_1, \ldots, \phi_p) \xrightarrow{(f,f')} \right)$
      5. $Q_S \; := \; Q_S \cup \{\phi\}$
      6. $\delta_S \; := \; \delta_S \cup \left\{ (\phi_1, \ldots, \phi_p) \xrightarrow{f,f'} \phi \right\}$
   7. **od**
8. **until** no new states or rules can be added to Q_S and δ_S
9. $F_S \; := \; \{\phi \in Q_S | \; (\phi \cap F^*) \neq \emptyset\}$

end

Figure 2: Algorithm 2 : Computing symbolic transducer

Corollary 5.4 For a transducer D and its corresponding history and symbolic transducers H and S, we have $R(S) = R(H)$.

From Lemma 5.1 and Corollary 5.4 we get

Theorem 5.5 For a transducer D and its corresponding symbolic transducer S, we have $R(S) = (R(D))^*$.

Termination Since there are infinitely many regular expressions over the set of states of D, the algorithm in Figure 2 may in general not terminate.

Example 5.6 Consider the token tree transducer of Example 4.3.

A transition of the corresponding history transducer is $(q_0q_0q_0, q_1q_0q_0) \xrightarrow{(N,N)} q_2q_1q_0$ corresponding to the three transductions $(q_0, q_1) \xrightarrow{(N,T)} q_2$, followed by $(q_0, q_0) \xrightarrow{(T,N)} q_1$, followed by $(q_0, q_0) \xrightarrow{(N,N)} q_0$.

When we run Algorithm 2 on the protocol, we get e.g. $\phi_0 = {q_0}^* = \left(\xrightarrow{(n,n)} \right)$ and $\phi_1 = q_1{q_0}^* = \left(\xrightarrow{(t,n)} \right)$. If we consider the pair of symbols (N, N) and the regular expressions (ϕ_0, ϕ_1), we get, in the next step of the algorithm, the new expression $\phi_2 = q_2q_1q_0^* = \left((\phi_0, \phi_1) \xrightarrow{(N,N)} \right)$. These steps give the rules $\xrightarrow{(n,n)} \phi_0$, $\xrightarrow{(t,n)} \phi_1$, and $(\phi_0, \phi_1) \xrightarrow{(N,N)} \phi_2$, respectively. In Section 8 and Appendix B, we give a description of the whole transducer we get from running Algorithm 2 on the example.

□

6 Saturation

In order to make the algorithm in Figure 2 terminate more often, we present in this section a method to accelerate the iterations of the algorithm. We do that by defining the notion of *idempotent states* and then *saturating* all generated regular expressions by such states.

Idempotent States To define idempotent states, we need some preliminaries. First, we define the notion of context. Intuitively, a context is a tree with a single "hole" at one of its 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 there is exactly one $n_c \in S_C$ with $\lambda_C(n_c) = \square$. In the sequel, we will always assume $n_c \in S_C$ to be the unique node with $\lambda_C(n_c) = \square$.

For a context $C = (S_C, \lambda_C)$ and a tree $T = (S, \lambda)$, we define $C[T]$ to be the tree (S_1, λ_1), where

- $S_1 = S_C \cup \{n_c \cdot n |\ n_c \in S_C$ and $\lambda_C(n_c) = \square$ and $n \in S\}$.
- for each $n \in S_C$ with $n \neq n_c$ we have $\lambda_1(n) = \lambda_C(n)$.
- for each $n_1 = n_c \cdot n$ with $n \in S$ we have $\lambda_1(n_1) = \lambda(n)$.

Notice that the above operation represents a substitution, where we replace the hole in C by T.

Consider a tree transducer $D = (Q, F, \delta)$ over a ranked alphabet Σ. We extend the notion of runs to contexts. Let q be a state and $C = (S_C, \lambda_C)$ a context. A *run* r of D on C from q is defined in a similar manner to a run (Section 4) except that $r(n_c) = q$. In other words, the leaf labeled with $\square$ is annotated by q. We use $C(q_1) \stackrel{r}{\Longrightarrow}_A q_2$ to denote that r is a run of A on C from q_1 such that $r(\epsilon) = q_2$. The notation $C(q_1) \Longrightarrow_A q_2$ and the extension to sets of states are explained in a similar manner to runs on trees.

A context $C = (S_C, \lambda_C)$ over $\Sigma^\bullet(2)$ is said to be *copying* if for each $n \in S_C$ with $n \neq n_c$ we have $\lambda_C(n) = (f, f)$ for some $f \in \Sigma$. In other words, the context corresponds to a copy operation on all its nodes.

For an automaton $A = (Q, F, \delta)$ We define the *suffix* of a state $q \in Q$ as follows:

$$\mathit{suff}(q) \;=\; \{C : \text{context} |\ C(q) \Longrightarrow_A F\}$$

For a set $X \subseteq Q$, we define its suffix: $\mathit{suff}(X) = \bigcup_{q \in X} \mathit{suff}(q)$.

Then, we define a state q to be *idempotent* if and only if $\mathit{suff}(q)$ contains only copying contexts. Intuitively, idempotent states denote states from which the transducer only accepts contexts corresponding to a copy operation on nodes.

Saturation Consider a transducer $D = (Q, F, \delta)$, its history transducer $H = (Q_H, F_H, \delta_H)$, and its symbolic transducer $S = (Q_S, F_S, \delta_S)$. Consider $W \subseteq Q_H$ and $X \subseteq Q$. We define the *saturation* of W by X, denoted $\lceil W \rceil_X$ as the smallest set W' containing W and closed under the following two rules for each $q \in X$

- if $w_1 \cdot w_2 \in W'$ then $w_1 \cdot q \cdot w_2 \in W'$.

- if $w_1 \cdot q \cdot q \cdot w_2 \in W'$ then $w_1 \cdot q \cdot w_2 \in W'$.

Let $Q_{idm} \subseteq Q$ to be the set of all idempotent states in Q. For $W \subseteq Q_H$, we use $\lceil W \rceil$ to denote the set $\lceil W \rceil_{Q_{idm}}$. The saturation operation obviously defines an equivalence relation on sets of states of H, and therefore also the states Q_S of the symbolic transducer S. This allows us to derive a new transducer S_{SAT} by merging all equivalent states in S. We can achieve that by changing the termination condition of Algorithm 2 (line 8), so that the algorithm stops if all new states generated are equivalent to the previous ones. Notice that this guarantees termination in case the number of equivalence classes is finite. The following theorem states that saturation does not affect the relation recognized by the symbolic transducer.

Theorem 6.1 $R(S) = R(S_{SAT})$.

We devote the rest of this subsection to the proof of Theorem 6.1 (achieved through Lemmas 6.2 to 6.4).

The following lemma states that the saturation operation does not add any element to the suffix of a set of states in H.

Lemma 6.2 Let $w_1, w_2 \in Q_H$ and let $q \in Q$ be an idempotent state.

- $\mathit{suff}(w_1 \cdot q \cdot w_2) \subseteq \mathit{suff}(w_1 \cdot w_2)$.
- $\mathit{suff}(w_1 \cdot q \cdot w_2) \subseteq \mathit{suff}(w_1 \cdot q \cdot q \cdot w_2)$.

