Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Abstract Regular (Tree) Model Checking

Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar

LIAFA, University Paris Diderot—Paris 7/CNRS and FIT, Brno University of Technology

Received: date / Revised version: date

Abstract. Regular model checking is a generic technique for verification of infinite-state and/or parametrised systems which uses finite word automata or finite tree automata to finitely represent potentially infinite sets of reachable configurations of the systems being verified. The problems addressed by regular model checking are typically undecidable. In order to facilitate termination in as many cases as possible, acceleration is needed in the incremental computation of the set of reachable configurations in regular model checking. In this work, we describe how various incrementally refinable abstractions on finite (word and tree) automata can be used for this purpose. Moreover, the use of abstraction does not only increase chances of the technique to terminate, but it also significantly reduces the problem of an explosion in the number of states of the automata that are generated by regular model checking. We illustrate efficiency of abstract regular (tree) model checking in verification of simple systems with various sources of infinity such as unbounded counters, queues, stacks, and parameters. We then show how abstract regular tree model checking can be used for verification of programs manipulating tree-like dynamic data structures. Even more complex data structures can be handled using a suitable tree-like encoding.

1 Introduction

Model checking is nowadays widely accepted as a powerful technique for verification of finite-state systems. However, many real-life systems exhibit various aspects of infinity. In the case of discrete systems that we concentrate on in this paper, infinity can arise due to dealing with various kinds of *unbounded data structures* such as push-down stacks needed for dealing with recursive procedures, queues of waiting processes or messages, unrestricted counters (or integer variables), or dynamic linked data structures (such as lists or trees). A need to deal with infinite state spaces may also arise due to various kinds of *parameters* (such as the maximum value of some variable, the maximum length of a queue, or the number of processes in a system) when one wants to verify a given parametric system for any value of its parameters. In the last case, to be more precise, we are dealing with *infinite families* of systems which themselves may be finite-state or infinite-state. Nevertheless, the need to verify the system for any member of the family leads anyhow to infinite-state verification as the union of the state spaces of all the family members is infinite.

To deal with infinity in model checking, one can, e.g., try to identify sufficient finite bounds on the sources of infinity—the so called *cut-offs*, one can use various finite-range *abstractions*, or techniques of *automated induction* (for an overview of such techniques, see, e.g., [58]). Yet another approach is to use *symbolic model checking* based on a finite representation of infinite sets of states by means of logics, automata, grammars, etc. Among successful symbolic verification methods, we have the so called *regular (tree) model checking* R(T)MC, first mentioned in [36], on which we concentrate in this paper.

In R(T)MC, configurations of systems are encoded as words or trees over a finite alphabet whereas transitions are modelled as finite state transducers or, more generally, as regularity preserving relations on words or trees[1]. Finite (tree) automata can then naturally be used to represent and manipulate potentially infinite sets of configurations, allowing reachability properties to be checked by computing transitive closures of transducers [35, 23, 5, 10, 3] or images of automata by iteration of transducers [16, 52]—depending on whether dealing with *reachability*

[1] Such relations can be expressed, e.g., as special operations on automata.

relations or *reachability sets* is preferred. To facilitate termination of the computation, which is in general not guaranteed as the problem being solved is undecidable, various acceleration methods are usually used.

In this paper, we, in particular, concentrate on using *abstraction* as a means of acceleration. The description builds on our proposal [15,13] of combining R(T)MC with the *CEGAR* loop [21]. Instead of precise acceleration techniques, we use abstract fixpoint computations in some *finite* domain of automata. The abstract fixpoint computations always terminate and provide overapproximations of the reachability sets (relations). To achieve this, we define techniques that systematically map any automaton M to an 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 counterexample is detected, we provide effective techniques allowing the abstraction to be refined such that the new abstract computation does not encounter the same counterexample.

Both for the word and tree cases, we discuss two general purpose classes of techniques for abstracting automata[2]. They take into account the structure of the automata and are based on collapsing their states according to some equivalence relation. The first one is inspired by *predicate abstraction* [30]. However, contrary to classical predicate abstraction, we associate predicates with states of automata representing sets of configurations rather than with the configurations themselves. An abstraction is defined by a set of regular *predicate languages* L_P. We consider a state q of an automaton M to "satisfy" a predicate language L_P if the intersection of L_P with the language $L(M,q)$ accepted from the state q is not empty. Then, two states are equivalent if they satisfy the same predicates. The second abstraction technique is then based on considering two automata states equivalent if their *languages of words up to a certain fixed length* (or *trees up to a certain fixed height*) are equal. For both of these two abstraction methods, we provide effective refinement techniques allowing us to discard spurious counterexamples.

All the above mentioned techniques have up to now been implemented in prototype tools and tested on various case studies. In particular, abstract regular *word* model checking was successfully applied for verification of parametric networks of processes, pushdown systems, counter automata, systems with queues, and programs with dynamic singly-linked structures [15,12]. Abstract regular *tree* model checking was applied for verification of parametric networks of processes [13] and programs with generic dynamic-linked data structures [14]. In this paper, we briefly report on all these applications and describe the last mentioned application in more detail.

[2] In [12], some specialised abstractions optimised for verification of list manipulating programs are proposed. These abstractions are, however, beyond the scope of this paper.

2 Preliminaries

2.1 Finite Word Automata and Transducers

A (non-deterministic) *finite-state automaton* is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ a finite alphabet, $\delta : Q \times \Sigma \to 2^Q$ a transition function, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of final states. The transition relation $\xrightarrow[M]{} \subseteq Q \times \Sigma^* \times Q$ of M is defined as the smallest relation satisfying: (1) $\forall q \in Q : q \xrightarrow[M]{\varepsilon} q$, (2) if $q' \in \delta(q,a)$, then $q \xrightarrow[M]{a} q'$, and (3) if $q \xrightarrow[M]{w} q'$ and $q' \xrightarrow[M]{a} q''$, then $q \xrightarrow[M]{wa} q''$ for $a \in \Sigma, w \in \Sigma^*$. We drop the subscript M if no confusion is possible. M is called *deterministic* iff $\forall q \in Q\ \forall a\ \in \Sigma : |\delta(q,a)| \leq 1$.

The *language* recognised by a finite-state automaton $M = (Q, \Sigma, \delta, q_0, F)$ from a state $q \in Q$ is defined by $L(M,q) = \{w \in \Sigma^* \mid \exists q_F \in F : q \xrightarrow[M]{w} q_F\}$. The language $L(M)$ of M is equal to $L(M, q_0)$. A set $L \subseteq \Sigma^*$ is a *regular set* iff there exists a finite-state automaton M such that $L = L(M)$. We also define the *backward language* $\overleftarrow{L}(M,q) = \{w \mid q_0 \xrightarrow[M]{w} q\}$ and the *forward/backward languages of words up to a certain length*: $L^{\leq n}(M,q) = \{w \in L(M,q) \mid |w| \leq n\}$ and similarly $\overleftarrow{L}^{\leq n}(M,q)$. We define the *forward/backward trace languages* of states $T(M,q) = \{w \in \Sigma^* \mid \exists w' \in \Sigma^* : ww' \in L(M,q)\}$ and similarly $\overleftarrow{T}(M,q)$. Finally, we define accordingly forward/backward trace languages $T^{\leq n}(M,q)$ and $\overleftarrow{T}^{\leq n}(M,q)$ of *traces up to a certain length.*

Given a finite-state automaton $M = (Q, \Sigma, \delta, q_0, F)$ and an equivalence relation $\sim$ on its set of states Q, $M_{/\sim}$ denotes the *quotient automaton* of M wrt. $\sim$, $M_{/\sim} = (Q_{/\sim}, \Sigma, \delta_{/\sim}, [q_0]_{/\sim}, F_{/\sim})$ where $Q_{/\sim}$ and $F_{/\sim}$ are the partitions of Q and F wrt. $\sim$, respectively, $[q_0]_{/\sim}$ is the equivalence class of Q wrt. $\sim$ containing q_0, and $\delta_{/\sim}$ is defined s.t. $[q_1]_{/\sim} \xrightarrow[M_{/\sim}]{a} [q_2]_{/\sim}$ for $[q_1]_{/\sim}, [q_2]_{/\sim} \in Q_{/\sim}$, $a \in \Sigma$ iff $q_1' \xrightarrow[M]{a} q_2'$ for some $q_1' \in [q_1]_{/\sim}, q_2' \in [q_2]_{/\sim}$.

A *finite-state transducer* over Σ is a 5-tuple $\tau = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ a finite input/output alphabet, $\delta : Q \times \Sigma_\varepsilon \times \Sigma_\varepsilon \to 2^Q$ a transition function, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of final states. A finite-state transducer is called a *length-preserving transducer* if its transitions do not contain ε. The transition relation $\xrightarrow[\tau]{} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ is defined as the smallest relation satisfying: (1) $q \xrightarrow[\tau]{\varepsilon/\varepsilon} q$ for every $q \in Q$, (2) if $q' \in \delta(q,a,b)$, then $q \xrightarrow[\tau]{a/b} q'$, and (3) if $q \xrightarrow[\tau]{w/u} q'$ and $q' \xrightarrow[\tau]{a/b} q''$, then $q \xrightarrow[\tau]{wa/ub} q''$ for $a, b \in \Sigma_\varepsilon, w, u \in \Sigma^*$. The subscript τ will again be dropped if no confusion is possible. A finite-state transducer $\tau = (Q, \Sigma, \delta, q_0, F)$ defines the *relation* $\varrho_\tau = \{(w,u) \in \Sigma^* \times \Sigma^* \mid \exists q_F \in F : q_0 \xrightarrow[\tau]{w/u} q_F\}$.

A relation $\varrho \subseteq \Sigma^* \times \Sigma^*$ is a *regular relation* iff there exists a finite-state transducer τ such that $\varrho = \varrho_\tau$. For a set $L \subseteq \Sigma^*$ and a relation $\varrho \subseteq \Sigma^* \times \Sigma^*$, we denote by $\varrho(L)$ the set $\{w \in \Sigma^* \mid \exists w' \in L : (w', w) \in \varrho\}$. A relation $\varrho \subseteq \Sigma^* \times \Sigma^*$ is called *regularity preserving* iff $\varrho(L)$ is regular for any regular set $L \subseteq \Sigma^*$. Note that not all regularity preserving relations are regular—as an example of a regularity preserving, non-regular relation, one can take, e.g., the relation $\{(w, w^R) \mid w \in \Sigma^*\}$, for w^R being the reversal of w, $|\Sigma| > 1$.

2.2 Finite Tree Automata and Transducers

A finite *alphabet* Σ is *ranked* if there exists a *rank* function $\# : \Sigma \to \mathbb{N}$. 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.

A finite ordered *tree* t over a set of labels L is a mapping $t : \mathcal{P}os(t) \to L$ where $\mathcal{P}os(t) \subseteq \mathbb{N}^*$ is a finite, prefix-closed set of *positions* in the tree satisfying (1) for all $p \in \mathcal{P}os(t)$, if $t(p) \in \Sigma_n$ with $n \geq 1$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \{1, \ldots, n\}$ and (2) for all $p \in \mathcal{P}os(t)$, if $t(p) \in \Sigma_0 \cup \chi$, then $\{j \mid pj \in \mathcal{P}os(t)\} = \emptyset$. A term $t \in T_\Sigma[\chi]$ can naturally be also viewed as a tree whose leaves are labelled by constants and variables, and each node with k sons is labelled by a symbol from Σ_k [22]. Therefore, below, we sometimes exchange terms and trees. We denote by $\mathcal{N}l\mathcal{P}os(t) = \{p \in \mathcal{P}os(t) \mid \exists i \in \mathbb{N} : pi \in \mathcal{P}os(t)\}$ the set of *non-leaf* positions.

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 $q, q', q_1, \ldots, q_n \in Q$. Below, we denote bottom-up tree automata simply as tree automata.

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 leaf is a symbol $a \in \Sigma_0$ and there is a rule $a \to_\delta q \in \delta$, the leaf 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 $q, q', q_1, \ldots, q_n \in Q$. 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 $f(q_1(x_1), q_2(x_2), \ldots, q_n(x_n)) \to_\delta q(u) \in \delta$, 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. A transducer τ defines the relation $\varrho_\tau = \{(t, t') \in T_\Sigma \times T_\Sigma \mid t \to^*_\delta q(t') \text{ for some } q \in F\}$. For a set $L \subseteq T_\Sigma$ and a relation $\varrho \subseteq T_\Sigma \times T_\Sigma$, we denote $\varrho(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w', w) \in \varrho\}$ and $\varrho^{-1}(L)$ the set $\{w \in T_\Sigma \mid \exists w' \in L : (w, w') \in \varrho\}$. If τ is a linear tree transducer and L is a regular tree language, then the sets $\varrho_\tau(L)$ and $\varrho_\tau^{-1}(L)$ are regular and effectively constructible [27,22]. Finally, the notions of *regular tree relations* and *regularity preserving tree relations* can be introduced analogously to the word case.

3 Regular Model Checking

3.1 The Basic Idea

As we have already mentioned in the introduction, the basic idea behind regular model checking is to encode particular configurations of the considered systems as

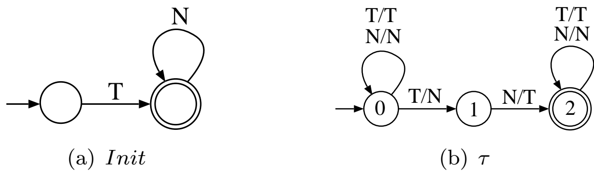


Fig. 1. A model of a simple token passing protocol: (a) an automaton *Init* encoding the initial set of configurations $I = L(Init)$, (b) a transducer τ encoding the 1-step transition relation $\varrho = \varrho_\tau$

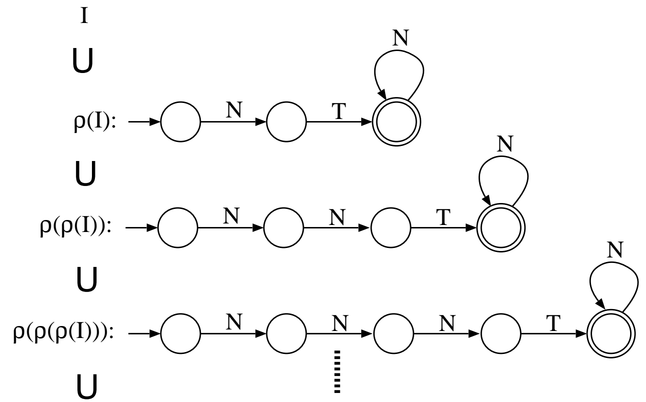


Fig. 2. Divergence of the non-accelerated reachability set computation $\varrho^*(I) = I \cup \varrho(I) \cup \varrho(\varrho(I)) \cup ...$ for the protocol from Fig. 1

words over a suitable finite alphabet and to represent infinite, but regular, sets of such configurations by *finite-state automata*. Transitions between the configurations, constituting the one-step transition relation of the given system, are then encoded using (one or more) *finite-state transducers*, or, more generally, using (one or more) regularity preserving relations expressed, e.g., by specialised automata operations.[3] In this section, we, for simplicity, concentrate on using a single transducer encoding the one-step transition relation of a given system.

