Regular Tree Model Checking*

Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso

Dept. of Computer Systems. P.O. Box 337, 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.

1 Introduction

Regular model checking has recently been advocated for model checking of *parameterized systems*, i.e. systems whose description is parameterized by the number of components in them (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

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

D. Brinksma and K. G. Larsen (Eds.): CAV 2002, LNCS 2404, pp. 555–568, 2002.

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, PS00].

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.

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 $\{(w_1, \ldots, w_{i-1}, w_{i+1}, \ldots, w_m) |\ (w_1, w_2, \ldots, w_m) \in R\}$. 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 the standard notation (e.g. found in [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 $S_n = \{b |\ nb \in S\}$ 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 $(T_1, \cdots, T_m, T_1', \cdots, T_n')$ such that $(T_1, \cdots, T_m) \in R$, $(T_1', \cdots, T_n') \in R'$, 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 $R^+ = \cup_{i \geq 1} R^i$.

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

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 1. *Regularity is closed under composition.*

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

$\xrightarrow{0} q_0 \quad \xrightarrow{1} q_1 \quad (q_0, q_0) \xrightarrow{or} q_0 \quad (q_0, q_1) \xrightarrow{or} q_1 \quad (q_1, q_0) \xrightarrow{or} q_1$
$(q_1, q_1) \xrightarrow{or} q_1 \quad (q_0, q_0) \xrightarrow{and} q_0 \quad (q_0, q_1) \xrightarrow{and} q_0 \quad (q_1, q_0) \xrightarrow{and} q_0 \quad (q_1, q_1) \xrightarrow{and} q_1$

B recognizes the tree language which is the set of true Boolean expressions over Σ.

Example 2. **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:

$$\xrightarrow{(n,n)} q_0 \qquad \xrightarrow{(t,n)} q_1 \qquad (q_0,q_0) \xrightarrow{(T,N)} q_1 \qquad (q_1,q_0) \xrightarrow{(N,T)} q_2$$
$$(q_0,q_1) \xrightarrow{(N,T)} q_2 \qquad (q_0,q_0) \xrightarrow{(N,N)} q_0 \qquad (q_0,q_2) \xrightarrow{(N,N)} q_2 \qquad (q_2,q_0) \xrightarrow{(N,N)} q_2$$

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 $(w_1(i) \ldots, w_p(i)) \xrightarrow{(f_i, f_{i+1})} w(i)$ 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. 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 2. *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 $(\phi_1, \ldots, \phi_p) \xrightarrow{(f,f')}$ to be the set $\{w |\ \exists w_1 \in \phi_1 . \ldots . \exists w_p \in \phi_p .\ (w_1, \ldots, w_p) \xrightarrow{(f,f')} w \in \delta_H\}$.

Lemma 3. *For regular expressions $\phi_1, \ldots, \phi_p$ and symbols f, f', the set $(\phi_1, \ldots, \phi_p) \xrightarrow{(f,f')}$ 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 $(\phi_1, \ldots, \phi_p) \xrightarrow{(f,f')} \phi$. The transducer S is derived from D according to Algorithm 1 (see Figure 1). Observe that, by Lemma 3, the regular expression ϕ at line 4 of the code is always computable. Notice that in the first iteration, we have to choose $p = 0$ at line 3.

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. $\phi := (\phi_1, \ldots, \phi_p) \xrightarrow{(f,f')}$
5. $Q_S := Q_S \cup \{\phi\}$
6. $\delta_S := \delta_S \cup \{(\phi_1, \ldots, \phi_p) \xrightarrow{f,f'} \phi\}$
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

Fig. 1. Algorithm 1 : Computing symbolic transducer

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

Lemma 4. *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 $w \in \phi$.*

Corollary 1. *For a transducer D and its corresponding history and symbolic transducers H and S, we have $R(S) = R(H)$.*

From Lemma 2 and Corollary 1 we get

Theorem 1. *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 1 may in general not terminate.

