INFINITY 2005 Preliminary Version

Abstract Regular Tree Model Checking

Ahmed Bouajjani, Peter Habermehl [1,2]

LIAFA, University Paris 7, Case 7014, 2, place Jussieu, F-75251 Paris Cedex 05, France

Adam Rogalewicz, Tomáš Vojnar [3,4]

FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic

Abstract

Regular (tree) model checking (RMC) is a promising generic method for formal verification of infinite-state systems. It encodes configurations of systems as words or trees over a suitable alphabet, possibly infinite sets of configurations as finite word or tree automata, and operations of the systems being examined as finite word or tree transducers. The reachability set is then computed by a repeated application of the transducers on the automata representing the currently known set of reachable configurations. In order to facilitate termination of RMC, various acceleration schemas have been proposed. One of them is a combination of RMC with the abstract-check-refine paradigm yielding the so-called abstract regular model checking (ARMC). ARMC has originally been proposed for word automata and transducers only and thus for dealing with systems with linear (or easily linearisable) structure. In this paper, we propose a generalisation of ARMC to the case of dealing with trees which arise naturally in a lot of modelling and verification contexts. In particular, we first propose abstractions of tree automata based on collapsing their states having an equal language of trees up to some bounded height. Then, we propose an abstraction based on collapsing states having a non-empty intersection (and thus "satisfying") the same bottom-up tree "predicate" languages. Finally, we show on several examples that the methods we propose give us very encouraging verification results.

1 Introduction

Regular model checking [14,4,5] is a general method for formal verification of infinite-state systems. Configurations of systems are encoded as finite words over a finite alphabet Σ and transitions are encoded as relations over words. Then, word automata over Σ can naturally be used to represent and manipulate (infinite) sets of configurations and transducers over $(\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})$ are used to represent the transition relation. To verify safety properties, a reachability analysis is performed by calculating transitive closures of transducers or images of automata by iteration of transducers. Termination is usually not guaranteed and therefore various acceleration methods have been proposed.

[1] Supported by the French ministry of research (ACI project Securité Informatique).

[2] Email: abou@liafa.jussieu.fr, Peter.Habermehl@liafa.jussieu.fr

[3] Supported by the Czech Grant Agency projects 102/05/H050, 102/03/D211, and 102/04/0780.

[4] Email: rogalew@fit.vutbr.cz, vojnar@fit.vutbr.cz

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

As one of the most successful acceleration methods and also as a way to cope with the problem of state space explosion in automata representing configurations, *abstract regular model checking* (ARMC) [8] has been introduced recently. This generic method uses the well known *abstract-check-refine* paradigm within regular model checking. Abstractions are defined on word automata representing configurations. Then, an abstract reachability analysis which is guaranteed to terminate is performed. Suitable refinements of abstractions are defined for the case a spurious counter-example is encountered. In this way, an abstraction detailed just enough to answer a particular verification question is computed. ARMC has been successfully applied to a lot of different systems, like counter automata, parameterised networks of processes, and programs with lists [7].

To handle other structures than linear (or easily linearisable) ones, regular *tree* model checking [14,6,1,19,2] has been proposed. Instead of words, configurations are finite trees and instead of word automata, tree automata are used to represent sets of configurations. Then, tree transducers model transitions. Like in the word case, several acceleration approaches for reachability analysis exist.

Tree like structures are very common and appear naturally in many modelling and verification contexts. For example, in the case of parameterized tree networks, labelled trees of arbitrary height represent a configuration of the network: each process is a node of the tree and the label its control state. Trees also arise naturally, e.g., as a representation of configurations of multithreaded recursive programs [12,17], as a representation structure of heaps [15], or when representing structured data such as XML documents [9].

In this paper, we extend the framework of ARMC from words to trees. We use bottom-up tree automata and transducers. Like in ARMC, we use abstract fixpoint computations in some *finite* domain of automata. The abstract fixpoint computations always terminate and provide overapproximations of the reachability sets. To achieve this, we define techniques that systematically map any tree automaton M to a tree automaton M' from some finite domain such that M' recognises a superset of the language of M. For the case that the computed overapproximation is too coarse and a spurious counter-example is detected, we give effective principles allowing the abstraction to be refined such that the new abstract computation does not encounter the same counter-example.