Before going into more technical details of regular model checking, we present a simple illustrating example from the area of verification of parametric networks of processes with a linear topology. When dealing with such systems, each letter in a word representing a configuration will typically model the state of a single process, and the length of the word will correspond to the number of processes in the given instance of the system. Let us in particular consider a very simple token passing protocol. We have an arbitrary, but finite number of processes arranged into a linear network. Each process either does not have a token and is waiting for a token to arrive from its left neighbour, or it has a token and then it can pass it to its right neighbour. We suppose that initially there is only one token which is owned by the left-most process. To encode the state of each process in our protocol, we suffice with the alphabet $\Sigma = \{N, T\}$ where N means that the process does not have a token whereas T means the process has a token. Then, the set I of all possible initial configurations can be encoded by the automaton *Init* shown in Fig. 1(a) and the single-step transition relation by the transducer τ in Fig. 1(b).

Once we have a transducer encoding the single-step transition relation ϱ of the system and an automaton encoding its set of initial configurations I, there are two basic strategies we can follow. We can either try to directly compute the set of all reachable configurations $\varrho^*(I)$, or the reachability relation ϱ^* of the system. The set $\varrho^*(I)$ can be obtained by repeatedly applying the single-step transition relation ϱ on the set of the so far reached states and by taking the union of all such sets, i.e., $\varrho^*(I) = I \cup \varrho(I) \cup \varrho(\varrho(I)) \cup ....$ On the other hand, the reachability relation ϱ^* can be obtained by repeatedly composing ϱ with the so far computed reachability relation and by taking the union of all such relations, i.e., $\varrho^* = \iota \cup \varrho \cup (\varrho \circ \varrho) \cup (\varrho \circ \varrho \circ \varrho) \cup ...$ where ι is the identity relation.

The problem is that in the context of parameterised and infinite-state systems, if we try to compute the above infinite unions using a straightforward fixpoint computation, the computation will usually not terminate. We can illustrate this even on our simple token passing protocol. In Fig. 2, we give the first members of the sequence I, $\varrho(I)$, $\varrho(\varrho(I))$, $\varrho(\varrho(\varrho(I)))$, ..., which clearly show that a fixpoint will never be reached (the token can be at the beginning, one step to the right, two steps to the right, three steps to the right, etc.).

In order to make the computation of $\varrho^*(I)$ or ϱ^* terminate at least in many practical cases, we need some kind of *acceleration* of the computation which will allow us to obtain the result of an infinite number of the described computation steps at once (i.e., in some sense, to "jump" to the fixpoint). We can, e.g., notice that in our example, the token is moving step-by-step to the right, and we can accelerate the fixpoint computation by allowing the token to move arbitrarily far to the right in one step. If we use such an acceleration, we will immediately reach the fixpoint shown in Fig. 3(a), which represents the set of all reachable configurations of our protocol. In the literature, several different approaches to a systematic acceleration of fixpoint computations in regular model checking have been proposed. We will very briefly review them in Section 3.3.

3.2 Verification by Regular Model Checking

It is well known that checking of *safety properties* can be reduced to checking that no "bad" states are reachable in the given system. If the set of bad states, for which we want to check that they are not reachable in the given

[3] Several transducers/regularity preserving relations may always be united into a single transducer/regularity preserving relation, respectively. Dealing with one complex or more simple transducers or regularity preserving relations may, however, differ in efficiency in different scenarios.

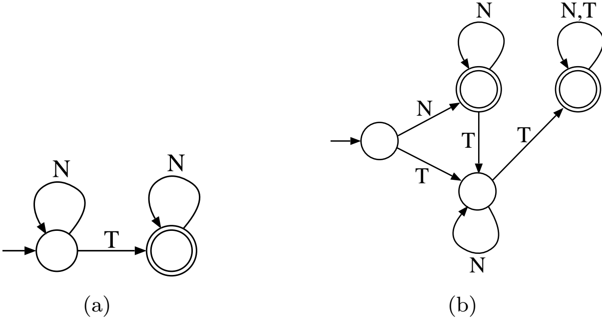


Fig. 3. The simple token passing protocol—automata encoding the set of (a) reachable and (b) bad configurations

system, can be expressed as a regular set B, we may simply compute the reachability set $\varrho^*(I)$ and check that $\varrho^*(I) \cap B = \emptyset$.

For instance, in our simple token passing protocol, we can consider as bad the situation when there is no token in the system or when there appear two or more tokens. The set of such bad states is encoded by the finite-state automaton in Fig. 3(b), and it is clear that its intersection with the set of reachable states from Fig. 3(a) is empty, and thus the system is safe in the given sense.

Checking of *liveness properties* within regular model checking is considerably more difficult. In the world of finite-state systems, it is known that liveness can be reduced to the repeated reachability problem. A similar approach can be taken in the context of regular model checking when the studied systems are modelled by *length-preserving transition relations*, which is typical, e.g., for parameterised networks of processes. In such cases, clearly, the only way how a system can loop is to repeatedly go through some configuration. In a similar way as above, we can then instrument the system by a Büchi automaton[4] (or automata) encoding the undesirable behaviours, and check, e.g., that $\varrho^*(I) \cap A \ \cap$ $domain(\varrho^+ \cap \iota) = \emptyset$. Here, A is the set of accepting configurations, ι is the identity relation, and *domain* is the projection of a relation onto its domain.

Note that in the above described computation, we need to compute not only the reachability set, but also the reachability relation. Nevertheless, this step may be avoided by guessing when an accepting cycle begins, doubling every letter in the given configuration word, then continuing the computation only on the even letters and detecting a closure of the loop by looking for a situation when all the even letters correspond to the odd ones—we have practically tested this technique in some of the experiments presented in Section 4.5 (and it was studied more deeply in [49]).

A systematic framework for modelling parameterised networks of processes as well as specifying their properties to be checked via regular model checking (including liveness properties) has been proposed in [2,3]. The framework uses as a modelling as well as a specification language LTL(MSO) that is a combination of the linear time temporal logic LTL for expressing temporal relations and the monadic second-order logic on words for expressing properties on configuration words. (The MSO part is used for specifying, e.g., that every process in a configuration has to satisfy some condition, or that in the configuration there must exist a process for which some condition holds, and so on.) The work also proposes an automatic translation of the models as well as properties to be checked over them into an automata framework suitable for regular model checking. A computation of the reachability relation is then used for the actual verification.

Finally, checking liveness properties for systems modelled using *non-length-preserving transition relations* is even more complex than checking liveness in the length-preserving case. This is because a non-length-preserving system may exhibit infinite behaviours infinitely going through an accepting state of the monitoring Büchi automaton even when it does not loop at all—it suffices to imagine a system with a queue that keeps growing beyond every bound. For such cases, [17] has proposed an approach based on using regular model checking for automatically computing the greatest simulation relation on the reachable configurations which is compatible with the property being tracked. Then, instead of checking that an accepting configuration can be reached that is reachable from itself too, one checks that an accepting configuration c_1 is reachable from which an accepting configuration c_2 simulating c_1 (i.e., allowing at least the same behaviours from the point of view of the tracked property) is reachable. An alternative approach based on learning fixpoints of specially proposed modalities from their generated samples using language inference algorithms has then been proposed in [56].

3.3 Acceleration in Regular Model Checking

Acceleration methods designed for regular model checking include acceleration schemes [44], quotienting [5], extrapolation [16,39], inference of regular languages [29, 33], and abstraction of automata [15]. The use of abstraction is described in detail in Section 4. A short description of the other methods is given below.

The use of *acceleration schemes* has been proposed in [44]. Acceleration schemes allow one to derive (from the original transitions of a system) meta-transitions encoding the effect of firing some of the original transitions an arbitrary number of times. The work [44] has provided three particular schemes for which it is experimentally checked that they suffice for verification of many cases of parameterised networks of processes. In particular, the following schemes are considered: (1) *local acceleration* allowing an arbitrary number of successive transitions of

[4] Büchi automata are finite automata that accept infinite words by infinitely looping through some of their accepting states (for a formal definition and the associated theory see, e.g., [43]).

a single process to be fired at once, (2) *global acceleration of unary transitions* allowing any number of processes to fire a certain transition in a sequential order within one accelerated step, and (3) *global acceleration of binary transitions* allowing any number of processes to fire in a sequential order two consecutive transitions each—and thus communicate with both of its neighbours—in one atomic step (this way, e.g., a token in a token passing protocol can "jump" any number of positions ahead in one accelerated step). This method has been implemented in the TLV[P] tool [50].

The *quotienting technique* has been elaborated in the works [16,35,41,23,4,5,42]. Let $\tau = (Q, \Sigma, \delta, q_0, F)$ be a length-preserving transducer encoding the single-step transition relation ϱ of a system being examined. The basic idea of the quotienting technique stems from viewing the result of an arbitrary number of compositions of ϱ encoded by τ as an infinite-state "history" transducer $\tau_{hist} = (Q^+, \Sigma, \delta_{hist}, \{q_0\}^+, F^+)$ whose states[5] reflect the history of their creation in terms of which states of τ have been passed at a particular position in a word in the first, second, and further transductions. Therefore, δ_{hist} is defined such that $q_1 q_2 ... q_n \xrightarrow[\tau_{hist}]{a/a'} q'_1 q'_2 ... q'_n$ for some $n \geq 1$ iff there exist $a_1, a_2, ..., a_{n+1} \in \Sigma$ such that $a = a_1$, $a' = a_{n+1}$, and $\forall i \in \{1, ..., n\} : q_i \xrightarrow[\tau]{a_i/a_{i+1}} q'_i$. Intuitively, this means that $q_1 q_2 ... q_n \xrightarrow[\tau_{hist}]{a/a'} q'_1 q'_2 ... q'_n$ represents the composition of the $q_i \xrightarrow[\tau]{a_i/a_{i+1}} q'_i$ transductions for $i = 1, ..., n$. Clearly, τ_{hist} encodes the reachability relation ϱ^+. Of course, the history transducer τ_{hist} is of no practical use as it is infinite-state. The idea is to come up with some *column equivalence* $\simeq$ on its states—i.e., on sequences (or, in the original terminology, columns) of states of the original transducer τ—such that the *quotient transducer* $\tau_{hist/\simeq}$ is (1) finite-state as often as possible, and at the same time, (2) describes exactly the same relation as τ_{hist}. Suitable column equivalences have been proposed along with ways on how to build the quotient transducer incrementally (e.g., by gradually adding new transitions obtained by composing transitions as described above while also gradually quotienting the automaton—obviously, one cannot construct a history transducer and only then quotient it).

The *extrapolation* (or *widening*) approach to regular model checking [16,10] is based on comparing successive elements of the sequence I, $\varrho(I)$, $\varrho(\varrho(I))$, ..., trying to find some repeated growth pattern in it, and adding an arbitrary number of its occurrences into the reachability set. In particular, following [16], let $L \subseteq \Sigma^*$ be a so far computed reachability set and $\varrho \subseteq \Sigma^* \times \Sigma^*$ a regular one-step transition relation. One can check whether there are regular sets L_1, L_2, and Δ satisfying the following two conditions: (C1) $L = L_1.L_2$ and $\varrho(L) = L_1.\Delta.L_2$ and (C2) $L_1.\Delta^*.L_2 = \varrho(L_1.\Delta^*.L_2) \cup L$. If the conditions hold, $L_1.\Delta^*.L_2$ is added to the so far computed reachability set. Intuitively, C1 means that the effect of applying ϱ is to add Δ between L_1 and L_2. C2 then ensures that $\varrho^*(L) \subseteq L_1.\Delta^*.L_2$, and so we add at least all the configurations reachable from L by iterating ϱ. Note that the exactness of the acceleration—i.e., whether $L_1.\Delta^*.L_2 \subseteq \varrho^*(L)$ holds too—is not guaranteed in general. However, [16] gives a sufficient condition on ϱ under which C1 and C2 lead to an exact acceleration. This condition in particular requires ϱ to be *well-founded*, i.e., not allowing any word to have an infinite number of predecessors wrt. ϱ. There is also a syntactic criterion for the so called *simple rewriting relations* that are guaranteed to satisfy the above condition and that seem to appear quite often in practice.

Regular model checking based on *inference of regular languages* was studied in [33] extending [29]. Here, an important observation is that, for an infinite-state system whose behaviour is described by a length-preserving transducer τ, a set containing all reachable words up to the given length can be computed by a simple iterative application of τ on the set of initial configurations. These configurations are taken as a sample. Then some language inference algorithm may be applied to learn the whole reachable set (or its overapproximation) from this sample. As shown in [33], termination of the method is guaranteed whenever the set of all reachable configurations is regular. This is not the case for other acceleration methods. In [55,54,56,57], similar results were proved, covering even omega-regular model checking and checking of branching-time properties.

4 Abstract Regular Model Checking

Apart from the need to accelerate the reachability computation to make it terminate in as many practical scenarios as possible, another crucial problem to be faced in regular model checking is the *state space explosion in automata (transducer) representations* of the sets of configurations (or reachability relations) being examined. One of the sources of this problem is related to the nature of the previously mentioned regular model checking techniques. Typically, these techniques try to calculate the *exact* reachability sets (or relations) independently of the property being verified. However, it is often enough to only compute an overapproximation of the reachability set (or relation) precise enough just to verify the given property of interest. Indeed, this is the way how large (or infinite) state spaces are often being successfully handled outside the domain of regular model checking using the so called *abstract-check-refine* paradigm often implemented in the form of some *counterexample guided abstraction refinement* (CEGAR) loop [30,8,48,21,34,24].

Inspired by the above, we have proposed in [15] a new approach to regular model checking which is based on

[5] We allow here a set of initial states.

the abstract-check-refine paradigm. Instead of a precise acceleration, we use abstract fixpoint computations in some *finite* domain of automata. As we have already briefly mentioned in the introduction, the abstract fixpoint computations always terminate and provide overapproximations of the reachability sets (relations). To achieve this, we define techniques that systematically map any automaton M to an automaton M' from some finite domain such that M' recognises a superset of the language of M.