From Lemma 6.2 we get the next lemma stating that equivalent states have identical suffixes.

Lemma 6.3 For $W_1, W_2 \subseteq Q_H$, if $\lceil W_1 \rceil = \lceil W_2 \rceil$ then $\mathit{suff}(W_1) = \mathit{suff}(W_2)$.

A consequence of Lemma 6.3 follows in the next lemma. This lemma states that the equivalence relation we consider is in fact a congruence. In the proof of the lemma, we will assume that S contains no useless states (ϕ_u with $\mathit{suff}(\phi_u) = \emptyset$) since these states do not change the language recognized by S and can be removed.

Lemma 6.4 If $\left((\phi_1, \ldots, \phi_i, \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow} \phi\right) \in \delta_S$, and $\mathit{suff}(\phi_i) = \mathit{suff}(\phi'_i)$ for some $i : 1 \leq i \leq p$, then there exists ϕ' such that $\mathit{suff}(\phi) = \mathit{suff}(\phi')$ and $\left((\phi_1, \ldots, \phi'_i, \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow} \phi'\right) \in \delta_S$.

The fact that our equivalence relation is a congruence (Lemma 6.4) implies that we can apply the extension of the MyHill-Nerode theorem to trees (described in [CDG$^+$99]).

Syntactic Characterization of Idempotent States

Consider a transducer $D = (Q, F, \delta)$, and a state $q \in Q$ that we want to check for idempotency.

Input:

Tree Transducer $D = (Q, F, \delta)$, State q.

Begin

$up := \{q\}, side := \emptyset$

Repeat

for each $q_u \in up$, if q_u appears in the left-hand side of a rule $r \in \delta$, then

(i) add the right-hand side state of r to up;

(ii) add all states in the left-hand side of r that are different from q_u to $side$.

Until no more addition can be made.

Repeat

for each $q_d \in side$, if q_d appears as right-hand side of a rule $r \in \delta$, then

add all states in the left-hand side of r to $side$.

Until no more addition can be made.

End

Output: "Yes" if all rules r encountered were copy rules and at least one final state is in up, and "No" otherwise.

Figure 3: Determining Whether A State Is Idempotent

In the case of word automata, this check can be done by simply computing the closure of reachable states from q, and then making sure that all of the reachable states are copying states.

For tree automata, we need to keep in mind that suffixes are in fact contexts. This means that we need to perform our closure not only upward but also *sideward* from q.

Algorithm 3 determines whether a given state is idempotent (see Figure 3).

Example 6.5 When we run Algorithm 2 on our token tree protocol, we get expressions like $\phi_0 = q_0{}^*$ and $\phi_1 = q_1 q_0{}^*$. Their saturated version (q_2 being the idempotent state) is $\lceil \phi_0 \rceil = (q_0 + q_2)^*$ and $\lceil \phi_1 \rceil = q_2^* \cdot q_1 \cdot (q_0 + q_2)^*$.

□

7 Termination

As noted earlier, saturation enables us to define an equivalence relation on regular expressions, and thus to collapse several states of S together. However, to have termination of Algorithm 2, we need to make sure that we generate only a finite number of equivalence classes. In this section, we introduce a class of transducers which include all the protocols we consider in this paper, and for which termination is guaranteed.

More precisely, we consider transducers where the set of states can be partitioned into three parts: states whose prefixes only perform copy operations, states which are idempotent i.e. states whose suffixes only perform copy operations (Section 6), and states which perform the changes. For the latter, we require that they satisfy the *finite local depth* property (described below).

To simplify the proofs, we first consider the class of *well-behaved transducers* satisfying the above condition on state partitioning. Then, we consider a number of larger classes of transducers (Remark 7.5), which we obtain by relaxing the conditions on well-behaved transducers. We indicate how to modify the proofs to show that termination is guaranteed even for these larger classes of transducers.

Notice that well-behaviour is just one sufficient condition for termination. The algorithm may still terminate even in cases where the given transducer is not well-behaved.

Through this section, we let $D = (Q, F, \delta)$ be a transducer, and H and S be its history and symbolic transducers. We first need some definitions.

Copying Prefix States For a state $q \in Q$, its prefix is the set of trees:

$$pref(q) = \{T : \text{tree} | \; T \Longrightarrow_D q\}$$

We now define the notion of *copying tree*: a tree $T = (S, \lambda)$ is copying if for each node $n \in S$ we have $\lambda(n) = (f, f)$ for some symbol f. We say that q is a *copying prefix state* if $pref(q)$ only contains copying trees.

Local Depth Let $Q_1 \subseteq Q$. For a regular relation $R(D)$ and a natural number k, we say that R has *local depth* k with respect to Q_1 if R satisfies the following condition: Consider any two trees $T = (S, \lambda)$ and $T' = (S, \lambda')$, with $(T, T') \in R^m$. Then, there are trees $T_i = (S, \lambda_i)$ for $i : 0 \leq i \leq m$ such that $T_0 = T$, $T_m = T'$, related by accepting runs $T_i \times T_{i+1} \stackrel{r_i}{\Longrightarrow}_D F$, and for each node $n \in S$, there are at most k different j with $r_j(n) \in Q_1$.

We are now ready to state the conditions that will allow us to ensure termination.

Well-Behaved Transducer A transducer $D = (Q, F, \delta)$ is said to be *well-behaved* if Q contains a single copying prefix state q_{cpy}, a single idempotent state q_{idm}, and $R(D)$ has a finite local depth with respect to $Q \setminus \{q_{cpy}, q_{idm}\}$.

We devote the rest of this section to proving that if the transducer we are considering is *well-behaved*, then Algorithm 2 terminates.

The following lemma means that a state q_{cpy} can only appear as q_{cpy}^* in the regular expressions we generate.

Lemma 7.1 For any regular expression ϕ generated by Algorithm 2, if $w^l \cdot q_{cpy} \cdot w^r \in \phi$ then $w^l \cdot z \cdot w^r \in \phi$ for any $z \in q_{cpy}^*$.

We recall (Theorem 6.1) that we can assume that all sets generated in Algorithm 2 are saturated with q_{idm}. This (together with Lemma 7.1) leads to the the following Lemma.

Lemma 7.2 Let ϕ be a regular expression generated by Algorithm 2. If $\phi \subseteq \{q_{cpy}, q_{idm}\}^*$ then $\lceil \phi \rceil_{\{q_{idm}\}}$ is the union of one or more of the following seven regular expressions:

1. q_{idm}^*
2. q_{idm}^+
3. $(q_{cpy} + q_{idm})^*$
4. $q_{idm} \; (q_{cpy} + q_{idm})^*$
5. $(q_{cpy} + q_{idm})^* \; q_{idm}$
6. $q_{idm} \; (q_{cpy} + q_{idm})^* \; q_{idm}$
7. $(q_{cpy} + q_{idm})^* \; q_{idm} \; (q_{cpy} + q_{idm})^*$

Lemma 7.2 and finite local depth imply that we only need to consider regular expressions of a restricted form:

Lemma 7.3 For a well-behaved transducer D with local depth k, Algorithm 2 needs only consider regular expressions of the form

$$\phi_0 \cdot q_1 \cdot \phi_1 \cdot q_2 \cdots \phi_{n-1} \cdot q_n \cdot \phi_n$$

with $n \leq k$, each $\lceil \phi_i \rceil$ is the union of one or more of the seven regular expressions described in Lemma 7.2, and $q_i \notin \{q_{cpy}, q_{idm}\}$.