We, in particular, propose two abstractions for tree automata. Similarly to ARMC, both of them are based on collapsing automata states according to a suitable equivalence relation. The first is based on considering two tree automata states equivalent if their *languages of trees up to a certain fixed height* are equal. The second abstraction is defined by a set of regular *predicate languages* L_P. We consider a state q of a tree automaton M to "satisfy" a predicate language L_P if the intersection of L_P with the tree language $L(M, q)$ accepted from the state q is not empty. Then, two states are equivalent if they satisfy the same predicates.

We have implemented the above abstractions in a prototype tool using the Timbuk [13] tree automata library. We have experimented with the tool on various parameterized tree network protocols. The results are very encouraging and compare very well with other tools, which gives us a very good basis and motivation for a further development of the method.

2 Regular Tree Languages and Transducers

This section is a brief introduction to regular tree languages and transducers. A more detailed description can be found, e.g., in [10,11].

An *alphabet* Σ is a finite set of symbols. Σ is called *ranked* if there exists a *rank* function $\rho : \Sigma \to \mathbb{N}$. For each $k \in \mathbb{N}$, $\Sigma_k \subseteq \Sigma$ is the set of all symbols with rank k. Symbols of Σ_0 are called *constants*. Let χ be a denumerable set of symbols called *variables*. $T_\Sigma[\chi]$ denotes the set of *terms* over Σ and χ. The set $T_\Sigma[\emptyset]$ is denoted by T_Σ, and its elements are called *ground terms*. A term t from $T_\Sigma[\chi]$ is called *linear* if each variable occurs at most once in t. Terms in $T_\Sigma[\chi]$ can be viewed as trees—leaves are labelled by constants and variables, and each node with k sons is labelled by a symbol from Σ_k.

A *bottom-up tree automaton* over a ranked alphabet Σ is a tuple $A = (Q, \Sigma, F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, and δ is a set of transitions of the following types: (i) $f(q_1, \ldots, q_n) \to_\delta q$, (ii) $a \to_\delta q$, and (iii) $q \to_\delta q'$ where $a \in \Sigma_0$, $f \in \Sigma_n$, and $q, q', q_1, \ldots, q_n \in Q$.

Note: Below, we call a bottom-up tree automaton simply a tree automaton.

Let t be a ground term. A run of a tree automaton A on t is defined as follows. First, leaves are labelled with states. If a leave is a symbol $a \in \Sigma_0$ and there is a rule $a \to_\delta q \in \delta$, the leave is labelled by q. An internal node $f \in \Sigma_k$ is labelled by q if there exists a rule $f(q_1, q_2, \ldots, q_k) \to_\delta q \in \delta$ and the first son of the node has the state label q_1, the second one q_2, ..., and the last one q_k. Rules of the type $q \to_\delta q'$ are called *ε-steps* and allow us to change a state label from q to q'. If the top symbol is labelled with a state from the set of final states F, the term t is accepted by the automaton A.

A set of ground terms accepted by a tree automaton A is called a *regular tree language* and is denoted by $L(A)$. Let $A = (Q, \Sigma, F, \delta)$ be a tree automaton and $q \in Q$ a state, then we define the *language of the state q*—$L(A, q)$—as the set of ground terms accepted by the tree automaton $A_q = (Q, \Sigma, \{q\}, \delta)$. The language $L^{\leq n}(A, q)$ is defined to be the set $\{t \in L(A, q) \mid height(t) \leq n\}$.

A *bottom-up tree transducer* is a tuple $\tau = (Q, \Sigma, \Sigma', F, \delta)$ where Q is a finite set of states, $F \subseteq Q$ is a set of final states, Σ is an input ranked alphabet, Σ' is an output ranked alphabet, and δ is a set of transition rules of the following types: (i) $f(q_1(x_1), \ldots, q_n(x_n)) \to_\delta q(u)$, $u \in T_{\Sigma'}[\{x_1, \ldots, x_n\}]$, (ii) $q(x) \to_\delta q'(u)$, $u \in T_{\Sigma'}[\{x\}]$, and (iii) $a \to_\delta q(u)$, $u \in T_{\Sigma'}$ where $a \in \Sigma_0$, $f \in \Sigma_n$, $x, x_1, \ldots, x_n \in \chi$, and $q, q', q_1, \ldots, q_n \in Q$.