The abstraction techniques we discuss below take into account the structure of automata and are based on collapsing their states according to some equivalence relation. The first one is inspired by predicate abstraction. We consider a state q of an automaton M to "satisfy" a predicate language L_P if the intersection of L_P with the language $L(M, q)$ accepted from the state q is not empty. Subsequently, two states are equivalent if they satisfy the same predicates. The second abstraction technique is then based on considering two automata states equivalent if their *languages of words up to a certain fixed length* are equal. For both of these two abstraction methods, we provide effective refinement techniques allowing us to discard spurious counterexamples.

We also introduce several natural alternatives to the above basic approaches, based on backward and/or trace languages of states of automata. For them, it is not always possible to guarantee the exclusion of a spurious counterexample, but according to our experience, they still provide good practical results.

All of our techniques can be applied to dealing with reachability sets (obtained by iterating length-preserving or even general transducers) as well as length-preserving reachability relations.

4.1 A Running Example and Some Basic Assumptions

As a simple running example capable of illustrating the different techniques of abstract regular model checking that we discuss here, we consider a slight modification of the token passing protocol from Fig. 1. The modification consists in that each process can pass the token to its *third* right neighbour (instead of its direct right neighbour). The one-step transition relation of the system is encoded by the transducer τ in Fig. 4 (a). The transducer includes the identity relation too. In the initial configurations described by the automaton *Init* from Fig. 4 (c), the second process has the token, and the number of processes is divisible by three. We want to show that it is not possible to reach any configuration where the last process has the token. This set is described by the automaton *Bad* from Fig. 4 (b).

Note that in the following, in order to shorten the descriptions, we *identify a transducer and the relation it represents* and write $\tau(L)$ instead of $\varrho_\tau(L)$. Let $\iota \subseteq \Sigma^* \times \Sigma^*$ be the identity relation and $\circ$ the composition

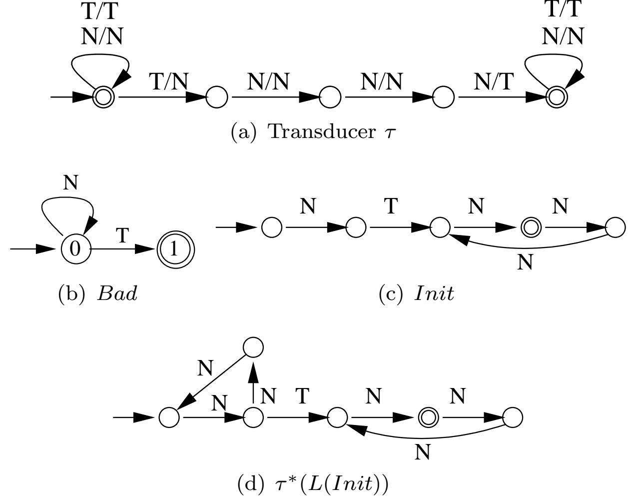


Fig. 4. A transducer τ modelling a modified token passing protocol and automata describing the initial, bad, and reachable configurations of the system

of relations. We define recursively the relations (transducers) $\tau^0 = \iota$, $\tau^{i+1} = \tau \circ \tau^i$, and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. As in our running example, we suppose $\iota \subseteq \tau$ for the rest of the section meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

For our running example, $\tau^*(L(Init))$ is depicted in Fig. 4(d), and the property of interest clearly holds. However, in general, $\tau^*(L(Init))$ is neither guaranteed to be regular nor computable. In the following, the verification task is thus to find a regular overapproximation $L \supseteq \tau^*(L(Init))$ such that $L \cap L(Bad) = \emptyset$.

4.2 The Method of Abstract Regular Model Checking

We now describe the general principle of abstract regular model checking (ARMC) using a generic framework for automata abstraction based on collapsing states of the automata. This framework is then instantiated in several concrete ways in the following two sections. For simplicity, we restrict to the case where the one-step transition relation of the system at hand is given by a single transducer (the more general cases being analogical). Moreover, we concentrate on the use of ARMC for computing reachability sets only. However, ARMC can be applied for dealing with reachability relations too—though in the context of length-preserving transducers only[6].

4.2.1 The Basic Framework of Automata Abstraction

Let Σ be a finite alphabet and $\mathbb{M}_\Sigma$ the set of all finite automata over Σ. By an *automata abstraction function* α, we understand a function that maps every automaton M over Σ to an automaton $\alpha(M)$ whose language is an overapproximation of the one of M. To be more precise,

[6] Indeed, length-preserving transducers over an alphabet Σ can be seen as finite-state automata over $\Sigma \times \Sigma$.

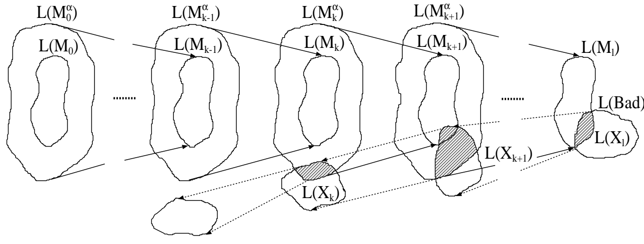


Fig. 5. A spurious counterexample in an abstract regular fixpoint computation

for some abstract domain of automata $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$, α is a mapping $\mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ such that $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. We call α *finitary* iff its range $\mathbb{A}_\Sigma$ is finite.

Working conveniently on the level of automata, given a transition relation expressed as a transducer τ over Σ and an automata abstraction function α, we introduce the *abstract transition function* τ_α as follows: For each automaton $M \in \mathbb{M}_\Sigma$, $\tau_\alpha(M) = \alpha(\hat{\tau}(M))$ where $\hat{\tau}(M)$ is the minimal deterministic automaton of $\tau(L(M))$[7]. Now, we can iteratively compute the sequence $(\tau_\alpha^i(M))_{i\geq 0}$. Since we suppose $\iota \subseteq \tau$, it is clear that if α is finitary, there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(M) = \tau_\alpha^k(M)$. The definition of α implies $L(\tau_\alpha^k(M)) \supseteq \tau^*(L(M))$. This means that in a finite number of steps, we can compute an overapproximation of the reachability set $\tau^*(L(M))$.

4.2.2 Refining Automata Abstractions

We call an automata abstraction function α' a *refinement* of α iff $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. A need to refine α arises when a situation depicted in Fig. 5 happens. Suppose we are checking whether no configuration from the set described by some automaton Bad is reachable from some given set of initial configurations described by an automaton M_0. We suppose $L(M_0) \cap L(Bad) = \emptyset$—otherwise the property being checked is broken already by the initial configurations. Let $M_0^\alpha = \alpha(M_0)$ and for each $i > 0$, $M_i = \hat{\tau}(M_{i-1}^\alpha)$ and $M_i^\alpha = \alpha(M_i) = \tau_\alpha(M_{i-1}^\alpha)$. There exist k and l ($0 \leq k < l$) such that: (1) $\forall i : 0 \leq i < l : L(M_i) \cap L(Bad) = \emptyset$. (2) $L(M_l) \cap L(Bad) = L(X_l) \neq \emptyset$. (3) If we define X_i as the minimal deterministic automaton accepting $\tau^{-1}(L(X_{i+1})) \cap L(M_i^\alpha)$ for all i such that $0 \leq i < l$, then $\forall i : k < i < l : L(X_i) \cap L(M_i) \neq \emptyset$ and $L(X_k) \cap L(M_k) = \emptyset$ despite $L(X_k) \neq \emptyset$. Next, we see that either $k = 0$ or $L(X_{k-1}) = \emptyset$, and it is clear that we have encountered a *spurious counterexample.*

Note that when no l can be found such that $L(M_l) \cap L(Bad) \neq \emptyset$, the computation eventually reaches a fixpoint, and the property is proved to hold. On the other hand, if $L(X_0) \cap L(M_0) \neq \emptyset$, we have proved that the property is broken.

The *spurious counterexample may be eliminated* by refining α to α' such that for any automaton M whose language is disjoint with $L(X_k)$, the language of its α'-abstraction will not intersect $L(X_k)$ either. Then, the same faulty reachability computation (i.e., the same sequence of M_i and M_i^α) may not be repeated because we exclude the abstraction of M_k to M_k^α. Moreover, reachability of the bad configurations is in general excluded unless there is another reason for it than overapproximating by subsets of $L(X_k)$.

A slightly *weaker way of eliminating the spurious counterexample* consists in refining α to α' such that at least the language of the abstraction of M_k does not intersect with $L(X_k)$. In such a case, it is not excluded that some subset of $L(X_k)$ will again be used for an overapproximation somewhere, but we still exclude a repetition of exactly the same faulty computation. The obtained refinement can be coarser, which may lead to more refinements and a slower computation. On the other hand, the computation may terminate sooner due to quickly jumping to the fixpoint and use less memory due to working with less structured sets of configurations of the systems being verified—the abstraction is prevented from becoming unnecessarily precise in this case. For the latter reason, as illustrated later, one may sometimes successfully use even some more heuristic approaches that guarantee that the spurious counterexample will only eventually be excluded (i.e., after a certain number of refinements) or that do not guarantee the exclusion at all.

An obvious danger of using a heuristic approach that does not guarantee an exclusion of spurious counterexamples is that the computation may easily start looping. Notice, however, that even when we refine automata abstractions such that spurious counterexamples are always excluded, and the computation does not loop, we do not guarantee that it will eventually stop—we may keep refining forever. Indeed, the verification problem we are solving is undecidable in general.

4.2.3 Abstracting Automata by Collapsing Their States

In the following two sections, we discuss several concrete automata abstraction functions. They are based on automata state equivalence schemas that define for each automaton from $\mathbb{M}_\Sigma$ an equivalence relation on its states. An automaton is then abstracted by collapsing all its states related by this equivalence. We suppose such an equivalence to reflect the fact that the future and/or history of the states to be collapsed is close enough, and the difference may be abstracted away.

Formally, an *automata state equivalence schema* $\mathbb{E}$ assigns an automata state equivalence $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ to each finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ over Σ. We define the *automata abstraction function* $\alpha_{\mathbb{E}}$ based on $\mathbb{E}$ s.t. $\forall M \in \mathbb{M}_\Sigma : \alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$. We call $\mathbb{E}$ *finitary* iff $\alpha_{\mathbb{E}}$ is finitary. We *refine* $\alpha_{\mathbb{E}}$ by refining $\mathbb{E}$ such that more states are distinguished in at least some automata.

The automata state equivalence schemas presented below are then all based on one of the following two ba-

[7] A generalisation of ARMC to dealing with nondeterministic automata is possible—cf. [11].

sic principles: (1) comparing states wrt. the intersections of their forward/backward languages with some *predicate languages* (represented by *predicate automata*) and (2) comparing states wrt. their forward/backward behaviours up to a certain *bounded length*.

4.3 State Equivalences Based on Predicate Languages

We start by introducing two automata state equivalence schemas defined wrt. a finite set of *predicate languages* represented by a set $\mathcal{P}$ of finite automata, which we denote as predicate automata. Namely, we introduce the schema $\mathbb{F}_\mathcal{P}$ based on forward languages of states and the schema $\mathbb{B}_\mathcal{P}$ based on backward languages. They compare two states of a given automaton according to the intersections of their forward/backward languages with the predicates.[8] Below, we first introduce the basic principles of the schemas and then add some implementation and optimisation notes.

4.3.1 The $\mathbb{F}_\mathcal{P}$ Automata State Equivalence Schema

The automata state equivalence schema $\mathbb{F}_\mathcal{P}$ defines two states of a given automaton to be equivalent when their languages have a *nonempty intersection with the same predicates* of $\mathcal{P}$. Formally, for an automaton $M = (Q, \Sigma, \delta, q_0, F)$, $\mathbb{F}_\mathcal{P}$ 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, as $\mathcal{P}$ 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, $\mathbb{F}_\mathcal{P}$ is *finitary.*

For our example from Fig. 4, if we take as $\mathcal{P}$ the automata of the languages of the states of *Bad*, the automaton *Init* from Fig. 4(c) is abstracted as follows: All states of *Init* except the final one become equivalent since their languages have all empty intersections with the languages accepted from states 0 and 1 of *Bad*. Hence, when equivalent states are collapsed, we obtain the automaton in Fig. 6(a), which after determinisation and minimisation gives the automaton in Fig. 6(b). Then, the intersection of $\hat{\tau}(\alpha(Init))$ with the bad configurations—cf. Fig. 6(d)—is not empty, and we have to refine the abstraction.

The $\mathbb{F}_\mathcal{P}$ schema may be *refined by adding new predicates* into the current set of predicates $\mathcal{P}$. In particular, we can extend $\mathcal{P}$ by automata corresponding to the languages of all the states in X_k from Fig. 5. Theorem 1 shows that this prevents abstractions of languages disjoint with $L(X_k)$, such as—but not only—$L(M_k)$, from intersecting with $L(X_k)$. Consequently, as we have already explained, a repetition of the same faulty computation is excluded, and the set of bad configurations will not be reached unless there is another reason for this than overapproximating by subsets of $L(X_k)$.

[8] The use of intersection with predicate languages needs not be the only possible way of constructing some predicate language abstraction. Proposing a different abstraction based on predicate languages may be an interesting subject for further work. However, such an abstraction should come with some way of counterexample-guided refinement. This is not straightforward and we are currently not aware of any other refinable abstractions based on predicate languages than using the $\mathbb{F}_\mathcal{P}$ and $\mathbb{B}_\mathcal{P}$ schemas.

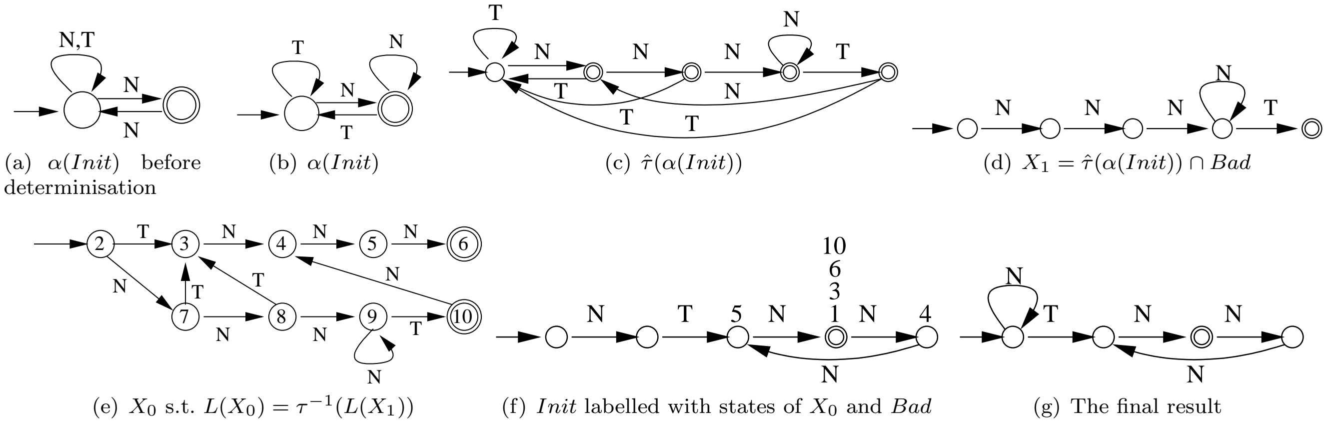
Theorem 1. *Let $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ and $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ be any two finite automata and let $\mathcal{P}$ be a finite set of predicate automata such that $\forall q_X \in Q_X : \exists P \in \mathcal{P} : L(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X) = \emptyset$ too.*

Proof. We prove the theorem by contradiction. Suppose $L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X) \neq \emptyset$. Let $w \in L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X)$. As w is accepted by $\alpha_{\mathbb{F}_\mathcal{P}}(M)$, 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 prefix of w 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).