Consequently, we can conclude that for a well-behaved system, Algorithm 2 terminates.

Theorem 7.4 Algorithm 2 terminates for any well-behaved transducer.

Remark 7.5 We can relax some of the conditions in the definition of well-behaved transiducers, and still guarantee termination of Algorithm 2.

1. It is possible to have several copying prefix states (see e.g. the PERCOLATE example of Section 8). These states need then to obey the following condition. We define an ordering $\prec$ on copying prefix states. For $q_{cpy} \neq q'_{cpy}$, we let $q_{cpy} \prec q'_{cpy}$ denote that δ contains two rules of the forms $(q_1, \ldots, q_p) \xrightarrow{(f,g)} q_{cpy}$ and $(q'_1, \ldots, q'_p) \xrightarrow{(g,h)} q'_{cpy}$, respectively. Observe that q_{cpy} can (immediately) preceed q'_{cpy} in a generated sequence of states only if $q_{cpy} \prec q'_{cpy}$. We can still guarantee termination in case $\prec$ is acyclic. More precisely, we can extend Lemma 7.3 by further dividing each ϕ before applying lemma 7.2. Each part of ϕ will now contain at most one coying prefix state and can be treated as in Lemma 7.3. Observe that acyclicity of $\prec$ can be checked syntactically from the set of rules δ.

2. It is possible to extend D with a bottom and/or top part which is a set of deterministic and copying rules operating on a set of fresh states, with the exception of a unique state to connect the new part to the existing rules; the reason is that such a part will only generate regular expressions of the form q^*. In the special case of words, this class of systems amounts to those covered in [BJNT00]. In fact, none of the examples we consider in Section 8 needs this extension.

Example 7.6 In Example 4.3, we have $q_{cpy} = q_0$ and $q_{idm} = q_2$. The local depth of $R(D)$ with respect to $\{q_1\}$ is 1.

In the PERCOLATE example of Section 8, the copying prefix states are q_u, q_0, q_1, with $q_u \prec q_0$ and $q_u \prec q_1$. Observe that $\prec$ is acyclic. There is a single idempotent state, namely q_d.

□

8 Experimental Results

We have implemented a prototype based on Algorithm 2 and its modifications described in Section 6 and Section 7. In this section we report the result of running our prototype on a number of protocols.

For each example, we model the behaviour of the protocol as a set of transitions described by a transducer D. We use Algorithm 2 to produce a transducer corresponding to the reflexive transitive closure $R^*(D)$. Also, once the algorithm has terminated, we apply standard techniques for minimization of tree automata (described e.g. in [CDG+99]) to reduce the number of states and transitions in $R^*(D)$.

Furthermore, for each protocol, we show correctness of the protocol with respect to some given safety property. More precisely, we provide two tree automata A_{init} and A_{bad}, characterizing a set of *initial* states, and a set of *bad* states which we do not want to occur during the execution of the protocol. Correctness then amounts to showing emptiness of the set

$$((L(A_{init}) \times T(\Sigma)) \ \cap \ R^*(D))|_1 \ \cap \ L(A_{bad})$$

where Σ is the alphabet.

1. Simple Token Protocol In Appendix B, we show the result of running the algorithm on the token tree protocol (Example 4.3). We also show how we have verified a safety property of the protocol.

2. Two-Way Token Protocol This example is a generalization of the previous one. Here, we allow the token to move downwards as well as upwards.

As in Example 4.3, the alphabet we use is $\Sigma = \{t, n, T, N\}$, and the states are $Q = \{q_0, q_1, q_2, q_3\}$. The accepting state is $F = \{q_2\}$.

Intuitively, these states correspond to the following

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

The transition relation δ is given by:

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

For this example, q_2 is the idempotent state, and q_0 is the copying prefix state.

We observe that for any two related trees, we can construct a sequence of intermediate trees such that, for each node, the token passes through the node (upwards or downwards) only once. Thus, the local depth with respect to $\{q_1, q_3\}$ is 1.

In Appendix B we show the result of applying our algorithm to compute the closure of the transition relation. The resulting transducer has 4 states and 20 transitions.

3. The Percolate Protocol The protocol PERCOLATE, described in [KMM$^+$97], operates on a tree of processes. Each process has a local variable with values {0,1} for the leaf nodes and {U,0,1} for internal nodes[1], (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 labelled with 0; do not make any change to them;

q_1 the current node and at least one node below is labelled with 1. No node below is labelled 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;

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

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

In this example, the copying prefix states are $\{q_u, q_1, q_0\}$, while the idempotent state is q_d. We notice that $q_u \prec q_0$ and $q_u \prec q_1$.

The details of applying our algorithm to the protocol is included in Appendix B. The resulting transducer has 3 states and 19 rules.

4. Tree Arbiter The tree arbiter protocol [ABH$^+$97] operates on a tree of processes and aims at preserving mutual exclusion. The leaf nodes try to access a shared resource, while the interior nodes are used to manage the resource. Access to the resource is represented by a token.

A leaf process can be labelled in either of following three ways – *idle*: the process does not do anything; *requesting*: the process wants to access the shared resource; and *token*: the process has been granted access to the resource.

An interior node can be labelled in either of following four ways – *idle*: the process together with all nodes below are idle; *requesting*: at least one child is requesting; *token*: token belongs to this node; and *below*: the token is somewhere in one subtree below this node (but not in the node itself).

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

When a leaf becomes *requesting*, the request is propagated upwards until it encounters a node which is aware of the presence of the token (i.e. a node in state *token* or *below*). If a node has the token, it can always pass it upwards, or pass it downwards to a child which is *requesting*.

Formally, we model the protocol using the alphabet $\Sigma = \{i, r, t, b\}$. As in the previous example, we use some symbols (namely i, r, t) both with arity 0 and arity 2. Symbol b exists only with arity 2.

The states used in our transducer are $Q = \{q_i, q_{req}, q_{rel}, q_{granted}, q_{comp}\}$. Their meaning is the following:

- q_i every node up to the current one are *idle*;
- q_{req} the current node is requesting the token for itself or on behalf of a child;
- q_{rel} the token is moving upwards from the current node;
- $q_{granted}$ the token is moving downwards to the current node;
- q_{comp} the token is either in this node or below it, i.e., nothing happens above the current node (accepting state).