Note: In the following, we call a bottom-up tree transducer simply a tree transducer. We always use tree transducers with $\Sigma = \Sigma'$.

A run of a tree transducer τ on a ground term t is similar to a run of a tree automaton on this term. First, rules of type *(iii)* are used. If a leaf is labelled by a symbol a and there is a rule $a \to_\delta q(u) \in \delta$, the leaf is replaced by the term

u and labelled by the state q. If a node is labelled by a symbol f, there is a rule $f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \rightarrow_\delta q(u) \in \delta$, the first subtree of the node has the state label q_1, the second one q_2, ..., and the last one q_n, then the symbol f and all subtrees of the given node are replaced according to the right-hand side of the rule with the variables $x_1, \ldots, x_n$ substituted by the corresponding left-hand-side subtrees. The state label q is assigned to the new tree. Rules of type *(ii)* are called ε-*steps*. They allow us to replace a q-state-labelled tree by the right hand side of the rule and assign the state label q' to this new tree with the variable x in the rule substituted by the original tree. A run of a transducer is successful if the root of a tree is processed and is labelled by a state from F.

A tree transducer is *linear* if all right-hand sides of its rules are linear (no variable occurs more than once). The class of linear bottom-up tree transducers is closed under composition. A tree transducer is called *structure-preserving* (or a *relabelling*) if it does not modify the structure of input trees and just changes the labels of their nodes. By abuse of notation, we identify a transducer τ with the relation $\{(t, t') \in T_\Sigma \times T_\Sigma \mid t \rightarrow_\delta^* q(t') \text{ for some } q \in F\}$. For a set $L \subseteq T_\Sigma$ and a relation $R \subseteq T_\Sigma \times T_\Sigma$, we denote $R(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w', w) \in R\}$ and $R^{-1}(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w, w') \in R\}$. If τ is a linear tree transducer and L is a regular tree language, then the sets $\tau(L)$ and $\tau^{-1}(L)$ are regular and effectively constructible [11,10].

Let $id \subseteq T_\Sigma \times T_\Sigma$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations $\tau^0 = id$, $\tau^{i+1} = \tau \circ \tau^i$ and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. Below, we suppose $id \subseteq \tau$ meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

3 Abstract Regular Tree Model Checking

In this section, we first recall the notion of regular tree model checking. Then, we introduce abstract regular tree model checking by defining several abstractions on tree automata.

3.1 Regular Tree Model Checking

Regular tree model checking [1,6,14] is a generalisation of regular model checking [5] to trees. A configuration of a system is encoded as a term (tree) over a ranked alphabet and a set of such terms as a regular tree automaton. The transition relation of a system is encoded as a linear tree transducer τ. We are given a tree automaton $Init$ encoding the set of initial states. For safety properties, a set of bad states (represented by a tree automaton Bad) is given. Then, the basic verification problem consists in deciding whether

$$\tau^*(L(Init)) \cap L(Bad) = \emptyset \qquad (1)$$

This problem is in general undecidable (an iterative computation of $\tau^*(L(Init))$ does not terminate). Several methods [1,2,6] have been proposed to calculate in some cases τ^* or $\tau^*(L(Init))$. These techniques all compute exact sets or relations. We tackle the model-checking problem by generalising the abstract regular model checking method [8] to tree automata. This method computes an overapproximation of $\tau^*(L(Init))$ with a precision just sufficient to safely solve the verification problem (1).

3.2 Abstract Regular Tree Model Checking

Abstract regular tree model checking (ARTMC) combines regular tree model checking with automatic abstraction. The main idea of ARTMC is a generalisation of abstract regular model checking [8] to regular tree languages. For this, the abstraction techniques designed for word automata have to be adapted to tree automata.

We start by recalling the basic framework of abstract regular model checking (here phrased directly for trees).

Let Σ be a ranked alphabet and $\mathbb{M}_\Sigma$ the set of all tree automata over Σ. We define an abstraction function as a mapping $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ where $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$ and $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. An abstraction α' is called a *refinement* of the abstraction α if $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. Given a tree transducer τ and abstraction α, we define a mapping $\tau_\alpha : \mathbb{M}_\Sigma \to \mathbb{M}_\Sigma$ as $\forall M \in \mathbb{M}_\Sigma : \tau_\alpha(M) = \hat{\tau}(\alpha(M))$ where $\hat{\tau}(M)$ is a minimal automaton describing the language $\tau(L(M))$. An abstraction α is *finite range* if the set $\mathbb{A}_\Sigma$ is finite.