Suppose that the minimum number of jumps needed to accept a word from $L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X)$ in M is i with $i > 0$, and let w' be such a word. Let the last jump within accepting w' in M be from some state $q_1 \in Q_M$ to some $q_2 \in Q_M$ such that $q_1 \sim_M^\mathcal{P} q_2$. Let $w' = w_1 w_2$ such that w_1 is read (possibly with jumps) just before the jump from q_1 to q_2. Clearly, $q_2 \xrightarrow[M]{w_2} q_3$ for some $q_3 \in F_M$. We know that X accepts w'. Suppose that after reading w_1, it is in some $q_X \in Q_X$. As $w_2 \in L(X, q_X)$ and $w_2 \in L(M, q_2)$, $L(M, q_2) \cap L(P) \neq \emptyset$ for the predicate(s) $P \in \mathcal{P}$ for which $L(P) = L(X, q_X)$. Moreover, as $q_1 \sim_M^\mathcal{P} q_2$, $L(M, q_1) \cap L(P) \neq \emptyset$ too. This implies there exists $w_2' \in L(P)$ such that $w_2' \in L(M, q_1)$ and $w_2' \in L(X, q_X)$. However, this means that $w_1 w_2' \in L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X)$ 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. □

In our example, we refine the abstraction by extending $\mathcal{P}$ with the automata representing the languages of the states of X_0 from Fig. 6(e). Fig. 6(f) then indicates, for each state q of *Init*, the predicates corresponding to the states of *Bad* and X_0 whose languages have a nonempty intersection with the language of q. For example, the third state from the left of *Init* is labelled by 5 because it accepts N which is also accepted by state 5 of X_0. The first two states of *Init* are equivalent and are collapsed to obtain the automaton from Fig. 6(g), which is a fixpoint showing that the property is verified. Notice that it is an overapproximation of the set of reachable configurations from Fig. 4(d).

The price of refining $\mathbb{F}_\mathcal{P}$ by adding predicates for all the states in X_k may seem prohibitive, but fortunately this is not the case in practice. As described later on in Section 4.3.3, we do not have to treat all the new predicates separately. We exploit the fact that they come from one original automaton and share large parts of their



(a) $\alpha(Init)$ before determinisation (b) $\alpha(Init)$ (c) $\hat{\tau}(\alpha(Init))$ (d) $X_1 = \hat{\tau}(\alpha(Init)) \cap Bad$

(e) X_0 s.t. $L(X_0) = \tau^{-1}(L(X_1))$ (f) $Init$ labelled with states of X_0 and Bad (g) The final result

Fig. 6. An example using abstraction based on predicate languages

structure. In fact, we can work just with the original automaton and each of its states may be considered an initial state of some predicate. This way, adding the original automaton as the only predicate and adding predicates for all of its states becomes roughly equal. Moreover, the refinement may be weakened by taking into account just some states of X_k as discussed later on.

4.3.2 The $\mathbb{B}_{\mathcal{P}}$ Automata State Equivalence Schema

The $\mathbb{B}_{\mathcal{P}}$ automata state equivalence schema is an alternative of $\mathbb{F}_{\mathcal{P}}$ using *backward languages of states* rather than the forward ones. For an automaton $M = (Q, \Sigma, \delta, q_0, F)$, it defines the state equivalence as the equivalence $\overleftarrow{\sim}^{\mathcal{P}}_{M}$ such that $\forall q_1, q_2 \in Q : q_1 \overleftarrow{\sim}^{\mathcal{P}}_{M} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap \overleftarrow{L}(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap \overleftarrow{L}(M, q_2) \neq \emptyset)$.

Clearly, $\mathbb{B}_{\mathcal{P}}$ is *finitary* for the same reason as $\mathbb{F}_{\mathcal{P}}$. It may also be *refined* by extending $\mathcal{P}$ by automata corresponding to the languages of all the states in X_k from Fig. 5. Theorem 2 shows that the effect is the same as for $\mathbb{F}_{\mathcal{P}}$.

Theorem 2. *Let $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ and $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ be any two finite automata and let $\mathcal{P}$ be a finite set of predicate automata such that $\forall q_X \in Q_X : \exists P \in \mathcal{P} : \overleftarrow{L}(X, q_X) = L(P)$. Then, if $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{B}_{\mathcal{P}}}(M)) \cap L(X) = \emptyset$ too.*

Proof. The theorem can be proved by contradiction in a similar way as Theorem 1. This time, as a consequence of working with backward languages of states, we do not deal with the last jump, but the first jump in accepting some $w' \in L(\alpha_{\mathbb{B}_{\mathcal{P}}}(M)) \cap L(X)$ in M. We do not look for a replacement w_2' of w_2 to be accepted from q_1 instead of q_2, but for a replacement w_1' of w_1 to be accepted before q_2 rather than before q_1. □

4.3.3 Optimising Collapsing Based on $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$

The abstraction of an automaton M wrt. the automata state equivalence schema $\mathbb{F}_{\mathcal{P}}$ can be implemented by first labelling states of M by the states of predicate automata in $\mathcal{P}$ with whose languages they have a non-empty intersection and then collapsing the states of M that are labelled by the initial states of the same predicates (provided the sets of states of the predicate automata are disjoint). The labelling can be done in a way similar to constructing a backward synchronous product of M with the particular predicate automata: (1) $\forall P \in \mathcal{P}\ \forall q_F^P \in F_P\ \forall q_F^M \in F_M$: q_F^M is labelled by q_F^P, and (2) $\forall P \in \mathcal{P}\ \forall q_1^P, q_2^P \in Q_P\ \forall q_1^M, q_2^M \in Q_M$: if q_2^M is labelled by q_2^P, and there exists $a \in \Sigma$ such that $q_1^M \xrightarrow[\delta_M]{a} q_2^M$ and $q_1^P \xrightarrow[\delta_P]{a} q_2^P$, then q_1^M is labelled with q_1^P. The abstraction of an automaton M wrt. the $\mathbb{B}_{\mathcal{P}}$ schema can be implemented analogously.

If the above construction is used, it is then clear that when *refining* $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$, we can just add X_k into $\mathcal{P}$ and modify the construction such that in the collapsing phase, we simply take into account all the labels by states of X_k and do not ignore the (anyway constructed) labels other than $q_0^{X_k}$.

Moreover, we can try to optimise the refinement of $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ by replacing X_k in $\mathcal{P}$ by its *important tail/head part* defined wrt. M_k as the subautomaton of X_k based on the states of X_k that appear in at least one of the labels of M_k wrt. $\mathbb{F}_{\mathcal{P} \cup \{X_k\}}/\mathbb{B}_{\mathcal{P} \cup \{X_k\}}$, respectively. The effect of such a refinement corresponds to the weaker way of refining automata abstraction functions described in Section 4.2.2.[9] This is due to the strong link of the important tail/head part of X_k to M_k wrt. which it is computed. A repetition of the same faulty computation is then excluded, but the obtained abstraction is coarser, which may sometimes speed up the computation as we have already discussed.

A further possible heuristic to optimise the refinement of $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ is trying to find just one or two *key states* of the important tail/head part of X_k such that if

[9] The key last/first jump in an accepting run of M mentioned in the proofs of Theorems 1, 2 is between states that can be labelled by some states of X. The concerned states of X are thus in the important tail/head part of X, and the proof construction of Theorems 1, 2 can still be applied.

their languages are considered in addition to $\mathcal{P}$, $L(M_k^\alpha)$ will not intersect $L(X_k)$.

We close the section by noting that in the *initial set of predicates* $\mathcal{P}$ of $\mathbb{F}_\mathcal{P}/\mathbb{B}_\mathcal{P}$, we may use, e.g., the automata describing the set of bad configurations and/or the set of initial configurations. Further, we may also use the domains or ranges of the transducers encoding the particular transitions in the systems being examined (whose union forms the one-step transition relation τ which we iterate). The meaning of the latter predicates is similar to using guards or actions of transitions in predicate abstraction [8].

4.4 State Equivalences Using Finite-Length Languages

We now present the approach of defining automata state equivalence schemas which is based on comparing automata states wrt. a certain bounded part of their languages. It is a simple, yet (according to our practical experience) often quite efficient approach. As a basic representative of this kind of schemas, we first present the schema $\mathbb{F}_n^L$ based on forward languages of words of a limited length. Then, we discuss its possible alternatives.

The $\mathbb{F}_n^L$ automata state equivalence schema defines two states of an automaton to be equal if their *languages of words of length up to a certain bound* n are identical. Formally, for an automaton $M = (Q, \Sigma, \delta, q_0, F)$, $\mathbb{F}_n^L$ defines the state equivalence as the equivalence $\sim_M^n$ such that $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$.

$\mathbb{F}_n^L$ is clearly *finitary*. It may be *refined* by incrementally increasing the bound n on the length of the words considered. This way, since we work with minimal deterministic automata, we may achieve the weaker type of refinement described in Section 4.2.2. Such an effect is achieved when n is increased to be equal or bigger than the number of states in M_k from Fig. 5 minus one. In a minimal deterministic automaton, this guarantees that all states are distinguishable wrt. $\sim_M^n$, and M_k will not be collapsed at all.

In Fig. 7, we apply $\mathbb{F}_n^L$ to the example from Fig. 4. We choose $n = 2$. In this case, the abstraction of the $Init$ automaton is $Init$ itself. Fig. 7(a) indicates the states of $\hat{\tau}(Init)$ that have the same languages of words up to size 2 and are therefore equivalent. Collapsing them yields the automaton shown in Fig. 7(b) (after determinisation and minimisation), which is a fixpoint. Notice that it is a different overapproximation of the set of reachable configurations than the one obtained using $\mathbb{F}_\mathcal{P}$. If we choose $n = 1$, we obtain a similar result, but we need one refinement step of the above described kind.

Let us, however, note that according to our practical experience, the increment of n by $|Q_M| - 1$ may often be too big. Alternatively, one may use a fraction of it (e.g., one half), increase n by the number of states in X_k (or a fraction of it), or increase n just by one. In such cases,

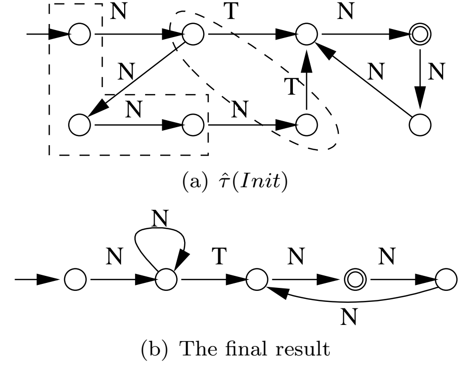


Fig. 7. An example using abstraction based on languages of words up to length n (for $n = 2$)

an immediate exclusion of the faulty run is not guaranteed, but clearly, such a computation will be *eventually excluded* because n will sooner or later reach the necessary value. The impact of working with abstractions refined in a coarser way is then like in the case of using $\mathbb{F}_\mathcal{P}/\mathbb{B}_\mathcal{P}$.

Regarding the *initial value of* n, one may use, e.g., the number of states in the automaton describing the set of initial configurations or the set of bad configurations, their fraction, or again just one.

As a natural alternative to dealing with forward languages of words of a limited length, one may also use *backward languages of words* of a limited length and forward/backward *languages of traces* of a limited length. The automata equivalence schemas $\mathbb{B}_n^L$, $\mathbb{F}_n^T$, as well as $\mathbb{B}_n^T$ based on them can be formally defined analogously to $\mathbb{F}_n^L$.

Clearly, all these schemas are *finitary*. Moreover, we can *refine* them in a similar way as $\mathbb{F}_n^L$. For $\mathbb{F}_n^T$ and $\mathbb{B}_n^T$, however, no guarantee of excluding a spurious counterexample may be provided. Using $\mathbb{F}_n^T$, e.g., we can never distinguish the last three states of the automaton in Fig. 7(b)—they all have the same trace languages. Thus, we cannot remember that the token cannot get to the last process. Nevertheless, despite this, our practical experience shows that the schemas based on traces may be quite successful in practice.

4.5 Experiments with Abstract Regular Model Checking

We have implemented the ideas described above in a prototype tool written in YAP Prolog using the FSA library [53]. To demonstrate that abstract regular model checking is applicable to verification of a broad variety of systems, we tried to apply the tool to a number of different verification tasks.

4.5.1 The Types of Systems Verified

Parameterised Networks of Processes. We have considered several somewhat idealised *mutual exclusion algorithms* for arbitrarily many processes (namely, the Bak-

ery, Burns, Dijkstra, and Szymanski algorithms in versions similar to [41]). In most of these systems, the particular processes are finite-state. We encode their global configurations by words whose length corresponds to the number of participating processes, and each letter represents the local state of some process. In the case of the Bakery algorithm where each process contains an unbounded ticket value, this value is not represented directly, but encoded in the ordering of the processes in the word.

We verified the mutual exclusion property of the algorithms, and for the Bakery algorithm, we verified that some process will always eventually get to the critical section (communal liveness) as well as that each individual process will always eventually get there (individual liveness) under suitable fairness assumptions. For checking liveness, we manually composed the appropriate Büchi automata with the system being verified. Loop detection was allowed by working with pairs of configurations consisting of a remembered potential beginning of a loop (fixed at a certain—randomly chosen—point of time) and the current configuration. Checking that a loop is closed then consisted in checking that a pair of the same configurations was reached. To encode the pairs of configurations using finite automata, we interleaved their corresponding letters.

Push-down Systems. We considered a simple system of *recursive procedures*—the plotter example from [28]. We verified a safety part of the original property of interest describing the correct order of plotter instructions to be issued. In this case, we use words to encode the contents of the stack.

Systems with Queues. We experimented with a model of the Alternating Bit Protocol (ABP) for which we checked correctness of the delivery order of the messages. A word encoding a configuration of the protocol contained two letters representing internal states of the communicating processes. Moreover, it contained the contents of the two *lossy communication channels* with a letter corresponding to each message. Let us note that in this case, as well as in the above and below cases, general (non-length-preserving) transducers were used to encode transitions of the systems.