The transition relation δ is given by

$$
\begin{array}{rrrr}
\xrightarrow{(i,i)} q_i & \xrightarrow{(i,r)} q_{req} & \xrightarrow{(r,r)} q_{req} & \xrightarrow{(r,t)} q_{grant} \\
\xrightarrow{(t,i)} q_{rel} & (q_i, q_i) \xrightarrow{(i,i)} q_i & (q_{req}, q_i) \xrightarrow{(i,r)} q_{req} & (q_i, q_{req}) \xrightarrow{(i,r)} q_{req} \\
(q_{req}, q_{req}) \xrightarrow{(i,r)} q_{req} & (q_{req}, q_i) \xrightarrow{(r,r)} q_{req} & (q_i, q_{req}) \xrightarrow{(r,r)} q_{req} & (q_{req}, q_{req}) \xrightarrow{(r,r)} q_{req} \\
(q_{req}, q_i) \xrightarrow{(r,t)} q_{grant} & (q_i, q_{req}) \xrightarrow{(r,t)} q_{grant} & (q_{req}, q_{req}) \xrightarrow{(r,t)} q_{grant} & (q_{grant}, q_i) \xrightarrow{(t,b)} q_{comp} \\
(q_i, q_{grant}) \xrightarrow{(t,b)} q_{comp} & (q_{grant}, q_{req}) \xrightarrow{(t,b)} q_{comp} & (q_{req}, q_{grant}) \xrightarrow{(t,b)} q_{comp} & (q_i, q_i) \xrightarrow{(t,i)} q_{rel} \\
(q_i, q_{req}) \xrightarrow{(t,i)} q_{rel} & (q_{req}, q_i) \xrightarrow{(t,i)} q_{rel} & (q_{req}, q_{req}) \xrightarrow{(t,i)} q_{rel} & (q_{rel}, q_i) \xrightarrow{(b,t)} q_{comp} \\
(q_i, q_{rel}) \xrightarrow{(b,t)} q_{comp} & (q_{rel}, q_{req}) \xrightarrow{(b,t)} q_{comp} & (q_{req}, q_{rel}) \xrightarrow{(b,t)} q_{comp} & (q_{comp}, q_i) \xrightarrow{(b,b)} q_{comp} \\
(q_i, q_{comp}) \xrightarrow{(b,b)} q_{comp} & (q_{comp}, q_{req}) \xrightarrow{(b,b)} q_{comp} & (q_{req}, q_{comp}) \xrightarrow{(b,b)} q_{comp} &
\end{array}
$$

The details of applying our algorithm to the protocol is included in Appendix B. The resulting transducer has 5 states and 66 rules.

9 Conclusions and Future Work

We have presented an approach to extend the applicability of model checking of parameterized systems. In contrast to earlier approaches which mainly operate on linear topologies, our method deals with systems which have tree-like structures. The approach is based on a method for computing transitive closures for regular tree relations. We have implemented a prototype based on our algorithm and we report on computing transitive closure of the transition relation and verification of a number of protocols.

An important direction for future research is to generalize the approach to other classes. Since our algorithm relies on a small set of properties of trees which are shared by other data structures, we believe that our approach can be lifted to a more general setting. In particular we aim to develop model checking algorithms for systems whose behaviours are captured by relations on general forms of graphs. This would allow us to verify systems with dynamic behaviours such as data security protocols.

Acknowledgments

This work was supported in part by the European Commission (FET project ADVANCE, contract No IST-1999-29082).

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. 9^{th} Int. Conf. on Computer Aided Verification*, volume 1254, pages 340–351, Haifa, Israel, 1997. Springer Verlag.

[ABJN99] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In *Proc. 11^{th} Int. Conf. on Computer Aided Verification*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145, 1999.

[APR+01] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L. Zuck. Parameterized verification with automatically computed inductive assertions. In *Proc. 13^{th} Int. Conf. on Computer Aided Verification*, pages 221–234, 2001.

[BCMD92] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

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

[BMT01] A. Bouajjani, A. Muscholl, and T. Touili. Permutation Rewriting and Algorithmic Verification. In *Proc. 16th Symp. on Logic in Computer Science (LICS'01)*, Boston (MA), USA, 2001. IEEE.

[CDG+99] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications.* October 1999.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[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.

[FP01] Dana Fisman and Amir Pnueli. Beyond regular model checking. In *Proc. 21th Conference on the Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, December 2001. To appear.

[JN00] Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS '00, 6^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, 2000.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc. 9^{th} Int. Conf. on Computer Aided Verification*, volume 1254, pages 424–435, Haifa, Israel, 1997. Springer Verlag.

[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.

[Mai01] M. Maidl. A unifying model checking approach for safety properties of parameterized systems. In *Proc. 13^{th} Int. Conf. on Computer Aided Verification*, pages 324–336, 2001.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[PRZ01] A. Pnueli, S. Ruah, and L. Zuck. Automatic deductive verification with invisible invariants. In *Proc. TACAS '01, 7^{th} Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031, pages 82–97, 2001.

[QS82] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *5th International Symposium on Programming, Turin*, volume 137 of *Lecture Notes in Computer Science*, pages 337–352. Springer Verlag, 1982.

[Tou01] T. Touili. Regular Model Checking using Widening Techniques. *Electronic Notes in Theoretical Computer Science*, 50(4), 2001. Proc. Workshop on Verification of Parametrized Systems (VEPAS'01), Crete, July, 2001.

[WB98] Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97, Vancouver, July 1998. Springer Verlag.

Appendix A – Proofs of Lemmas

Lemma 5.1

Let $D = (Q, F, \delta)$ and $H = (Q_H, F_H, \delta_H)$.

$\subseteq$

Consider $T = (S, \lambda)$ and $T' = (S, \lambda')$, with $(T, T') \in R(H)$. Then, there is an r such that $(T \times T') \xRightarrow{r}_H F_H$. By definition of a run and the form of transition rules in history transducers, we know that there is $k \geq 0$ such that for each node $n \in (T \times T')$ with children $n_1, \ldots, n_p$ and $\lambda(n) = (f, f')$:

- $|r(n_1)| = \cdots = |r(n_p)| = |r(n)| = k$.
- there are $f_1^n, f_2^n, \ldots, f_{k+1}^n$, with $f_1^n = f$, $f_{k+1}^n = f'$, and $(q_1(i) \ldots, q_p(i)) \stackrel{(f_i^n, f_{i+1}^n)}{\longrightarrow} q(i)$ belongs to δ.

Define $T_i = (S, \lambda_i)$, where $\lambda_i(n) = f_i^n$. Define r_i such that $r_i(n) = r(n)(i)$. From the definition of a run, it follows that $(T_i \times T_{i+1}) \xRightarrow{r_i}_D F$, and hence $(T_i, T_{i+1}) \in R(D)$ for $i : 1 \leq i \leq k$. This means that $(T, T') \in (R(D))^k$.

$\supseteq$

Consider $T = (S, \lambda)$ and $T' = (S, \lambda')$, with $(T, T') \in (R(D))^k$. Then, there are $T_1, \ldots, T_{k+1}$ with $T_i = (S, \lambda_i)$, $T_1 = T$, $T_{k+1} = T'$, and $(T_i, T_{i+1}) \in R(D)$ for $i : 1 \leq i \leq k$. Since $(T_i, T_{i+1}) \in R(D)$, we know that there is an r_i such that $(T_i, T_{i+1}) \xRightarrow{r_i}_D F$. By definition of a run, we know that for any node $n \in S$ with children $n_1, \ldots, n_p$

$$\left((r_i(n_1), \ldots, r_i(n_p)) \stackrel{(\lambda_i(n), \lambda_{i+1}(n))}{\longrightarrow} r_i(n) \right) \in \delta$$

We define r to be the mapping such that for any $n \in S$, $r(n) = r_0(n) \cdots r_k(n)$.