Example 3. Consider the token tree transducer of Example 2.

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 1 on the protocol, we get e.g. $\phi_0 = {q_0}^* = \xrightarrow{(n,n)}$ and $\phi_1 = q_1{q_0}^* = \xrightarrow{(t,n)}$. 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 $\phi_2 = q_2q_1q_0^* = (\phi_0, \phi_1) \xrightarrow{(N,N)}$. 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.

6 Saturation

In order to make the algorithm in Figure 1 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 $\Box \notin \Sigma$ with arity 0. A *context* over Σ is a tree (S_C, λ_C) over $\Sigma \cup \{\Box\}$ such that there is exactly one $n_c \in S_C$ with $\lambda_C(n_c) = \Box$. In the sequel, we will always assume $n_c \in S_C$ to be the unique node with $\lambda_C(n_c) = \Box$.

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 $\lambda_C(n_c) = \Box$ 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 $\Box$ 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. Note that idempotent states can be syntactically characterized (see full version of the paper).

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 1 (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 2. $R(S) = R(S_{SAT})$.

We devote the rest of this subsection to the proof of Theorem 2 (achieved through Lemmas 5 to 7).

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

Lemma 5. *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 5 we get the next lemma stating that equivalent states have identical suffixes.

Lemma 6. *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 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 7. *If $(\phi_1, \ldots, \phi_i, \ldots, \phi_p) \xrightarrow{(f,f')} \phi \in \delta_S$, 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 $(\phi_1, \ldots, \phi_i', \ldots, \phi_p) \xrightarrow{(f,f')} \phi' \in \delta_S$.*

The fact that our equivalence relation is a congruence (Lemma 7) implies that we can apply the extension of the MyHill-Nerode theorem to trees (described in [CDG+99]).

Example 4. When we run Algorithm 1 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 1, 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. In the full version of this paper, we indicate how to lift these restrictions to a larger class of systems.

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 1 terminates.

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

Lemma 8. *For any regular expression ϕ generated by Algorithm 1, 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 2) that we can assume that all sets generated in Algorithm 1 are saturated with q_{idm}. This (together with Lemma 8) leads to the the following Lemma.

Lemma 9. *Let ϕ be a regular expression generated by Algorithm 1. 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 9 and finite local depth imply that we only need to consider regular expressions of a restricted form:

Lemma 10. *For a well-behaved transducer D with local depth k, Algorithm 1 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 9, and $q_i \notin \{q_{cpy}, q_{idm}\}$.

Consequently, we can conclude that for a well-behaved system, Algorithm 1 terminates.

Theorem 3. *Algorithm 1 terminates for any well-behaved transducer.*

Example 5. In Example 2, we have $q_{cpy} = q_0$ and $q_{idm} = q_2$. The local depth of $R(D)$ with respect to $\{q_1\}$ is 1.

8 Experimental Results

We have implemented a prototype based on Algorithm 1 and its modifications described in Section 6 and Section 7. In this section, we give a description of the protocols we have verified with our algorithm.

We describe and report more thoroughly these examples in the full version of this paper.

1. Simple Token Protocol This porotocol is detailed in Example 2.

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.

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.

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

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 which can move inside the tree.

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. 556, 567

[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. 556

[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. 555, 556

[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. 555

[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. 555, 556

[BMT01] A. Bouajjani, A. Muscholl, and T. Touili. Permutation rewriting and algorithmic verification. In *Proc. LICS' 01* 17^{th} *IEEE Int. Symp. on Logic in Computer Science*. IEEE, 2001. 556

[CDG+99] H. Common, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. October 1999. 557, 559, 564

[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. 556

[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. 555, 556

[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. 555, 556

[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. 555, 556, 566

[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. 555, 556

[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. 555

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. 555

[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. 555, 556

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proc.* 12^{th} *Int. Conf. on Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343, 2000. 556

[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. 556

[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. 555, 556