Petri Nets and Systems with Counters. We examined a general *Petri net* with inhibitor arcs, which can be considered an example of a system with *unbounded counters* too. In particular, we modelled a Readers/Writers system extended with a possibility of dynamic creation and deletion of processes, for which we verified mutual exclusion between readers and writers and between multiple writers. We considered a correct version of the system as well as a faulty one, in which we omitted one of the Petri net arcs. Markings of places in the Petri net were encoded in unary, and the particular values were put in parallel[10]. Further, we also considered the Bakery algorithm for two processes modelled as a counter automaton with two unbounded counters. For the actual verification, a binary encoding of the values of counters like in NDDs [59] was successfully used.

```
1: x = NULL;
2: while (list → next) {
3:     y = list → next;
4:     list → next = x;
5:     x = list;  list = y;
6: }
7: list → next = x;
```

Fig. 8. Nonempty list reversal

Dynamic Linked Data Structures. As a representative case study, we considered verification of a *procedure for reversing* (non-empty) *singly-linked lists*—cf. Fig. 8.

When abstracting the memory manipulated by the procedure, we focused on the cases where in the memory there are at most two linked lists linking consecutive cells, the first list in a descending way and the second one in an ascending way. We represented configurations of the procedure as words over the following alphabet: List items were represented by symbols $\underline{i}$, left/right pointers by $<$/$>$, pointer variables were represented by their names (*list* is shortened to l), and $\underline{o}$ was used to represent the memory outside the list. Moreover, we used symbols $\underline{iv}$ (resp. $\underline{ov}$) to denote that v points to i (resp. outside the list). We used $|$ to separate the ascending and descending lists. Pointer variables pointing to null were not present in the configuration representations. A typical abstraction of the memory then looked like $\underline{i} < \underline{i} < \underline{i} \mid \underline{il} > \underline{i}\ \underline{ox}$ where the first list contains three items, the second one two, *list* points to the beginning of the second list, x points outside the two lists, and y points to null. For such an abstraction of the memory contents (prefixed with the current control line), it is not difficult to associate transducers with each command of the procedure. For example, the transducer corresponding to the command $list \rightarrow next := x$ at line 4 transforms a typical configuration $4\ \underline{i} < \underline{ix} \mid \underline{il} > \underline{iy} > \underline{i}\ \underline{o}$ to the configuration $5\ \underline{i} < \underline{ix} < \underline{il} \mid \underline{iy} > \underline{i}\ \underline{o}$ (the successor of the item pointed to by l is not anymore the one pointed to by y, but the one pointed to by x). Then, the transducer τ corresponding to the whole procedure is the union of the transducers of all the commands.

If the memory contents did not fit the above described form, it would be abstracted to a single word with the "don't know" meaning. However, starting from configurations like $1\ \underline{il} > \underline{i} > \underline{i}\ \underline{o}$ or $1\ \underline{i} < \underline{i} < \underline{il}\ \underline{o}$, the verification showed that such a situation could not happen. Via a symmetry argument exploiting the fact that the procedure never refers to concrete addresses, the results

[10] Using this encoding, a marking of a net with places p and q, two tokens in p, and four in q is written as $q|q|pq|pq$.

Table 1. Results of experimenting with abstract regular model checking using the finite-length-languages-based abstractions

Experiment	$\mathbb{F}_n^L/\mathbb{F}_n^T/\mathbb{B}_n^L/\mathbb{B}_n^T$	T_{best}
Bakery	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert/2$	0.02
Bakery/comm. liv.	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert$	0.14
Bakery/ind. liv.	Fw, $\mathbb{F}_n^T$, 1	8.66
Bakery – counters	Bw, $\mathbb{B}_n^L$, $\lvert Q_{Bad}\rvert$	0.08
ABP	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.32
Burns	Fw, $\mathbb{B}_n^T$, 1	0.31
Dijkstra	Fw, $\mathbb{F}_n^T$, 1	1.75
PDS	Bw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.02
Petri net/Read. Wr.	Fw, $\mathbb{B}_n^T$, special n	21.07
Faulty PN/Rd. Wr.	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert$	0.73
Szymanski	Fw, $\mathbb{B}_n^T$, 1	0.25
Rev. Lists	Fw,$\mathbb{F}_n^L$, $\lvert Q_{Init}\rvert/2+\lvert Q_{X_k}\rvert/2$	0.61
Rev. Lists/Transd.	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Init}\rvert/2$	21.79

Table 2. Results of experimenting with abstract regular model checking using the predicate-based abstractions

Experiment	$\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$	T_{best}
Bakery	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$	0.02
Bakery/comm. liv.	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad\lvert Grd]$	0.13
Bakery/ind. liv.	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$, Key St.	19.41
Bakery – counters	Bw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\lvert Grd]$	0.09
ABP	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init\lvert Grd]$	0.68
Burns	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.06
Dijkstra	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.73
PDS	Bw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$	0.02
Petri net/Read. Wr.	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\lvert Grd]$	5.86
Faulty PN/Rd. Wr.	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init\lvert Grd]$	0.81
Szymanski	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Init\lvert Grd]$	0.55
Rev. Lists	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\lvert Grd\lvert Act]$	1.29
Rev. Lists/Transd.	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init\lvert Grd\lvert Act]$	42.60

of the verification easily generalise to lists with items stored at arbitrary memory locations.

By computing an abstraction of the reachability set $\tau^*(Init)$, we checked that the procedure outputs a list. Moreover, by computing an overapproximation of the reachability relation τ^* of the system, we checked that the output list has the same length as the input one.

In [12], a generalised encoding for 1-selector linked structures was provided and various list-manipulating procedures were successfully verified. Moreover, later, abstract regular *tree* model checking was used in [14] for verification of programs with dynamic data structures with more selectors and various complicated topologies. In this paper in Section 6, we concentrate on the latter, more recent and more general approach.

4.5.2 A Summary of the Results

The efficiency of using the $\mathbb{F}_n^L$, $\mathbb{F}_n^T$, $\mathbb{B}_n^L$, or $\mathbb{B}_n^T$ automata state equivalence schemas heavily depends on the *choice of the initial value of* n and the *strategy of increasing it.* In our experiments, we have tried $|Q_{Bad}|$, $|Q_{Bad}|/2$, $|Q_{Init}|$, $|Q_{Init}|/2$, and 1 as the initial value of n and $|Q_{M_k}|$, $|Q_{M_k}|/2$, $|Q_{X_k}|$, $|Q_{X_k}|/2$, and 1 as its increment. The results we obtained are summarised in Table 1. In the table, we always mention the scenario for which we obtained the shortest execution time[11]. We first say whether it was in a forward or backward computation (i.e., starting from the initial configurations or the "bad" ones), then the automata state equivalence schema used, followed by the initial value of n, and if it was needed, the increment of n written behind a plus symbol. In the case of the Readers/Writers example, the time consumption was relatively high, and we tried to iteratively find a value of n for which it was the best.

[11] In some cases, a few scenarios gave a very similar result out of which just one is mentioned.

Similarly to the above, the efficiency of using the $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ automata state equivalence schemas depends a lot on the choice of the *initial predicates.* As the basic initial predicates in our experiments, we considered using automata representing the set of bad or initial configurations. We used them alone or together with automata corresponding to the domains or ranges of the transducers encoding the particular transitions in the systems being examined. The scenarios that lead to the best results are listed in Table 2. The *heuristic optimisation* of the refinements described in Section 4.3.3, had a very significant positive impact in the case of checking individual liveness in the Bakery example. In the other cases, the effect was neutral or negative.

The times presented in Tables 1 and 2 are in seconds and were obtained on a computer with a 1.7 GHz Intel Pentium 4 processor. They do not include the time needed for reading the input model. Taking into account that the tool used was an early prototype written in YAP Prolog using the FSA library [53][12], the results are very positive. For example, the Uppsala Regular Model Checker [5] took from about 8 to 11 seconds when applied to a comparable encoding of the Burns, Szymanski, and Dijkstra examples (and the situation did not change much with [42]). Finally, Tables 1 and 2 also show that apart from cases where the approaches based on languages of words/traces up to a bounded length and the ones based on intersections with predicate languages are roughly equal, there are really cases where either the former or the latter approach is faster. This experimentally justifies our interest in both of the techniques.

[12] Prolog was chosen as a rapid, but still relatively efficient, prototyping environment.

5 Regular Tree Model Checking

As was already noted, regular tree model checking is a generalisation of regular (word) model checking 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 typically encoded as a linear tree transducer τ[13].

To illustrate the use of tree automata and transducers, let us consider a simple example—namely, a generalisation of the simple token passing protocol from Section 3.1 to trees. We suppose having a tree-shaped network of processes of an arbitrary size. Initially, a token is situated in one of the leaf nodes. Then, it is to be sent up to the root. We would like to check that the token does not disappear nor duplicate.

The initial configurations of the simple tree token passing protocol is encoded by the tree automaton $Init = (Q_{Init}, \Sigma, F_{Init}, \delta_{Init})$ where $\Sigma = \Sigma_0 \cup \Sigma_2$ with $\Sigma_0 = \{T_0, N_0\}$ and $\Sigma_2 = \{T, N\}$[14], $Q_{Init} = \{p_0, p_1\}$, $F_{Init} = \{p_1\}$, and δ_{Init} contains the following transitions:

$$\begin{array}{ll} N_0 \to p_0 & T_0 \to p_1 \\ N(p_0, p_0) \to p_0 & \\ N(p_1, p_0) \to p_1 & N(p_0, p_1) \to p_1 \end{array}$$

The one-step transition relation is represented by the tree transducer τ with Σ used as the input/output alphabet, $Q_\tau = \{q_0, q_1, q_2\}$, $F_\tau = \{q_2\}$, and the following transitions[15]:

$$\begin{array}{ll} N_0/N_0 \to q_0 & T_0/N_0 \to q_1 \\ N/N(q_0, q_0) \to q_0 & T/N(q_0, q_0) \to q_1 \\ N/T(q_1, q_0) \to q_2 & N/T(q_0, q_1) \to q_2 \\ N/N(q_2, q_0) \to q_2 & N/N(q_0, q_2) \to q_2 \end{array}$$

Finally, the set of bad configurations is encoded by the tree automaton Bad with Σ as its ranked alphabet, $Q_{Bad} = \{r_0, r_1, r_2\}$, $F_{Bad} = \{r_0, r_2\}$, and the following transitions:

$$\begin{array}{ll} N_0 \to r_0 & T_0 \to r_1 \\ N(r_0, r_0) \to r_0 & T(r_0, r_0) \to r_1 \\ N(r_1, r_0) \to r_1 & N(r_0, r_1) \to r_1 \\ T(r_1, r_0) \to r_2 & T(r_0, r_1) \to r_2 \\ N \text{ or } T(r_1, r_1) \to r_2 & N \text{ or } T(r_0 \text{ or } r_1, r_2) \to r_2 \\ N \text{ or } T(r_2, r_0 \text{ or } r_1) \to r_2 & N \text{ or } T(r_2, r_2) \to r_2 \end{array}$$

Similarly to the case of classical word regular model checking, the basic *safety verification problem of regular tree model checking* consists in deciding whether $\varrho^*_\tau(L(Init)) \cap L(Bad) = \emptyset$ holds. Of course, this problem is again in general undecidable, an iterative computation of $\varrho^*_\tau(L(Init))$ does not necessarily terminate, and so some acceleration techniques are needed to make it terminate as often as possible. Generalisations of the various acceleration schemes from regular model checking into trees have been considered—see, e.g., [50,51, 18,19,6,7]. Below, we concentrate on a generalisation of using abstraction for this purpose.

[13] Like in RMC, another possibility is to use several transducers and/or special-purpose operations on tree automata.

[14] To respect the formal definition of a ranked alphabet, we distinguish leaf and non-leaf nodes with/without a token.

[15] We are dealing with a relabelling transducer and for a better readability, we write its transitions in the form $f/g(q_1, q_2) \to q$ where f is an input symbol and g an output symbol.

5.1 Abstract Regular Tree Model Checking

A generalisation of abstract regular model checking to trees was originally considered in [13]. The proposed approach allows one to deal with structure-preserving as well as non-preserving tree transducers. Similarly to the word case, the introduction of an *automated abstraction* with a counterexample-guided refinement brings in not only an efficient acceleration technique, but also a quite efficient way for fighting the state explosion problem in the number of tree automata states.

In particular, two abstractions for tree automata have been proposed. Similarly to abstract word regular model checking, 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 tree predicate languages* as an analogy to the word automata predicate abstraction.

The proposed technique was successfully applied for verification of parametric tree networks of processes [13] and also programs with complex dynamic data structures [14], which we will discuss in detail in Section 6.

5.1.1 The Framework of ARTMC

We can formalise the basic framework of abstract regular tree model checking (ARTMC) in a way quite similar to word regular model checking. We basically phrase all the needed concepts not for classical finite automata, but for finite tree automata.

Note that in the following as in ARMC, in order to shorten the descriptions, we *identify a tree transducer and the relation it represents* and write $\tau(L)$ instead of $\varrho_\tau(L)$. Let $\iota \subseteq T_\Sigma \times T_\Sigma$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations $\tau^0 = \iota$, $\tau^{i+1} = \tau \circ \tau^i$ and $\tau^* = \cup_{i=0}^{\infty} \tau^i$. Below, we suppose $\iota \subseteq \tau$ meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

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 an 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 $\iota \subseteq \tau$, 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 safety verification problem checking whether $\tau^*(L(Init)) \cap L(Bad) = \emptyset$ has a positive answer. Otherwise, the answer is not necessarily negative since during the computation of the set $\tau_\alpha^*(L(Init))$, the abstraction α may introduce extra behaviours leading to $L(Bad)$. Let us examine this case. Assume $\tau_\alpha^*(Init) \cap L(Bad) \neq \emptyset$, meaning that there is a symbolic path $Init$, $\tau_\alpha(Init)$, $\tau_\alpha^2(Init)$, $\ldots$, $\tau_\alpha^n(Init)$ 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: (1) $X_0 = L(Init) \cap (\tau^{-1})^n(X_n) \neq \emptyset$, which means that the safety verification problem has a *negative answer*, or (2) there is a $k \geq 0$ such that $X_k = \emptyset$, and this means that the considered symbolic path is actually a *spurious counterexample* due to the fact that α is too coarse. In this last situation, we need to refine α and iterate the procedure. Therefore, ARTMC is based on abstraction schemas allowing to compute families of (automatically) refinable abstractions.

5.1.2 Abstractions over Tree Automata

Below, we discuss two tree automata abstraction schemas based on tree automata state equivalences. First, tree automata states are split into several equivalence classes by an equivalence relation. Then, states from each equivalence class are collapsed 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.