By definition, it follows that $(T, T') \xRightarrow{r}_H F_H$, and hence $(T, T') \in R(H)$.

$\square$

Lemma 5.2

Consider $R_1 = [\{(g, g', q_1, \ldots, q_p, q) : (q_1, \ldots, q_p) \stackrel{g, g'}{\longrightarrow} q \in \delta\}^+]$. R_1 is a regular word relation on $\Sigma \times \Sigma \times Q_H \times \ldots \times Q_H$.

Let $R_1' = \{(f, f, \epsilon, \ldots, \epsilon) : f \in \Sigma\}$, the set of tuples with $p + 1$ ϵ's if f has arity p, which is also a regular relation on $\Sigma \times \Sigma \times Q_H \times \ldots \times Q_H$.

Let R_2 be all pairs of words $\left(f_1 f_2 \ldots f_n, {f_1}' {f_2}' \ldots {f_n}' \right)$ such that $f = f_1, f' = f_n$ and ${f_{i+1}}' = f_i$ for all $1 \leq i < n$. R_2 is also regular.

$$\Delta_{(f,f')} = \left((R_1 \cup R_1') \cap (R_2 \times Q_H \times \ldots \times Q_H) \right) |_{\{1,2\}}$$

is thus regular.

The set $(\phi_1, \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow}$ is computed by composing $\Delta_{(f,f')}$ with each of the ϕ_i, thus obtaining a regular set. □

Lemma 5.3

- Suppose that $(T \times T') \stackrel{r}{\Longrightarrow}_S \phi$, with $(T \times T') = (S, \lambda)$. Take $w \in \phi$. We show that there is r' with $(T \times T') \stackrel{r'}{\Longrightarrow}_H w$. We use induction on $|S|$.

Base Case ($S = \{\epsilon\}$). Let $\lambda(\epsilon) = (f, f')$. This means that $\left(\stackrel{(f,f')}{\longrightarrow} \phi\right) \in \delta_S$. By Algorithm 2 we know that $\left(\stackrel{(f,f')}{\longrightarrow} w\right) \in \delta_H$. We define r' such that $r'(\epsilon) = w$. We observe that $(T \times T') \stackrel{r'}{\Longrightarrow}_H w$.

Induction Step Let $\lambda(\epsilon) = (f, f')$. Let $n_1, \ldots, n_p \in (T \times T')$ be the children of ϵ (i.e. the children of the root). Let $(T_1 \times T_1'), \ldots, (T_p \times T_p')$ be the subtrees of $(T \times T')$ at $n_1, \ldots, n_p$ respectively. Since $(T \times T') \stackrel{r}{\Longrightarrow}_S \phi$, we know by definition of a run that there are $\phi_1, \ldots, \phi_p$ and $r_1, \ldots, r_p$ such that $(T_i \times T_i') \stackrel{r_i}{\Longrightarrow}_S \phi_i$ for $i : 1 \leq i \leq p$, and $\left((\phi_1, \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow} \phi\right) \in \delta_S$. By Algorithm 2 we know that there are $w_1 \in \phi_1, \ldots, w_p \in \phi_p$ with $\left((w_1, \ldots, w_p) \stackrel{(f,f')}{\longrightarrow} w\right) \in \delta_H$. By the induction hypothesis it follows that there are $r_1', \ldots, r_p'$ with $(T_i \times T_i') \stackrel{r_i'}{\Longrightarrow}_H w_i$ for $i : 1 \leq i \leq p$. Define r' such that $r'(i \cdot n) = r_i'(n)$ for $i : 1 \leq i \leq p$ and $n \in (T_i \times T_i')$, and $r'(\epsilon) = w$. We observe that $(T \times T') \stackrel{r'}{\Longrightarrow}_H w$.

- Suppose that $(T \times T') \stackrel{r}{\Longrightarrow}_H w$, with $(T \times T') = (S, \lambda)$. We show that there is ϕ and r' with $w \in \phi$ and $(T \times T') \stackrel{r'}{\Longrightarrow}_S \phi$. We use induction on $|S|$.

Base Case ($S = \{\epsilon\}$). Let $\lambda(\epsilon) = (f, f')$. This means that $\left(\stackrel{(f,f')}{\longrightarrow} w\right) \in \delta_H$. By Algorithm 2 we know that there is ϕ with $w \in \phi$ and $\left(\stackrel{(f,f')}{\longrightarrow} \phi\right) \in \delta_S$. We define r' such that $r'(\epsilon) = \phi$. We observe that $(T \times T') \stackrel{r'}{\Longrightarrow}_S \phi$.

Induction Step Let $\lambda(\epsilon) = (f, f')$. Let $n_1, \ldots, n_p \in (T \times T')$ be the children of ϵ (i.e. the children of the root). Let $(T_1 \times T_1'), \ldots, (T_p \times T_p')$ be the subtrees of $(T \times T')$ at $n_1, \ldots, n_p$ respectively. Since $(T \times T') \stackrel{r}{\Longrightarrow}_H w$, we know by definition of a run that there are $w_1, \ldots, w_p$ and $r_1, \ldots, r_p$ such that $(T_i \times T_i') \stackrel{r_i}{\Longrightarrow}_H w_i$ for $i : 1 \leq i \leq p$, and $\left((w_1, \ldots, w_p) \stackrel{(f,f')}{\longrightarrow} w\right) \in \delta_H$. By the induction hypothesis it follows that there are $\phi_1, \ldots, \phi_p$ $r_1', \ldots, r_p'$ with $w_i \in \phi_i$ and $(T_i \times T_i') \stackrel{r_i'}{\Longrightarrow}_S \phi_i$ for $i : 1 \leq i \leq p$. By Algorithm 2 we know that $\left((\phi_1, \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow} \phi\right) \in \delta_S$ for some ϕ with $w \in \phi$.

Define r' such that $r'(i \cdot n) = r_i'(n)$ for $i : 1 \leq i \leq p$ and $n \in (T_i \times T_i')$, and $r'(\epsilon) = \phi$. We observe that $(T \times T') \stackrel{r'}{\Longrightarrow}_S \phi$. □

Lemma 6.2

- Let $k = |w_1|$. Let $C \in \mathrm{suff}(w_1.q.w_2)$ be a context and r a run such that $C(w_1.q.w_2) \stackrel{r}{\Longrightarrow}_H$

F_H.

We define a run r' as follows: for each node $n \in C$, $r'(n)$ is a copy of $r(n)$ with the $(k+1)$-th position removed.

Because q is idempotent, $r(n)$ will have its $(k+1)$-th position be a copying state. Thus, r' satisfies $C(w_1.w_2) \stackrel{r'}{\Longrightarrow}_H F_H$.

Thus $\text{suff}(w_1.q.w_2) \subseteq \text{suff}(w_1.w_2)$.

- Let $k = |w_1|$. Let $C \in \text{suff}(w_1.q.w_2)$ be a context and r a run such that $C(w_1.q.w_2) \stackrel{r}{\Longrightarrow}_H F_H$.