Let $Init$ be a tree automaton representing the set of initial configurations and Bad be a tree automaton representing the set of bad configurations. Now, we may iteratively compute the sequence $(\tau_\alpha^i(Init))_{i\geq 0}$. Since we suppose $id \subseteq \tau$, it is clear that if α is finitary, there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(Init) = \tau_\alpha^k(Init)$. The definition of α implies $L(\tau_\alpha^k(Init)) \supseteq \tau^*(L(Init))$. This means that in a finite number of steps, we can compute an overapproximation of the reachability set $\tau^*(L(Init))$.

If $L(\tau_\alpha^k(Init)) \cap L(Bad) = \emptyset$, then the verification problem (1) has a positive answer. Otherwise, the answer to the problem (1) is not necessarily negative since during the computation of $\tau_\alpha^*(L(Init))$, the abstraction α may introduce extra behaviours leading to $L(Bad)$. Let us examine this case. Assume that $\tau_\alpha^*(Init) \cap L(Bad) \neq \emptyset$, which means that there is a symbolic path:

$$Init,\ \tau_\alpha(Init),\ \tau_\alpha^2(Init), \cdots \tau_\alpha^{n-1}(Init),\ \tau_\alpha^n(Init) \tag{2}$$

such that $L(\tau_\alpha^n(Init)) \cap L(Bad) \neq \emptyset$. We analyse this path by computing the sets $X_n = L(\tau_\alpha^n(Init)) \cap L(Bad)$, and for every $k \geq 0$, $X_k = L(\tau_\alpha^k(Init)) \cap \tau^{-1}(X_{k+1})$. Two cases may occur: (i) either $X_0 = L(Init) \cap (\tau^{-1})^n(X_n) \neq \emptyset$, which means that the problem (1) has a *negative answer*, or (ii) there is a $k \geq 0$ such that $X_k = \emptyset$, and this means that the symbolic path (2) is actually a *spurious counter-example* due to the fact that α is too coarse. In this last situation, we need to refine α and iterate the procedure. Therefore, our approach is based on the definition of abstraction schemas allowing to compute families of (automatically) refinable abstractions.

3.3 Abstraction Based on Automata State Equivalence

Below, we discuss two possible tree automata abstraction schemas which are based on tree automata state equivalence. First, tree automata states are split into several equivalence classes by an equivalence relation. Then, the abstraction function collapses states from each equivalence class into one state. Formally, a tree automata state equivalence schema $\mathbb{E}$ is defined as follows: To each tree automaton $M = (Q, \Sigma, F, \delta) \in \mathbb{M}_\Sigma$, an equivalence relation $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ is assigned. Then the automata abstraction function $\alpha_{\mathbb{E}}$ corresponding to the abstraction schema $\mathbb{E}$ is defined as $\forall M \in \mathbb{M}_\Sigma : \alpha_{\mathbb{E}}(M) = M/ \sim_M^{\mathbb{E}}$. We call $\mathbb{E}$ finitary if $\alpha_{\mathbb{E}}$ is finitary (i.e. there is a finite number of equivalence classes). We refine $\mathbb{E}$ by making $\sim_M^{\mathbb{E}}$ finer.

3.4 Abstraction Based on Languages of Finite Height

We now present the possibility of defining automata state equivalence schemas based on comparing automata states wrt. a certain bounded part of their languages. The abstraction schema $\mathbb{H}_n$ is a generalisation of a similar schema proposed for word automata in [8]. This schema defines two states of a tree automaton M as equivalent if their languages up to the given height n are identical.

Formally, for a tree automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{H}_n$ defines the state equivalence as the equivalence $\sim_M^n$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$.

There is a finite number of languages of trees with a maximal height n, and so this abstraction is finite range. Refining of the abstraction can be done by increasing the value of n.

The abstraction schema $\mathbb{H}_n$ can be implemented in a similar way as minimisation of tree automata. Just the main loop of the minimisation procedure is stopped after n iterations.

3.5 Abstraction Based on Predicate Languages

We next introduce a predicate-based abstraction schema $\mathbb{P}_{\mathcal{P}}$, which was inspired by the predicate based abstraction on words [8].