Abstraction Based on Tree Languages of Finite Height. We now present the possibility of defining automata state equivalence schemas which are based on comparing automata states wrt. a certain bounded part of their languages. The abstraction schema $\mathbb{H}_n$ is a generalisation of the schema based on languages of words up to a certain length (cf. Section 4.4). The $\mathbb{H}_n$ 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.

One can implement the abstraction schema $\mathbb{H}_n$ much like minimisation of tree automata [22], by simply stopping the main minimisation loop after n iterations.

Abstraction Based on Predicate Tree Languages. We next introduce a predicate-based abstraction schema $\mathbb{P}_\mathcal{P}$ that is inspired by the predicate-based abstraction on words discussed in Section 4.3.

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*. It can be refined by adding new predicates into $\mathcal{P}$ in a way analogous to the word case (cf. Section 4.3). Thus, we can show that a spurious counterexample can be eliminated 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 counterexample (recall that $X_k = \emptyset$) as presented in Section 5.1. Similar optimisations like those in Section 4.3.3 apply here too.

Above, we discussed the $\mathbb{P}_\mathcal{P}$ abstraction schema inspired by the predicate-based abstraction from word abstract regular model checking. In particular, it is inspired by the *backward* predicate-based abstraction schema $\mathbb{B}_\mathcal{P}$. Interestingly, as illustrated in Figure 9, it is impossible to obtain a tree analogy with the *forward* predicate-based abstraction schema $\mathbb{F}_\mathcal{P}$ of word abstract regular model checking. The tree analogy would be to label a state with a predicate state if the languages of their contexts—i.e., trees where we substitute Σ^* for the language of the node being labelled/used for labelling—have a non-empty intersection. However, in this case, the refinement schema we use in all our predicate-based abstractions does not work. For instance, consider tree automata whose fractions are shown in Figure 9. In the figure, L_i below a state means that the language of that state is L_i, and we assume that $L_i \cap L_j = \emptyset$ for any $i \neq j$. Suppose we start with no predicates and want to refine the abstraction so that the refined abstraction of M does not intersect the language of *Bad*. To ensure this using our refinement schema, we should take the context languages of the states of *Bad* as the new predicates. Assume we

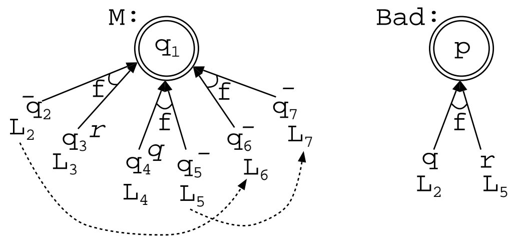


Fig. 9. A problem with the forward tree predicate abstraction

do so and try to abstract M. In Figure 9, the upper index of the states of M shows by which states of Bad they are labelled when abstracting M (the minus sign means that no state appears in the label). The abstraction would now collapse states q_2, q_5, q_6, and q_7. Consequently, as the arrows show, the resulting automaton could accept trees $f(t_1, t_2)$ for $t_1 \in L_2$ and $t_2 \in L_5$ belonging to $L(Bad)$ despite $L(M)$ does not contain such trees. So, the refinement has not excluded trees from $L(Bad)$ from the language of the abstraction of M and there is no way how to refine the abstraction further using our refinement schema.

6 Verification of Programs with Pointers

This section discusses our fully-automated method for analysing various important properties of programs manipulating *complex dynamic linked data structures* that was first published in [14]. We consider non-recursive, sequential C programs manipulating dynamic linked data structures with possibly several next pointer selectors, storing data from finite domains. The properties to be checked include basic consistency of pointer manipulations (i.e., checking that there are no *null pointer assignments*, no use of *undefined pointers*, no references to *deleted elements*). Moreover, further undesirable behaviours of the programs at hand (such as, for instance, disobedience of certain *shape invariants*, e.g., due to introducing undesirable sharing, cycles, etc.) may be detected via *testers written in C* and attached to the verified programs. Then, verification of such properties reduces to reachability of a designated error location.

Our verification method uses the approach of *abstract regular tree model checking.* In order to be able to apply it on programs manipulating dynamic linked data structures whose configurations (naturally viewed as the so called *shape graphs*) need not be tree-like, we proceed as follows. We use trees to encode the *tree skeletons* of shape graphs. The edges of a shape graph that are not directly encoded in the tree skeleton are represented by *routing expressions* over the tree skeleton—i.e., regular expressions over directions in a tree (such as left up, right down, etc.) and the kind of nodes that can be visited on the way. The routing expressions are referred to from the tree skeletons. Both the tree skeletons and the routing expressions are automatically discovered by our method. The idea of using routing expressions is inspired by PALE [40] and graph types [38].

We implemented our method in a prototype tool built on top of the Mona tree libraries [37]. We have tested it on a number of non-trivial procedures manipulating singly-linked lists (SLL), doubly-linked lists (DLL), trees (including the Deutsch-Schorr-Waite tree traversal), lists of lists, and also trees with linked leaves. All the procedures were automatically verified for absence of null pointer dereferences, absence of manipulation with undefined pointers, and absence of dereferencing of deleted objects. Additionally, further shape properties (such as absence of sharing, acyclicity, preservation of input elements, etc.) were also verified for some of the procedures.

6.1 Related Approaches

The area of research on automated verification of programs manipulating dynamic linked data structures is very active. Various approaches to verification of such programs differing in their principles, degree of automation, generality, and scalability have emerged. They are based, e.g., on monadic second order logic [40], 3-valued predicate logic with transitive closure [47], separation logic [45,31,60,20][16], or automata [25,12,14]. Among all of these approaches, the method presented here is one of the most general and fully automated at the same time.

The closest approach to what we present here is the one of PALE that also uses tree automata (derived from WSkS formulae) as well as the idea of a tree skeleton and routing expressions. However, first, the encoding of PALE is different in that the routing expressions must deterministically choose their target, and also, for a given memory node, selector, and program line, the expression is fixed and cannot dynamically change during the run of the analysed program. Further, program statements are modelled as transformers on the level of WSkS formulae, not as transducers on the level of tree automata. Finally, the approach of PALE is not fully automatic as the user has to manually provide loop invariants and all needed routing expressions, which are automatically synthesised in our approach.

6.2 The Considered Programs

We consider standard, non-recursive, sequential C programs manipulating dynamic linked data structures with possibly several next pointer selectors. We do not consider pointer arithmetics, and we suppose all non-pointer data to be abstracted to a finite domain by some of the existing techniques before our method is applied. The abstract syntax of the considered programs is given in Figure 10(a), where Lab is a finite set of program labels (one for each control location), $\mathcal{V}$ is a finite set of

[16] We briefly comment on these approaches in Section 6.8 too.

$l, l_1, l_2 \in Lab, \quad x, y, z \in \mathcal{V}, \quad d \in \mathcal{D},$
$\quad next \in \mathcal{S}$
$Program := \{l : Stmnt;\}^*$
$Stmnt := IfStmnt \mid Update \mid Asgn \mid Goto$
$IfStmnt :=$ `if` $(Cond)$ `then goto` l_1;
$\quad$ `else goto` l_2;
$Cond := x == y \mid x ==$ `NULL` $\mid$
$\quad x$`->data` $== d$
$Update := x =$ `malloc()` $\mid$ `free`(x)
$Asgn := x = y \mid x =$ `NULL` $\mid x = y$`->`$next$
$\quad x$`->`$next = y \mid x$`->data` $= d$
$Goto :=$ `goto` l

```
// doubly-linked lists
typedef struct {
    DLL *next, *prev;
} DLL;

DLL *DLL_reverse(DLL *x) {
    DLL *y,*z;
    if (x==NULL) return x;
    z = NULL;
    y = x->next;
    while (y!=NULL) {
        x->next = z;
        x->prev = y;
        z = x; x = y;
        y = x->next;
    }
    return x;
}
```

```
// A DLL shape tester example
x = aDLLHead;
while (x != NULL && random())
    x = x->next;
if (x != NULL && x->next->prev != x)
    error();
```

Fig. 10. (a) Abstract syntax of the considered programs, (b) a running example: reversion of DLLs, (c) a shape tester example

pointer variables, $\mathcal{D}$ is a finite set of data values, and $\mathcal{S}$ is a finite set of selectors. We suppose other commonly used statements (such as `while` loops or nested dereferences) to be encoded by the listed statements. An example of a typical program that our method can handle is the reversion of doubly-linked lists (DLLs) shown in Figure 10(b), which we also use as our running example.

Memory Configurations. Memory configurations of the considered programs with a finite set of pointer variables $\mathcal{V}$, a finite set of selectors $\mathcal{S} = \{1, ..., k\}$, and a finite domain $\mathcal{D}$ of data stored in dynamically allocated memory cells can be described as shape graphs of the following form. A *shape graph* is a tuple $SG = (N, S, V, D)$ where N is a finite set of memory nodes, $N \cap \{\bot, \top\} = \emptyset$ (we use $\bot$ to represent null, and $\top$ to represent an undefined pointer value), $N_{\bot,\top} = N \cup \{\bot, \top\}$, $S : N \times \mathcal{S} \to N_{\bot,\top}$ is a successor function, $V : \mathcal{V} \to N_{\bot,\top}$ is a mapping that defines where the pointer variables are currently pointing to, and $D : N \to \mathcal{D}$ defines what data are stored in the particular memory nodes.

6.3 The Considered Properties

First of all, the properties we intend to check include *basic consistency of pointer manipulations*, i.e., absence of null and undefined pointer dereferences and references to already deleted nodes. Further, we would like to check various *shape invariance properties* such as absence of sharing, acyclicity, or, e.g., the fact that if `x->next == y` (and `y` is not null) in a DLL, then also `y->prev == x`, etc. To define such properties, we use the so called *shape testers* written in the C language. They can be seen as instrumentation code trying to detect violations of the memory shape properties at selected control locations of the original program.

For defining testers, we slightly extend the C language by allowing next pointers to be followed backwards and by non-deterministic branching. The testers become a part of the code being verified. An error is announced when a line denoted by an error label is reached. This way, we can check a whole range of properties, including acyclicity, absence of sharing, and other shape invariants such as the relation of next and previous pointers in DLLs—cf. Fig 10(c). Shape testers can be directly written by the user, or they can be generated from a more declarative specification based, e.g., on the specialised logic proposed in [14].

In theory, bad shapes may be described directly using a tree automata memory encoding. The problem is to not miss any of their possible encodings since—as we will see—the memory encoding that we are going to use is not canonical. This problem does not arise when using shape testers as in their case, only reachability of a certain line is tested and the choice of a suitable encoding is subject to the automatic abstraction refinement.

6.4 The Verification Problem

Above, we have explained that for checking preservation of shape invariants, we use shape testers, for which we need to check unreachability of their designated error location. Moreover, we model all program statements such that if some basic memory consistency error (like a null pointer assignment) happens, the control is automatically transferred to a unique error control location. Thus, we are in general interested in *checking unreachability of certain error control locations* in a program.

6.5 Encoding the Programs in Tree Automata

In this section, we describe our encoding of memory configurations of the considered programs into trees and tree automata and our encoding of program statements using tree transducers and specialised automata operations.

6.5.1 Encoding of Sets of Memory Configurations

As was described in Section 6.2, memory configurations of the considered programs with a finite set of pointer variables $\mathcal{V}$, a finite set of selectors $\mathcal{S} = \{1, \ldots, k\}$, and a finite domain $\mathcal{D}$ of data stored in dynamically allocated memory cells can be described as *shape graphs* $SG = (N, S, V, D)$. We suppose $\top \in \mathcal{D}$—the data value $\top$ is used to denote "zombies" of deleted nodes, which we keep and detect all erroneous attempts to access them.

To be able to describe the way we encode sets of shape graphs using tree automata, we first need a few auxiliary notions. First, to allow for dealing with more general shape graphs than tree-like, we do not simply identify the next pointers with the branches of the trees accepted by tree automata. Instead, we use the tree structure just as a backbone over which links between the allocated nodes are expressed using the so called *routing expressions*, which are regular expressions over directions in a tree (like move up, move left down, etc.) and over the nodes that can be seen on the way. From nodes of the trees described by tree automata, we refer to the routing expressions via their symbolic names called *pointer descriptors*—we suppose dealing with a finite set of pointer descriptors $\mathcal{R}$. Moreover, we couple each pointer descriptor with a unique *marker* from a set $\mathcal{M}$ (and so $|\mathcal{R}| = |\mathcal{M}|$). The routing expressions may identify several target nodes for a single source memory node and a single selector. Markers associated with the target nodes can then be used to decrease the nondeterminism of the description (only nodes marked with the right marker are considered as the target).

Let us now fix the sets $\mathcal{V}$, $\mathcal{S}$, $\mathcal{D}$, $\mathcal{R}$, and $\mathcal{M}$. We use a *ranked alphabet* $\Sigma = \Sigma_2 \cup \Sigma_1 \cup \Sigma_0$ consisting of symbols of ranks $k = |\mathcal{S}|$, 1, and 0. Symbols of rank k represent allocated memory nodes or nodes that were allocated, but later they have been deleted (freed). Allocated nodes may be pointed to by pointer variables whereas deleted nodes are not pointed to by any variables since we make all variables pointing to such nodes undefined. Allocated as well as deleted nodes may be marked by some markers as targets of some next pointers, they contain some data, and have k next pointers that are either null, undefined (which is the only possibility for deleted nodes), or given by some next pointer descriptor. Thus, $\Sigma_2 = \Sigma_{2,a} \cup \Sigma_{2,d}$ where $\Sigma_{2,a} = 2^{\mathcal{V}} \times 2^{\mathcal{M}} \times \mathcal{D} \times (\mathcal{R} \cup \{\perp, \top\})^k \times \{\texttt{alloc}\}$ and $\Sigma_{2,d} = \{\emptyset\} \times 2^{\mathcal{M}} \times \mathcal{D} \times \{\top\}^k \times \{\texttt{del}\}$. Given an element $n \in \Sigma_2$, we use the notation $n.var$, $n.mark$, $n.data$, and $n.s$ (for $s \in \mathcal{S}$) to refer to the pointer variables, markers, data, and descriptors associated with n, respectively. Σ_1 is used for specifying nodes with undefined and null pointer variables, and so $\Sigma_1 = 2^{\mathcal{V}}$. Finally, in our trees, the leaves are all the same (with no special meaning), and so $\Sigma_0 = \{\bullet\}$.