Similarly here, we take r, and for each node $n \in C$ we define $r'(n)$ to be $r(n)$ with the $(k+1)$-th position duplicated. Since q is idempotent, all the duplicated states are copying states. thus r' will satisfy $C(w_1.q.q.w_2) \stackrel{r'}{\Longrightarrow}_H F_H$.

Thus $\text{suff}(w_1.q.w_2) \subseteq \text{suff}(w_1.q.q.w_2)$. □

Lemma 6.3

By hypothesis, $\lceil W_1 \rceil = \lceil W_2 \rceil$, i.e. $\text{suff}(\lceil W_1 \rceil) = \text{suff}(\lceil W_2 \rceil)$. From Lemma 6.2, we know that $\text{suff}(\lceil W_1 \rceil) = \text{suff}(W_1)$ and similarly $\text{suff}(\lceil W_2 \rceil) = \text{suff}(W_2)$. Hence the result. □

For the proof of Lemma 6.4, we first show a sublemma which states that if two rules of S have in their left-hand side regular expressions that are equivalent, then their right-hand side have also equivalent expressions.

Lemma .1 If $\left((\phi_1, \ldots, \phi_i, \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow} \phi\right) \in \delta_S$, and $\left((\phi_1, \ldots, \phi_i', \ldots, \phi_p) \stackrel{(f,f')}{\longrightarrow} \phi'\right) \in \delta_S$, and $\textit{suff}\,(\phi_i) = \textit{suff}\,(\phi_i')$, then $\textit{suff}\,(\phi) = \textit{suff}\,(\phi')$.

Proof Without loss of generality, we can assume $i = 1$ for this proof.

- $\textit{suff}\,(\phi) \subseteq \textit{suff}\,(\phi')$

Consider any context $C \in \textit{suff}\,(\phi)$ (from the assumption that there no useless states, we know there is at least one).

Now consider trees $T_2, \ldots, T_p$ such that $T_i \Longrightarrow_S \phi_i$ for each $i : 2 \leq i \leq p$. From Algorithm 2, we know that these trees exist.

Let $C = (S, \lambda)$ and $T_i = (S_i, \lambda_i)$ for each $i : 2 \leq i \leq p$. Consider the new context $C_1 = (S_1, \lambda_1)$ defined as follows. We let $S_1 = S \cup \{n_c \cdot 1\} \cup \{n_c \cdot i \cdot n |\ 2 \leq i \leq p \text{ and } n \in S_i\}$. For any node $n_1 \in S_1$, we let $\lambda_1(n_1)$ be:

- if $n_1 \in S$ and $n_1 \neq n_c$, then $\lambda_1(n_1) = \lambda(n_1)$;
- if $n_1 \in S$ and $n_1 = n_c$, then $\lambda_1(n_1) = (f, f')$;
- if $n_1 = n_c \cdot 1$, then $\lambda_1(n_1) = \Box$;
- if $n_1 = n_c \cdot i \cdot n$ with $n \in S_i$, for some $i : 2 \leq i \leq p$, then $\lambda_1(n_1) = \lambda_i(n)$.

More intuitively, we create a context with (f, f') at its root, and trees $T_2, \ldots, T_p$ as children. The remaining child is the hole. We "plug" this new context into the hole of C, and get C_1.

We observe that $C_1 \in \mathit{suff}(\phi_1)$. Since $\mathit{suff}(\phi_1) = \mathit{suff}(\phi'_1)$, we know that $C_1 \in \mathit{suff}(\phi'_1)$. By hypothesis, we have $\left((\phi'_1, \phi_2, \ldots, \phi_p) \xrightarrow{(f,f')} \phi'\right) \in \delta_S$. By Algorithm 2, S is deterministic, and we can conclude that $C \in \mathit{suff}(\phi')$.

- $\mathit{suff}(\phi) \supseteq \mathit{suff}(\phi')$

Similar.

$\square$

Lemma 6.4

Without loss of generality, we can assume $i = 1$ for this proof. Consider any context $C \in \mathit{suff}(\phi)$ (from the assumption that there are no useless states, we know there is at least one).

Now consider trees $T_2, \ldots, T_p$ such that $T_i \Longrightarrow_S \phi_i$ for each $i : 2 \leq i \leq p$. From Algorithm 2, we know that these trees exist.

Consider the new context C_1 constructed exactly as in the proof of Lemma .1. We observe that $C_1 \in \mathit{suff}(\phi_1)$. Since $\mathit{suff}(\phi_1) = \mathit{suff}(\phi'_1)$, we know that $C_1 \in \mathit{suff}(\phi'_1)$. I.e. there is a run r_1 with $C_1(\phi'_1) \overset{r_1}{\Longrightarrow}_S F_S$. By definition of a run, this means that there is ϕ' and a rule $\left((\phi'_1, \phi_2, \ldots, \phi_p) \xrightarrow{(f,f')} \phi'\right) \in \delta_S$.

We apply Lemma .1, and we conclude that $\mathit{suff}(\phi) = \mathit{suff}(\phi')$. $\square$

Lemma 7.1

We prove the claim by induction.

Base case Let f, f' be symbols of null arity. We suppose that we generate a rule in δ_S:

$$\xrightarrow{(f,f')} \phi$$

such that $w = w^l.q.w^r \in \phi$. Since q is a copying prefix state, we know that δ contains only rules of the form $\xrightarrow{(g,g)} q$. Hence, we can conclude that both $w^l.w^r$ and $w^l.q.q.w^r$ are in ϕ as well.

Induction step

We suppose that we generate a rule in δ_S:

$$(\phi_1, \ldots, \phi_p) \xrightarrow{(f,f')} \phi$$

such that ϕ_1 through ϕ_p respect the hypothesis, and $w = w^l.q.w^r \in \phi$.

By definition, there exist $w_1 \in \phi_1, \ldots, w_p \in \phi_p$ words such that $(w_1, \ldots, w_p) \xrightarrow{(f,f')} w \in \delta_H$.

Since q is a copying prefix state, we know that each of the p children of w can be divided as follows: for any $i : 1 \leq i \leq p$, $w_i = w_i^l.q_i.w_i^r$ for some $q_i \in Q_B$. We must also have $(q_1, \ldots, q_p) \xrightarrow{(g,g)} q \in \delta$ for some symbol g.

By hypothesis, we know that ϕ_i also contains the words $w_i^l.q_i \cdot q_i.w_i^r$.

We thus conclude that both $w^l.w^r$ and $w^l.q.q.w^r$ are in ϕ as well. $\square$

Lemma 7.2

We show that for each $w \in \phi$ there is a regular expression ϕ_w of one of the seven form described in the Lemma, such that $w \in \phi_w \subseteq \phi$. We consider the following seven exhaustive cases. In each case, the claim follows from Theorem 6.1 and Lemma 7.1.