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$ be a set of *predicates*. Each predicate $P \in \mathcal{P}$ is a tree language represented by a tree automaton. Let $M = (Q, \Sigma, F, \delta)$ be a tree automaton, then two states $q_1, q_2 \in Q$ are equivalent if their languages $L(M, q_1)$ and $L(M, q_2)$ have a nonempty intersection with exactly the same subset of predicates from the set $\mathcal{P}$.

Formally, for an automaton $M = (Q, \Sigma, F, \delta)$, $\mathbb{P}_{\mathcal{P}}$ defines the state equivalence as the equivalence $\sim_M^{\mathcal{P}}$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^{\mathcal{P}} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap L(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap L(M, q_2) \neq \emptyset)$.

Clearly, since $\mathcal{P}$ is finite and there is only a finite number of subsets of $\mathcal{P}$ representing the predicates with which a given state has a nonempty intersection, $\mathbb{P}_{\mathcal{P}}$ is *finitary*. This schema can be refined by adding new predicates into the set $\mathcal{P}$. The following theorem shows that we may eliminate a spurious counter-example by extending the predicate set $\mathcal{P}$ by the languages of all states of the tree automaton representing X_{k+1} in the analysis of the spurious counter-example (recall that $X_k = \emptyset$) as presented in Section 3.2.

Theorem 3.1 *Let us have any two tree automata $M = (Q_M, \Sigma, F_M, \delta_M)$ and $X = (Q_X, \Sigma, F_X, \delta_X)$ and a finite set of predicate automata $\mathcal{P}$ s.t. $\forall q_X \in Q_X : \exists P \in \mathcal{P} : L(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X) = \emptyset$ too.*

Proof. The proof is a generalisation of the proof [8] for word automata. We prove the theorem by contradiction. Suppose $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X) \neq \emptyset$. Let $t \in L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$. As t is accepted by $\alpha_{\mathbb{P}_\mathcal{P}}(M)$, M must accept it when we allow it to perform a certain number of "jumps" between states equal wrt. $\sim_M^\mathcal{P}$—after accepting a subtree of t and getting to some $q \in Q_M$, M is allowed to jump to any $q' \in Q_M$ such that $q \sim_M^\mathcal{P} q'$ and go on accepting from there (with or without further jumps).

Let $i > 0$ be the minimum number of jumps needed for accepting a tree from $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$ in M and let t' be such a tree. When looking at the acceptance of t' in M (with some jumps allowed), we can identify maximum subtrees of t' that may be accepted without jumps—in the worst case, they are just the leaves. Let us take any of such subtrees. Such a subtree t_1 is accepted in some q_1, from which M jumps to some q_2 and goes on accepting the rest of the input. Suppose that t_1 is accepted in some $q_X \in Q_X$ in X. As $t_1 \in L(M, q_1)$, $L(M, q_1) \cap L(P) \neq \emptyset$ for the predicate $P \in \mathcal{P}$ for which $L(P) = L(X, q_X)$. Moreover, as $q_1 \sim_M^\mathcal{P} q_2$, $L(M, q_2) \cap L(P) \neq \emptyset$ too. This implies there exists $t_2 \in L(P)$ such that $t_2 \in L(M, q_2)$ and $t_2 \in L(X, q_X)$. However, this means that the tree t'' that we obtain from t' by replacing its subtree t_1 with t_2 and that clearly belongs to $L(\alpha_{\mathbb{P}_\mathcal{P}}(M)) \cap L(X)$ can be accepted in M with $i - 1$ jumps, which is a contradiction to the assumption of i being the minimum number of jumps needed. □

The abstraction of an automaton M wrt. the state equivalence based on predicate languages $\mathbb{P}_\mathcal{P}$ can be implemented as labelling each state of M by the predicates with which its language has a non-empty intersection, and then collapsing states with an equal labelling. Here, let us stress that when refining $\mathbb{P}_\mathcal{P}$, it is not necessary to store each of the newly introduced predicates corresponding to the states of X_{k+1} independently and then perform the labelling independently for each of them. We may keep just X_{k+1} and then perform labelling not by just X_{k+1} but by each of its states. Moreover, this labelling may be implemented by one simultaneous run through M and X_{k+1}, which corresponds to an efficient simultaneous labelling by all the predicates contained in X_{k+1}.