We can now specify the *tree memory backbones* we use to encode memory configurations as the trees that belong to the language of the tree automaton with the following rules[17]: (1) $\bullet \rightarrow q_i$, (2) $\Sigma_2(q_i/q_m, \ldots, q_i/q_m) \rightarrow q_m$, (3) $\Sigma_1(q_m/q_i) \rightarrow q_n$, and (4) $\Sigma_1(q_n) \rightarrow q_u$. Intuitively, q_i, q_m, q_n, and q_u are automata states where q_i accepts the leaves, q_m accepts the memory nodes, q_n accepts the node encoding null variables, and q_u, which is the accepting state, accepts the node with undefined variables. Note that there is always a single node with undefined variables, a single node with null variables, and then a sub-tree with the memory allocated nodes. Thus, every memory tree t can be written as $t = undef(null(t'))$ for $undef, null \in \Sigma_1$. We say a memory tree $t = undef(null(t'))$ is *well-formed* if the pointer variables are assigned unique meanings, i.e., $undef \cap null = \emptyset \wedge \forall p \in \mathcal{N}lPos(t') : t'(p).var \cap (null \cup undef) = \emptyset \wedge \forall p_1 \neq p_2 \in \mathcal{N}lPos(t') : t'(p_1).var \cap t'(p_2).var = \emptyset$.

We let $\mathcal{S}^{-1} = \{s^{-1} \mid s \in \mathcal{S}\}$ be the set of "inverted selectors" allowing one to follow the links in a shape graph in the reverse order. A *routing expression* is then formally defined as a regular expression on pairs $s.p \in (\mathcal{S} \cup \mathcal{S}^{-1}).\Sigma_2$. Intuitively, each pair used as a basic building block of a routing expression describes one step over the tree memory backbone: The step follows a certain branch up or down after which a certain kind of node should be encountered (most often, we will use the node components of routing expressions to check whether a certain marker is set in the target node).

A *tree memory encoding* is a tuple (t, μ) where t is a tree memory backbone and μ a mapping from the set of pointer descriptors $\mathcal{R}$ to routing expressions over the set of selectors $\mathcal{S}$ and the memory node alphabet Σ_2 of t. An example of a tree memory encoding for a *doubly-linked list* (DLL) is shown in Fig. 11.

Let (t, μ), $t = undef(null(t'))$, be a tree memory encoding with a set of selectors $\mathcal{S}$ and a memory node alphabet Σ_2. We call $\pi = p_1 s_1 \ldots p_l s_l p_{l+1} \in \Sigma_2.((\mathcal{S} \cup \mathcal{S}^{-1}).\Sigma_2)^l$ a *path in* t of length $l \geq 1$ iff $p_1 \in \mathcal{P}os(t')$ and $\forall i \in \{1, \ldots, l\} : (s_i \in \mathcal{S} \wedge p_i.s_i = p_{i+1} \wedge p_{i+1} \in \mathcal{P}os(t')) \vee (s_i \in \mathcal{S}^{-1} \wedge p_{i+1}.s_i = p_i)$. For $p, p' \in \mathcal{N}lPos(t')$ and a selector $s \in \mathcal{S}$, we write $p \xrightarrow{s} p'$ iff (1) $t'(p).s \in \mathcal{R}$, (2) there is a path $p_1 s_1 \ldots p_l s_l p_{l+1}$ in t for some $l \geq 0$ such that $p = p_1$, $p_{l+1} = p'$, and (3) $s_1 t'(p_2) \ldots t'(p_l) s_l t'(p_{l+1}) \in \mu(t'(p).s)$.

The *set of shape graphs represented by a tree memory encoding* (t, μ) with $t = undef(null(t'))$ is denoted by $[\![(t, \mu)]\!]$ and given as all the shape graphs $SG = (N, S, V, D)$ for which there is a bijection $\beta : \mathcal{P}os(t') \rightarrow N$ such that:

1. $\forall p, p' \in \mathcal{N}lPos(t')\ \forall s \in \mathcal{S} : (t'(p).s \notin \{\perp, \top\} \wedge p \xrightarrow{s} p') \Leftrightarrow S(\beta(p), s) = \beta(p')$, i.e., the links between memory nodes are respected.

[17] We use a set of symbols instead of a single input symbol in a transition rule to concisely describe a set of rules using any of the symbols in the set. Similarly, a use of q_1/q_2 instead of a single state means that one can take either q_1 or q_2, and if there is a k-tuple of states, one considers all possible combinations of the states.

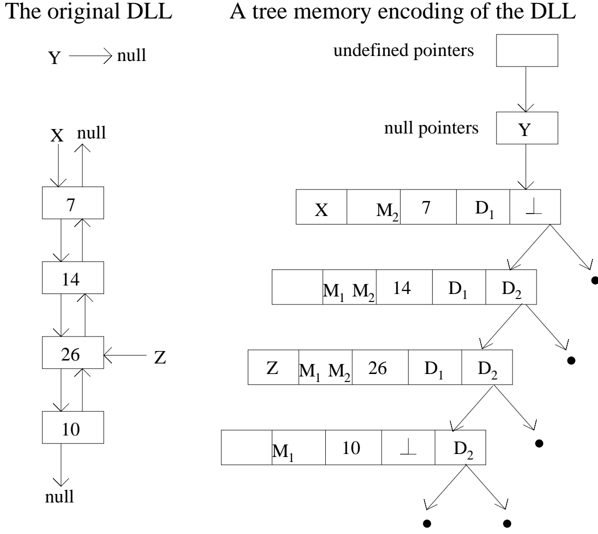


Fig. 11. A tree memory encoding for a doubly linked list (DLL). The descriptors are mapped as follows: $D_1 : 1.M_1$ and $D_2 : \overline{1}.M_2$. Only allocated nodes are present, hence the `alloc` flag is omitted.

2. $\forall p \in \mathcal{N}lPos(t')\ \forall s \in \mathcal{S}\ \forall x \in \{\perp, \top\} : t'(p).s = x \Leftrightarrow S(\beta(p), s) = x$, i.e., null and undefined successors are respected.
3. $\forall v \in \mathcal{V}\ \forall p \in \mathcal{P}os(t') : v \in t'(p).var \Leftrightarrow V(v) = \beta(p)$, i.e., assignment of memory nodes to variables is respected.
4. $\forall v \in \mathcal{V} : (v \in null \Leftrightarrow V(v) = \perp) \wedge (v \in undef \Leftrightarrow V(v) = \top)$, i.e., assignment of null and undefinedness of variables are respected.
5. $\forall p \in \mathcal{N}lPos(t')\ \forall d \in \mathcal{D} : t'(p).data = d \Leftrightarrow D(\beta(p)) = d$, i.e., data stored in memory nodes is respected.

A *tree automata memory encoding* is a tuple (A, μ) where A is a tree automaton accepting a regular set of tree memory backbones and μ is a mapping as above. Naturally, A represents the set of shape graphs defined by $[\![(A, \mu)]\!] = \bigcup_{t \in L(A)} [\![(t, \mu)]\!]$.

The tree automata memory encoding is clearly *not canonical*, i.e. two tree automata having different languages might represent the same set of shape graphs. Nevertheless, as we show below, program statements can still be encoded faithfully, partly using relabelling tree transducers and partly specialised operations on tree automata. Another important property of the encoding is that given a tree automata memory encoding (A, μ), the set $[\![(A, \mu)]\!]$ can be empty although $L(A)$ is not empty (since the routing expressions can be incompatible with the tree automaton). Of course, if $L(A)$ is empty, then $[\![(A, \mu)]\!]$ is also empty. Therefore, checking emptiness of $[\![(A, \mu)]\!]$ (which is important for applying the ARTMC framework, see Section 6.7) can be done in a sound way by checking emptiness of $L(A)$.

6.5.2 Pointer Descriptors and Routing Expressions

As for the set of *pointer descriptors* $\mathcal{R}$, we restrict ourselves to a unique pointer descriptor for each destructive update `x->`s `= y` that appears in the program. This is because statements of this kind establish new links among the allocated memory nodes and having one descriptor per such a statement appears to be sufficient according to our practical experience. In addition, we might have some further descriptors if they are a part of the specification of the input configurations (see Section 6.7).

Further, in our automata-based framework, we encode *routing expressions* using tree transducers. A transducer representing a routing expression r simply copies the input tree memory backbone on which it is applied up to: (1) looking for a data node n_1 that is labelled with a special token $\blacklozenge \notin \mathcal{V} \cup \mathcal{M} \cup \mathcal{D}$ and (2) moving $\blacklozenge$ to a data node n_2 that is the target of the next pointer described by r and that is also marked with the appropriate marker. As described in the next section, we can then implement program statements that follow the next pointers (e.g., `x = y->`s) by putting the token $\blacklozenge$ to a node pointed to by `y`, applying the transducer implementing the appropriate routing expression, and making `x` point to the node to which $\blacklozenge$ was moved. Due to applying abstraction, the target may not always be unique—in such a case, the transducer implementing the routing expression simply returns a set of trees in which $\blacklozenge$ is put to some target data node such that all possibilities where it can get via the given routing expression are covered.

Note that the use of tree transducers for encoding routing expressions allows us in theory to express more than using just regular expressions. In particular, we can refer to the tree context of the nodes via which the given route is going. In our current implementation, we, however, do not use this fact.

6.5.3 Encoding of Program Statements

We encode the considered pointer-manipulating statements as relabelling tree transducers or sets of such transducers being applied sequentially, in one case combined with an application of an additional specialised operation on the tree automata being handled[18]. When simulating the various program statements, we expect the tree memory encoding to be extended by a new root symbol, corresponding to the *current program line* or to an *error indicator* when an error is found during the analysis. The encoding of the program statements works in such a way that the effect of the statements is simulated on any set of shape graphs represented by a tree automata memory encoding. If a shape graph SG represented by a tree memory encoding is changed by

[18] The primary reason for this is to avoid a need of implementing non-structure preserving transducers on top of the MONA tree automata library [37], which we use to implement our techniques.

a program statement to a shape graph SG', then the encoding of the statement transforms the tree memory encoding such that it represents SG'. This makes sure that although the memory encoding is not canonical, we simulate program statements faithfully.

Non-destructive Updates and Tests. The simplest statement to encode is the `x = NULL` assignment. The transducer implementing it simply goes through the input tree and copies it to the output with the exception that (1) it removes `x` from the labelling of the node in which it currently is, (2) it adds `x` to the labelling of the *null* node, and (3) it changes the current line appropriately. The transducer implementing an assignment `x = y` is similar, it just puts `x` not to the *null* node, but to the node which is currently labelled by `y`.

The transducers encoding conditional statements of the form `if (x == NULL) goto l1; else goto l2;` are very similar to the above—of course, they do not change the node in which `x` is, but only change the current program line to either `l1` or `l2` according to whether or not `x` is in the *null* node. If `x` is in *undef*, an error indication is used instead of `l1` or `l2`. The transducers encoding statements `if (x == y) goto l1; else goto l2;` are similar—they test whether or not `x` and `y` appear in the same node (both being different from *undef*).

The transducer for an `x = y->`s statement is a union of several complementary actions. If `y` is in *null* or *undef*, an error is indicated. If `y` is in a regular data node and its s-th next pointer node contains either $\perp$ or $\top$, the transducer removes `x` from the node it is currently in and puts it into the *null* or *undef* node, respectively. If `y` is in a regular data node n and its s-th next pointer node contains some pointer descriptor $r \in \mathcal{R}$, the ⧫ token is put to n. Then, the routing expression transducer associated with r is applied. Finally, `x` is removed from its current node and put into the node to which ⧫ was moved by the applied routing expression transducer. If the target is marked as deleted, an error is announced.

Destructive Updates. Destructive pointer updates of the form `x->`s `= y` are implemented as follows. If `x` is in *null* or *undef*, an error is announced. If `x` is defined and `y` is in *null* or *undef*, the transducer puts $\perp$ or $\top$ into the s-th next pointer node below `x`, respectively. Otherwise, the transducer puts the pointer descriptor r associated with the particular `x->`s `= y` statement being fired into the s-th next pointer node below `x`, and it marks the node in which `y` is by the marker coupled with r. Then, the routing expression transducer associated with r is updated such that it includes the path from the node of `x` to the node of `y`.

One could think of various strategies how to *extract the path* going from the node of `x` to the node of `y`. We consider a simple strategy, which is, however, successful in many practical examples as our experiments show. We extract the shortest path between `x` and `y` on the tree memory backbone, which consists of going some number of steps upwards to the closest common parent of `x` and `y` and then going some number of steps downwards. The upward or the downward phase can also be skipped when going just down or up, respectively. When extracting this shortest path, we project away all information about nodes we see on the way and about nodes not directly lying on the path. Only the directions (left/right up/down) and the number of steps are preserved.

Note that we, in fact, perform the operation of routing expression extraction on a tree automaton, and we extract all possible paths between where `x` and `y` may currently be. The result is transformed into a transducer $\tau_{\mathtt{xy}}$ that moves the token ⧫ from the position of `x` to the position of `y`, and $\tau_{\mathtt{xy}}$ is then united with the current routing expression transducer associated with the given pointer descriptor r. The extraction of the routing paths is done partly by rewriting the input tree automaton via a special transducer τ_π that in one step identifies all the shortest paths between all `x` and `y` positions and projects away the non-necessary information about the nodes on the way. The transducer τ_π is simple. It just checks that one follows some branch up from `x` and then some branch down to `y` where the up and down sweeps meet in a single node. The transition relation of the resulting transducer is then post-processed by changing the context of the path to an arbitrary one, which is done by directly modifying the structure of the transducer.[19]

Dynamic Allocation and Deallocation. Statements of the form `x = malloc()` are implemented by rewriting the right-most • leaf node to a new data node pointed to by `x`. All the k next pointers are set to $\top$.

To be able to exploit the regularity that is mostly present in algorithms allocating new data structures, which typically add new elements at the end/leaves of the structure, we also explicitly support a statement of the form `x.`s `= malloc()`. We even try to pre-process programs and compact all successive pairs of statements of the form `x = malloc(); y->`s `= x` (provided `x` is not used any further) to `y->`s`= malloc()`. This statement is then implemented by adding the new element directly under the node pointed to by y (provided it is a leaf) and joining it by a simple routing expression of the form "one level down via a certain branch". This typically yields much simpler and more precise routing expressions.

Finally, statements of the form `free(x)` are implemented by transducers that move all variables that are currently in the node pointed to by `x` to the *undef* node (if `x` is in *null* or *undef*, an error is announced). Then, the node is denoted as deleted, but it stays in our tree memory encoding with all its current markers set.

[19] A more precise, but also more costly, approach would be to preserve (some of) the context.

6.6 Input Structures for the Verified Programs

In order to encode the input structures, we can directly use the tree automata memory encoding. Such an encoding can be provided manually or derived automatically from a description of the concerned linked data structure provided, e.g., as a graph type [38]. The main advantage is that the verification process starts with an exact encoding of the set of all possible instances of the considered data structure. Another possibility is to attach a *constructor written in C* before the verified procedure. The verification then starts with the empty shape graph.