1. If $w = \epsilon$. Define $\phi_w = q_{idm}^*$.
2. If $w = q_{idm}^j$, for some $j \geq 1$ Define $\phi_w = q_{idm}^+$.
3. If $w = q_{cpy}^j$, for some $j \geq 1$ Define $\phi_w = (q_{cpy} + q_{idm})^*$.
4. If $w = q_{idm} \cdot w_1 \cdot q_{cpy}$, for some w_1 Define $\phi_w = q_{idm}\ (q_{cpy} + q_{idm})^*$.
5. If $w = q_{cpy} \cdot w_1 \cdot q_{idm}$, for some w_1 Define $\phi_w = (q_{cpy} + q_{idm})^*\ q_{idm}$.
6. If $w = q_{cpy} \cdot w_1 \cdot q_{cpy}$, for some w_1 Define $\phi_w = (q_{cpy} + q_{idm})^*\ q_{idm}\ (q_{cpy} + q_{idm})^*$.
7. If $w = q_{idm} \cdot w_1 \cdot q_{idm}$, for some w_1 Define $\phi_w = q_{idm}\ (q_{cpy} + q_{idm})^*\ q_{idm}$.

$\square$

Lemma 7.3

The definition of local depth assures us that we need only consider sequences that contain up to k states of $Q \setminus \{q_{cpy}, q_{idm}\}$.

We show here that discarding sequences with more than k states from $Q \setminus \{q_{cpy}, q_{idm}\}$ gives us the expected result.

Consider a generated regular expression ϕ. To keep only sequences that include states q_1 to q_n, with $n \leq k$, we compute

$$\phi' = \phi \cap (\{q_{cpy}, q_{idm}\}^* \cdot q_1 \cdot \{q_{cpy}, q_{idm}\}^* \cdot q_2 \cdots q_n \cdot \{q_{cpy}, q_{idm}\}^*)$$

which gives a regular expression of the form

$$\phi_0 \cdot q_1 \cdot \phi_2 \cdot q_2 \cdots q_n \cdot \phi_n$$

Each ϕ_i satisfies $\phi_i \subseteq \{q_{cpy}, q_{idm}\}^*$, and we can apply lemma 7.2 on it, which implies the result. $\square$

Appendix B – Experimental Data

1. Simple Token Protocol When running the algorithm on the token tree protocol (Example 4.3), we get the following transducer describing the reflexive transitive closure of the transition relation. The transducer has 4 states $\phi_0, \phi_1, \phi_2, \phi_3$ and 20 rules.

The transition relation δ_S is given by :

$$
\begin{array}{llll}
\xrightarrow{(n,n)} \phi_0 & \xrightarrow{(t,n)} \phi_1 & \xrightarrow{(t,t)} \phi_2 & \xrightarrow{(t,t)} \phi_3 \\
(\phi_0,\phi_0) \xrightarrow{(N,N)} \phi_0 & (\phi_0,\phi_1) \xrightarrow{(N,N)} \phi_1 & (\phi_1,\phi_0) \xrightarrow{(N,N)} \phi_1 & (\phi_0,\phi_2) \xrightarrow{(N,N)} \phi_2 \\
(\phi_2,\phi_0) \xrightarrow{(N,N)} \phi_2 & (\phi_0,\phi_1) \xrightarrow{(N,T)} \phi_2 & (\phi_1,\phi_0) \xrightarrow{(N,T)} \phi_2 & (\phi_0,\phi_0) \xrightarrow{(T,N)} \phi_1 \\
(\phi_0,\phi_0) \xrightarrow{(T,T)} \phi_2 & (\phi_0,\phi_0) \xrightarrow{(T,T)} \phi_3 & (\phi_0,\phi_3) \xrightarrow{(N,N)} \phi_3 & (\phi_3,\phi_0) \xrightarrow{(N,N)} \phi_3 \\
(\phi_3,\phi_3) \xrightarrow{(N,N)} \phi_3 & (\phi_0,\phi_3) \xrightarrow{(T,T)} \phi_3 & (\phi_3,\phi_0) \xrightarrow{(T,T)} \phi_3 & (\phi_3,\phi_3) \xrightarrow{(T,T)} \phi_3
\end{array}
$$

where the (saturated) symbolic states are given by

$\phi_0 = (q_0 + q_2)^*$, $\phi_1 = q_2^* \cdot q_1 \cdot (q_0 + q_2)^*$, $\phi_2 = q_2^+$, and $\phi_3 = q_2^*$.

For verification of the protocol, we take the set of initial states to be described by the following tree automaton, where q_1 is the accepting state.

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

Intuitively, the above tree automaton recognizes all trees with exactly one token in the tree.

The set of bad states is characterized by the following tree automaton, where q_2 is the accepting state.

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

Intuitively, this automaton recognizes all trees with at least tow tokens in the tree. This means that our algorithm verifies the mutual exclusion property for the protocol on trees of all sizes.

We have analogously carried out verification of similar safety properties for the subsequent protocols. We will include the details here.

2. Two-Way Token Protocol We show the result of applying our algorithm to the Two-Way Token Protocol (Section 8). The symbolic transitions we get are given by

$$
\begin{array}{llll}
\xrightarrow{(n,n)} \phi_0 & \xrightarrow{(t,n)} \phi_1 & \xrightarrow{(n,t)} \phi_3 & \xrightarrow{(t,t)} \phi_2 \\
(\phi_0,\phi_0) \xrightarrow{(N,N)} \phi_0 & (\phi_0,\phi_1) \xrightarrow{(N,N)} \phi_1 & (\phi_1,\phi_0) \xrightarrow{(N,N)} \phi_1 & (\phi_0,\phi_2) \xrightarrow{(N,N)} \phi_2 \\
(\phi_2,\phi_0) \xrightarrow{(N,N)} \phi_2 & (\phi_0,\phi_3) \xrightarrow{(N,N)} \phi_3 & (\phi_3,\phi_0) \xrightarrow{(N,N)} \phi_3 & (\phi_1,\phi_3) \xrightarrow{(N,N)} \phi_2 \\
(\phi_3,\phi_1) \xrightarrow{(N,N)} \phi_2 & (\phi_0,\phi_0) \xrightarrow{(T,N)} \phi_1 & (\phi_3,\phi_0) \xrightarrow{(T,N)} \phi_2 & (\phi_0,\phi_3) \xrightarrow{(T,N)} \phi_2 \\
(\phi_0,\phi_1) \xrightarrow{(N,T)} \phi_2 & (\phi_1,\phi_0) \xrightarrow{(N,T)} \phi_2 & (\phi_0,\phi_0) \xrightarrow{(N,T)} \phi_3 & (\phi_0,\phi_0) \xrightarrow{(T,T)} \phi_2
\end{array}
$$

where the (saturated) symbolic states are given by

$\phi_0 = (q_0 + q_2)^*$, $\phi_1 = q_2^* \cdot q_1 \cdot (q_0 + q_2)^*$, $\phi_2 = q_2^*$, and $\phi_3 = (q_0 + q_2)^* \cdot q_3 \cdot q_2^*$.