4 Experiments with ARTMC

In order to be able to practically evaluate the proposed methods of ARTMC, we have implemented them in a prototype tool. We have based our prototype tool on the *Timbuk* library [13] written in Ocaml. Timbuk provided us with the basic operations over tree automata needed in ARTMC (such as union, intersection, complementation, etc.). However, we had to extend Timbuk with a support for tree transducers. We added two implementations of tree transducers—a simpler and more efficient for structure-preserving transducers and a more complex for general transducers. The latter implementation exploits a decomposition of a tree transducer into three less complicated ones as described in [11]. This decomposition can be performed automatically for any tree transducer.

We have tested our verification methods on several examples of protocols using a parameterised tree-shaped network cited in the literature [14,3,1,2] where the necessity to cover all possible values of the parameters leads to dealing with infinite state spaces:

- *Simple Token Protocol*. A token is being passed in a tree-shaped network from a leave to the root. We check that the token does not disappear nor replicate.
- *Two-Way Token Protocol*. An analogy to the previous example, but we allow the token to be passed upwards as well as downwards.
- *Percolate Protocol*. A tree-shaped network of processors computes the logical disjunction of the boolean values that appear in the leave nodes. We check that the computed value is always correct.
- *Tree Arbiter Protocol*. A tree-shaped network is used to implement mutual exclusion among the leave processors. A request to enter the critical section is propagated upwards till a node is found which has a token allowing one to enter the critical section or which knows where the token is (because it granted the token to one of its children). A node with the token can always send the token upwards or grant it to any of its children. We check the mutual exclusion property.
- *Leader Election Protocol*. One of a set of processors is to be elected a leader and a tree-shaped network is used for this purpose. The leaves are divided into candidates and non-candidates. The information about the existence of candidates is propagated upwards. In the subsequent downward phase, a path leading from the root to one of the candidate nodes is non-deterministically selected and thus a leader is established. We check that exactly one leader is chosen.

All the above examples work with a tree-shaped network of a fixed structure. In order to test the ability of our method to work with non-structure-preserving systems, we have considered a *simple broadcast protocol*. In the protocol, the root sends a message to all leave nodes. They answer and the answers are combined when travelling upwards. An intermediate node may decide to resend the message downwards and wait for new data. New nodes may dynamically join the network at leaves and also leave the network in a suitable moment. We check that there is at most one active message on each path from the root to the leaves.

The results of our experiments are summarised in Table 1. We performed experiments with both the finite-height abstraction as well as with the predicate-based abstraction. We considered both forward as well as backward verification—i.e. starting with the set of initial states and checking that the bad states cannot be reached or vice versa. In the table, we always present the better result of these two approaches. For the finite-height abstraction, we considered the initial height one (and increased it by one if necessary—in the cases presented in Table 1, this was not necessary). For the predicate-based abstraction, we considered the automaton describing the set of bad states as the only initial predicate (or—more precisely—all the automata that can be obtained from it by considering each of its states as the only accepting one; in the cases presented in Table 1, no refinement was necessary when using these initial predicates). We experimented with the empty initial set of predicates too—this turned out to be the fastest option for the Percolate protocol (one refinement was necessary in this case).

Table 1
Some results of experimenting with ARTMC

Protocol	$\mathbb{H}_n$	$\mathbb{P}_{\mathcal{P}}$
Token passing	backwards: 0.08s	forwards: 0.06s
Two-way token passing	backwards: 1.0s	forwards: 0.09s
Percolate	backwards: 20.8s	forwards: 2.4s
Tree arbiter	backwards: 0.31s	backwards: 0.34s
Leader election	backwards: 2.0s	forwards: 1.74s
Broadcasting	backwards: 9.1s	forwards: 1.0s

Notice that the predicate-based abstraction is almost always better than the finite-height abstraction. This is different from the word case where the results differ. An explanation of this phenomenon is a part of our future work. The verification times presented in Table 1 were obtained on an Intel Centrino 1.6GHz machine with 768MB of memory. We consider these results very encouraging and we are now working on a new version of our tool that will be based on the Mona library [16]. This gives us hope of even better results and an expectation of a successful applicability of the tool on real-life case studies (including, e.g., verification of programs with dynamic linked data structures).