6.7 Applying ARTMC

Apparently, we assume ARTMC to be used in its more general form, having the one-step transition relation split into several transducers that are applied in some particular order, together with one special operation on the tree automata used when extracting the routing expressions. We compute an overapproximation of the reachable configurations for each program line in such a way that we start from an initial set of shape graphs represented by a tree automata memory encoding (possibly representing the empty heap when an input constructor is used) and we iterate the abstract fixpoint computation described in Section 5.1 along the control flow graph of the program (using the depth-first strategy). The fixpoint computation stops if the abstraction α that is used is finitary. In such a case, the number of abstracted tree automata that encode sets of memory backbones which can arise in the program being checked is finite. Moreover, the number of the arising routing expressions is also finite since they are extracted from the bounded number of tree automata describing the encountered sets of memory backbones.[20]

During the computation, we check whether a designated error location in the program is reached, a basic pointer exception is detected during simulating the effect of some statement, or whether a fixpoint is attained. In the latter case, the program is found correct. In the former case, we compute backwards along the path in the CFG that is being currently explored to check if the obtained counterexample is spurious as explained in Section 5.1. However, as said in Section 6.5.1, the check for emptiness is not exact and therefore we might conclude that we have obtained a real counterexample although this is not the case. However, such a situation has never happened in any of our experiments[21].

We use a slight refinement of the basic finite-height and predicate abstractions described in Section 5.1. Concretely, we prevent the abstraction from allowing a certain pointer variable to point to several memory nodes at the same time. In particular, this amounts to prohibiting collapsing of states that would create a loop over a node pointed to by some pointer variable.

Apart from the basic abstraction schemas, we support one more abstraction schema called the *neighbour abstraction*. Under this schema, only the tree automata states are collapsed that (1) accept nodes with equal labels and (2) that directly follow each other (i.e., they are neighbours). This strategy is very simple, yet it proved useful in some practical cases.

Finally, we allow the abstraction to be applied either at all program lines or only at the loop closing points. In some cases, the latter approach is more advantageous due to some critical destructive pointer updates are done without being interleaved with abstraction. This way, we may avoid having to remove lots of spurious counterexamples that may otherwise arise when the abstraction is applied while some important shape invariant is temporarily broken.

6.8 Experimental Results

We have implemented the above proposed method in a prototype tool[22] based on the Mona tree automata library [37]. We have performed a set of experiments with singly-linked lists (SLL), doubly-linked lists (DLL), trees, lists of lists, and trees with linked leaves. As one of the most complicated case studies, we have also considered the so called *task-lists*. The task-list structure is showed in Figure 12 and it is inspired by the structures often used in operating systems [9].

All three mentioned types of automata abstraction—the finite-height abstraction (with the initial height being one), predicate abstraction (with no initial predicates), and neighbour abstraction—proved useful in different experiments. All case studies were automatically verified for null/undefined/deleted pointer exceptions. Additionally, some further shape properties (such as absence of sharing, acyclicity, preservation of input elements, etc.) were verified in some case studies too. For a detailed overview of the case studies and verified properties, see [46].

Table 3 contains verification times for our experiments. We give the best result obtained using one of the three mentioned abstraction schemas and say for which schema the result was obtained. The note “restricted” accompanying the abstraction means that the abstraction was applied at the loop points only. The experiments were performed on a 64bit Opteron at 2.8 GHz. The column $|Q|$ gives information about the size (in numbers of states) of the biggest encountered automaton while N_{ref} gives the number of refinements. The column SP provides information whether preservation of some shape properties was verified (together with the default checks

[20] The non-canonicity of our encoding does not prevent the computation from stopping. It may just take longer since several encodings for the same graph could be added.

[21] A precise (but more costly) spuriousness check is to replay the obtained path from the beginning without using abstraction.

[22] `www.fit.vutbr.cz/research/groups/verifit/tools/artmc/`

Fig. 12. The "task-list" data structure

of undesired manipulation of null, undefined, or deleted objects).

Despite the prototype nature of our tool, which can still be optimised in many ways (some of them are mentioned below), the results are quite positive. For example, for checking the Deutsch-Schorr-Waite tree traversal, TVLA (version 2) took 1 minute on the same machine with manually provided instrumentation predicates and predicate transformers. In the case of the trees with linked leaves, we are not aware of any other fully automated tool with which experiments with this structure have been performed.

Recent fully automated tools based on separation logic appear to be more scalable, but the abstraction used in them is much more restricted to a particular shape of data structures (usually lists of lists [9, 60, 20] like in the case of task-lists[23]) and their particular properties. For example, the abstraction of [60] is fine-tuned not to report spurious errors in the considered experiments and the authors themselves make a note on usefulness of automated refinement.

7 Conclusions

We have discussed the approach of abstract regular (tree) model checking as a generic technique for verification of parameterised and infinite-state programs, using finite word and tree automata for finitely representing possibly infinite sets of reachable configurations. As a possible application of ARTMC, we have discussed verification of programs manipulating complex dynamic data structures. Other applications include verification of parameterised networks of processes or systems with unbounded queues, stacks, counters, etc.

An important on-going research on (abstract) regular (tree) model checking includes improvements in the underlying automata technology. This especially concerns the use of non-deterministic word or tree automata. For them to be useful, one needs to be able to perform all the needed operations without determinising the automata. This is problematic in the case of inclusion checking and reduction (minimisation) of the automata. However, recent advances in using antichain-based and/or simulation-based approaches [26, 1] have overcome this obstacle, although further improvements are still possible. Very significant improvements in the performance of ARTMC have already been reported on some case studies [11]. Further, development of abstractions and encodings particularly suitable for various application areas is needed. For instance, an interesting question is how to introduce the principle of separation (frame rule), which underlies the scalability of separation-logic-based approaches, into automata, combining it with automated refinement and higher generality while still retaining full automation [32].

Acknowledgement. This work was supported by the Czech Science Foundation (projects P103/10/0306 and 201/09 /P531), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the internal BUT FIT grant FIT-S-11-1, the Czech-French Barrande project MEB021023, the French ANR RNTL Averiles project and the French ANR-09-SEGI Veridyc project.

References

1. P.A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When Simulation Meets Antichains (on Checking Language Inclusion of NFAs). In *Proc. of TACAS'10*, LNCS 6015. Springer, 2010.
2. P.A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular Model Checking for MSO + LTL. In *Proc. of CAV'04*, LNCS 3114. Springer, 2004.
3. P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular Model Checking for LTL(MSO), *Special Section on Regular Model Checking*, STTT, in this volume, 2010.
4. P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Regular Model Checking Made Simple and Efficient. In *Proc. of CONCUR'02*, LNCS 2421. Springer, 2002.
5. P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Algorithmic Improvements in Regular Model Checking. In *Proc. of CAV'03*, LNCS 2725. Springer, 2003.
6. P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, LNCS 2404. Springer, 2002.
7. 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.

[23] In [31], more complex structures are automatically manipulated, but they are built by the programs in some "nice" way, suitable for the inductive predicates being constructed and used.

Table 3. Results of experiments with analysing programs manipulating dynamic data structures

Example	Time	Abstraction method	$\|Q\|$	N_{ref}	SP
Creation of SLLs	1s	predicates, restricted	25	0	yes
Reversion of SLLs	5s	predicates	52	0	yes
Deletion from DLLs	6s	finite height	100	0	yes
Insertion into DLLs	10s	neighbour, restricted	106	0	yes
Reversion of DLLs	7s	predicates	54	0	yes
Insertsort of DLLs	2s	predicates	51	0	no
Inserting into trees	23s	predicates, restricted	65	0	yes
Depth-first search	11s	predicates	67	1	yes
Linking leaves in trees	40s	predicates	75	2	yes
Inserting into a list of lists	5s	predicates, restricted	55	0	yes
Deutsch-Schorr-Waite tree traversal	47s	predicates	126	0	no
Insertion into task-lists	11m 25s	finite-height, restricted	277	0	yes
Deletion in task-lists	1m 41s	predicates, restricted	420	0	yes

8. S. Bensalem, Y. Lakhnech, and S. Owre. Computing Abstractions of Infinite State Systems Compositionally and Automatically. In *Proc. of CAV'98*, LNCS 1427. Springer, 1998.
9. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proc. of CAV'07*, LNCS 4490. Springer, 2007.
10. B. Boigelot, A. Legay, and P. Wolper. Iterating Transducers in the Large. In *Proc. of CAV'03*, LNCS 2725. Springer, 2003.
11. A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In *Proc. of CIAA'08*, LNCS 5148. Springer, 2008.
12. A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. of TACAS'05*, LNCS 3440. Springer, 2005.
13. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking. In *Proc. of Infinity'05*, ENTCS 149:37–48, 2006.
14. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, LNCS 4134. Springer, 2006.
15. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, LNCS 3114. Springer, 2004.
16. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. of CAV'00*, *LNCS* 1855. Springer, 2000.
17. A. Bouajjani, A. Legay, and P. Wolper. Handling Liveness Properties in (ω-)Regular Model Checking. In *Proc. of Infinity'04*, ENTCS 138:101–115, 2005.
18. A. Bouajjani, T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, LNCS 2404. Springer, 2002.
19. A. Bouajjani, and T. Touili. Widening Techniques for Regular Tree Model Checking, *Special Section on Regular Model Checking*, STTT, in this volume, 2010.
20. C. Calcagno, D. Distefano, P.W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-abduction. In *Proc. of POPL'09*. ACM Press, 2009.
21. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proc. of CAV'00*, LNCS 1855. Springer, 2000.
22. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications, 2005. URL: `http://www.grappa.univ-lille3.fr/tata`.
23. D. Dams, Y. Lakhnech, and M. Steffen. Iterating Transducers. In *Proc. of CAV'01*, LNCS 2102. Springer, 2001.
24. S. Das and D.L. Dill. Counter-Example Based Predicate Discovery in Predicate Abstraction. In *Proc. of FMCAD'02*, 2002.
25. J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, LNCS 3920. Springer, 2006.
26. L. Doyen and J.-F. Raskin. Antichain Algorithms for Finite Automata. In *Proc. of TACAS'10*, LNCS 6015. Springer, 2010.
27. J. Engelfriet. Bottom-up and Top-down Tree Transformations—A Comparison. *Mathematical System Theory*, 9:198–231, 1975.
28. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In *Proc. of CAV'00*, LNCS 1855. Springer, 2000.
29. L. Fribourg and H. Olsen. Reachability Sets of Parametrized Rings as Regular Languages. In *Proc. of Infinity'97*, ENTCS 9, 1997.
30. S. Graf and H. Saïdi. Construction of Abstract State Graphs with PVS. In *Proc. of CAV'97*, LNCS 1254. Springer, 1997.
31. B. Guo, N. Vachharajani, and D.I. August. Shape Analysis with Inductive Recursion Synthesis. In *Proc. of PLDI'07*. ACM Press, 2007.
32. P. Habermehl, L. Holík, A. Rogalewicz, J. Šimáček, and T. Vojnar. A Proposal of a New Automata-based Representation of Heaps, 2010. SVARM'10, work in progress.
33. P. Habermehl and T. Vojnar. Regular Model Checking Using Inference of Regular Languages. In *Proc. of Infinity'04*, ENTCS 138:21–36, 2005.
34. T.A. Henzinger, R. Jhala, R. Majumdar, G. Sutre. Lazy Abstraction. In *Proc. of POPL'02*. ACM Press, 2002.
35. B. Jonsson and M. Nilsson. Transitive Closures of Regular Relations for Verifying Infinite-State Systems. In *Proc. of TACAS'00*, LNCS 1785. Springer, 2000.
36. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, E. Shahar. Symbolic Model Checking with Rich Assertional Languages. In *Proc. of CAV'97*, LNCS 1254. Springer, 1997.
37. N. Klarlund and A. Møller. MONA Version 1.4 User Manual, 2001. BRICS, Department of Computer Science, University of Aarhus, Denmark.

38. N. Klarlund and M.I. Schwartzbach. Graph Types. In *Proc. of POPL'93*. ACM Press, 1993.
39. A. Legay. Extrapolating (Omega-)Regular Model Checking. *Special Section on Regular Model Checking*, STTT, in this volume, 2010.
40. A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001.
41. M. Nilsson. Regular Model Checking. Licentiate Thesis, Uppsala University, Sweden, 2000.
42. M. Nilsson. *Regular Model Checking*. PhD thesis, Uppsala University, 2005.
43. D. Perrin and J.-E. Pin. *Infinite Words: Automata, Semigroups, Logic and Games*. Academic Press, 2003.
44. A. Pnueli and E. Shahar. Liveness and Acceleration in Parameterized Verification. In *Proc. of CAV 2000*, LNCS 1855. Springer, 2000.
45. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.
46. A. Rogalewicz. *Verification of Programs with Complex Data Structures*. PhD thesis, FIT, Brno University of Technology, 2005.
47. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
48. H. Saidi. Model Checking Guided Abstraction and Analysis. In *Proc. of SAS'00*, LNCS 1824. Springer, 2000.
49. V. Schuppan and A. Biere. Liveness Checking as Safety Checking for Infinite State Spaces. In *Proc. of Infinity'05*, 2005.
50. E. Shahar. *Tools and Techniques for Verifying Parameterized Systems*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 2001.
51. E. Shahar, A. Pnueli. Acceleration in Verification of Parameterized Tree Networks. Technical Report MCS02-12, Weizmann Institute of Science, Rehovot, Israel, 2002.
52. T. Touili. Regular Model Checking Using Widening Techniques. *ENTCS*, 50, 2001.
53. G. van Noord. FSA6.2, 2004. URL: `http://odur.let.rug.nl/~vannoord/Fsa/`.
54. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Actively Learning to Verify Safety for FIFO Automata. In *Proc. of FSTTCS'04*, LNCS 3328. Springer, 2004.
55. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to Verify Safety Properties. In *Proc. of ICFEM'04*, LNCS 3308. Springer, 2004.
56. A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using Language Inference to Verify Omega-Regular Properties. In *Proc. of TACAS'05*, LNCS 3440. Springer, 2005.
57. A. Vardhan and M. Viswanathan. Learning to Verify Branching Time Properties. In *Proc. of ASE'05*. IEEE/ACM, 2005.
58. T. Vojnar. *Cut-offs and Automata in Formal Verification of Infinite-State Systems*. Habilitation thesis, FIT, Brno University of Technology, Czech Republic, 2007.
59. P. Wolper and B. Boigelot. Verifying Systems with Infinite but Regular State Spaces. In *Proc. of CAV'98*, LNCS 1427. Springer, 1998.
60. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV'08*, LNCS 5123. Springer, 2008.