3. The Percolate Protocol We show the result of applying our algorithm to the PERCOLATE Protocol (Section 8).

The symbolic transitions we get from running our tool is given by (we use ϕ_m as a shorthand to denote any member of $\{\phi_u, \phi_1, \phi_0\}$)

$$
\begin{array}{lll}
\xrightarrow{(0,0)} \phi_0 & \xrightarrow{(1,1)} \phi_1 & (\phi_0,\phi_0) \xrightarrow{(0,0)} \phi_0 \\
(\phi_0,\phi_1) \xrightarrow{(1,1)} \phi_1 & (\phi_1,\phi_0) \xrightarrow{(1,1)} \phi_1 & (\phi_1,\phi_1) \xrightarrow{(1,1)} \phi_1 \\
(\phi_m,\phi_m) \xrightarrow{(U,U)} \phi_u & (\phi_0,\phi_0) \xrightarrow{(U,0)} \phi_0 & (\phi_0,\phi_1) \xrightarrow{(U,1)} \phi_1 \\
(\phi_1,\phi_0) \xrightarrow{(U,1)} \phi_1 & (\phi_1,\phi_1) \xrightarrow{(U,1)} \phi_1 &
\end{array}
$$

where the (saturated) symbolic states are given by

$\phi_0 = (q_u + q_d)^* \cdot (q_0 + q_d)^*$, $\phi_1 = (q_u + q_d)^* \cdot (q_1 + q_d)^*$, $\phi_u = (q_u + q_d)^*$.

4. Tree Arbiter We show the result of applying our algorithm to the Tree Arbiter Protocol (Section 8). When computing the transitive closure of the arbiter, the algorithm gives 6 (saturated) symbolic states and 66 rules. The transition relation for the symbolic transducer is:

$$
\begin{aligned}
&\xrightarrow{(i,i)} \phi_0 \\
&\xrightarrow{(i,t)} \phi_4 \\
&\xrightarrow{(t,t)} \phi_5 \\
(\phi_1, \phi_0) &\xrightarrow{(i,r)} \phi_1 \\
(\phi_1, \phi_1) &\xrightarrow{(r,r)} \phi_1 \\
(\phi_0, \phi_1) &\xrightarrow{(t,r)} \phi_3 \\
(\phi_3, \phi_1) &\xrightarrow{(b,r)} \phi_3 \\
(\phi_2, \phi_1) &\xrightarrow{(b,r)} \phi_3 \\
(\phi_0, \phi_1) &\xrightarrow{(i,t)} \phi_4 \\
(\phi_4, \phi_1) &\xrightarrow{(i,b)} \phi_4 \\
(\phi_4, \phi_1) &\xrightarrow{(r,b)} \phi_4 \\
(\phi_1, \phi_0) &\xrightarrow{(t,t)} \phi_5 \\
(\phi_0, \phi_5) &\xrightarrow{(b,b)} \phi_5 \\
(\phi_2, \phi_4) &\xrightarrow{(b,b)} \phi_5 \\
(\phi_0, \phi_4) &\xrightarrow{(t,b)} \phi_5 \\
(\phi_3, \phi_0) &\xrightarrow{(b,t)} \phi_5 \\
(\phi_0, \phi_3) &\xrightarrow{(b,t)} \phi_5
\end{aligned}
$$

$$
\begin{aligned}
&\xrightarrow{(i,r)} \phi_1 \\
&\xrightarrow{(t,i)} \phi_2 \\
(\phi_0, \phi_0) &\xrightarrow{(i,i)} \phi_0 \\
(\phi_1, \phi_1) &\xrightarrow{(i,r)} \phi_1 \\
(\phi_0, \phi_0) &\xrightarrow{(t,i)} \phi_2 \\
(\phi_1, \phi_0) &\xrightarrow{(t,r)} \phi_3 \\
(\phi_0, \phi_3) &\xrightarrow{(b,r)} \phi_3 \\
(\phi_0, \phi_1) &\xrightarrow{(r,t)} \phi_4 \\
(\phi_1, \phi_0) &\xrightarrow{(i,t)} \phi_4 \\
(\phi_0, \phi_4) &\xrightarrow{(i,b)} \phi_4 \\
(\phi_0, \phi_4) &\xrightarrow{(r,b)} \phi_4 \\
(\phi_1, \phi_1) &\xrightarrow{(t,t)} \phi_5 \\
(\phi_1, \phi_5) &\xrightarrow{(b,b)} \phi_5 \\
(\phi_3, \phi_4) &\xrightarrow{(b,b)} \phi_5 \\
(\phi_1, \phi_4) &\xrightarrow{(t,b)} \phi_5 \\
(\phi_3, \phi_1) &\xrightarrow{(b,t)} \phi_5 \\
(\phi_1, \phi_3) &\xrightarrow{(b,t)} \phi_5
\end{aligned}
$$

$$
\begin{aligned}
&\xrightarrow{(r,i)} \phi_0 \\
&\xrightarrow{(r,t)} \phi_4 \\
(\phi_0, \phi_0) &\xrightarrow{(r,i)} \phi_0 \\
(\phi_0, \phi_1) &\xrightarrow{(r,r)} \phi_1 \\
(\phi_0, \phi_2) &\xrightarrow{(b,i)} \phi_2 \\
(\phi_1, \phi_1) &\xrightarrow{(t,r)} \phi_3 \\
(\phi_1, \phi_3) &\xrightarrow{(b,r)} \phi_3 \\
(\phi_1, \phi_0) &\xrightarrow{(r,t)} \phi_4 \\
(\phi_1, \phi_1) &\xrightarrow{(i,t)} \phi_4 \\
(\phi_1, \phi_4) &\xrightarrow{(i,b)} \phi_4 \\
(\phi_1, \phi_4) &\xrightarrow{(r,b)} \phi_4 \\
(\phi_5, \phi_0) &\xrightarrow{(b,b)} \phi_5 \\
(\phi_4, \phi_2) &\xrightarrow{(b,b)} \phi_5 \\
(\phi_4, \phi_0) &\xrightarrow{(t,b)} \phi_5 \\
(\phi_2, \phi_0) &\xrightarrow{(b,t)} \phi_5 \\
(\phi_0, \phi_2) &\xrightarrow{(b,t)} \phi_5
\end{aligned}
$$

$$
\begin{aligned}
&\xrightarrow{(r,r)} \phi_1 \\
&\xrightarrow{(t,r)} \phi_3 \\
(\phi_0, \phi_1) &\xrightarrow{(i,r)} \phi_1 \\
(\phi_1, \phi_0) &\xrightarrow{(r,r)} \phi_1 \\
(\phi_2, \phi_0) &\xrightarrow{(b,i)} \phi_2 \\
(\phi_3, \phi_0) &\xrightarrow{(b,r)} \phi_3 \\
(\phi_1, \phi_2) &\xrightarrow{(b,r)} \phi_3 \\
(\phi_1, \phi_1) &\xrightarrow{(r,t)} \phi_4 \\
(\phi_4, \phi_0) &\xrightarrow{(i,b)} \phi_4 \\
(\phi_4, \phi_0) &\xrightarrow{(r,b)} \phi_4 \\
(\phi_0, \phi_1) &\xrightarrow{(t,t)} \phi_5 \\
(\phi_5, \phi_1) &\xrightarrow{(b,b)} \phi_5 \\
(\phi_4, \phi_3) &\xrightarrow{(b,b)} \phi_5 \\
(\phi_4, \phi_1) &\xrightarrow{(t,b)} \phi_5 \\
(\phi_2, \phi_1) &\xrightarrow{(b,t)} \phi_5 \\
(\phi_1, \phi_2) &\xrightarrow{(b,t)} \phi_5
\end{aligned}
$$