5 Conclusions

We have proposed abstract regular tree model checking as a generalisation of the successful approach of abstract regular model checking. In particular, we have proposed two kinds of abstractions over tree automata based on collapsing in some sense equivalent states of these automata. One of the abstractions decides which states are equivalent by comparing their languages of trees of a bounded height while the second one compares the states wrt. whether their languages satisfy (i.e. are not disjoint with) a set of predicates having the form of regular tree languages. Both of these abstractions are automatically refinable when a spurious counterexample is found and allow one to deal with an overapproximation of the state space precise just enough to verify a given property of interest. In this way, the state explosion in automata representing the reachability set is fought. The above abstractions were inspired by some of the schemas used in the original ARMC.

We have implemented the proposed methods in a prototype tool and evaluated them on multiple verification examples with very encouraging results. Currently, we are building a new and much more elaborate version of our tool based on the tree libraries of Mona [16]. This tool promises even better results and a high potential for a successful application on real-life verification problems.

Apart from finishing the new version of our tool, our future work includes a research on the various application domains of ARTMC. They include, e.g. verification of programs with dynamic linked data structures. ARMC has already been shown useful for verification of programs with 1-selector linked dynamic data structures [7]. The use of ARTMC could allow us to handle much more general structures. To encode data structures with a graph shape, we plan to use trees with some special symbols placed in their nodes to describe additional edges over the tree. Another promising application area is the domain of XML manipulations. Indeed, XML documents have a tree structure and most of XML parsers are based on the tree automata theory—in particular, on hedge automata [9]. Furthermore, we intend to use our approach for programs with abstract data structures and cryptographic protocols along the lines of [18]. For all these applications we plan to study the encoding in tree automata and transducers and the possibility of defining application dependent abstractions.

References

[1] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, *LNCS* 2404. Springer, 2002.

[2] P.A. Abdulla, A. Legay, J. d'Orso, and A. Rezine. *Simulation-Based Iteration of Tree Transducers.* In *Proc. of TACAS'05*, *LNCS* 3440. Springer, 2005.

[3] R. Alur, R. Brayton, T. Henzinger, S. Qadeer, S. Rajamani. Partial-Order Reduction in Symbolic State Space Exploration. In *Proc. of CAV'97*, *LNCS* 1254. Springer, 1997.

[4] B. Boigelot and P. Wolper. Verifying systems with infinite but regular state spaces. In *Proc. of CAV'98*, *LNCS* 1427. Springer, 1998.

[5] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, *LNCS* 1855. Springer, 2000.

[6] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, *LNCS* 2404. Springer, 2002.

[7] A. Bouajjani, P. Habermehl, P. Moro, T. Vojnar. *Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking*. In *TACAS'05*, *LNCS* 3440. Springer, 2005.

[8] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking, In *Proc. CAV'04*, *LNCS* 3114. Springer, 2004.

[9] A. Bruggemann-Klein, M. Murata, and D. Wood. *Regular tree and regular hedge languages over unranked alphabets: Version 1*. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[10] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005. `http://www.grappa.univ-lille3.fr/tata`

[11] J. Engelfriet. Bottom-up and Top-down Tree Transformations—A Comparison, *Mathematical System Theory*, 9:198–231, 1975.

[12] J. Esparza. Grammars as Processes. In *Formal and Natural Computing*, *LNCS* 2300, 2002.

[13] T. Genet. Timbuk, a tree automata library, 2005. `http://www.irisa.fr/lande/genet/timbuk`

[14] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proc. CAV'97*, *LNCS* 1254. Springer, 1997.

[15] N. Klarlund and M.I. Schwartzbach. Graph Types. In *Proc. of POPL'93*, ACM, 1993.

[16] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, University of Aarhus, Denmark, 2001.

[17] M. Křetínský, V. Řehák, and J. Strejček. Extended Process Rewrite Systems: Expressiveness and Reachability. In *Proc. of Concur'04*, *LNCS* 3170. Springer, 2004.

[18] D. Monniaux. Abstracting cryptographic protocols with tree automata. In *Science of Computer Programming*, Volume 47, Issue 2-3 (May 2003).

[19] A. Pnueli and E. Shahar. *Acceleration in Verification of Parameterized Tree Networks*. Technical report MCS02-12, Weizmann Institute of Science, Israel, 2004.