CAV 2004 Submission—Category A, regular papers

Abstract Regular Model Checking

Ahmed Bouajjani[1], Peter Habermehl[1], and Tomáš Vojnar[2]

[1] LIAFA, University Paris 7, Case 7014, 2, place Jussieu, F-75251 Paris Cedex 05, France
e-mail:{Ahmed.Bouajjani,Peter.Habermehl}@liafa.jussieu.fr
[2] FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic
e-mail: vojnar@fit.vutbr.cz

Abstract. We propose *abstract regular model checking* as a new generic technique for verification of parametric and infinite-state systems. The technique combines the two approaches of regular model checking and verification by abstraction. We propose a general framework of the method as well as several concrete ways of abstracting automata or transducers, which we use for modelling systems and encoding sets of their configurations as usual in regular model checking. The abstraction is based on collapsing states of automata (or transducers) and its precision is being incrementally adjusted by analysing spurious counterexamples. We illustrate the technique on verification of a wide range of systems including a novel application of automata-based techniques to an example of systems with dynamic linked data structures.

Key words: Infinite-state model checking, regular model checking, verification by abstraction, counterexample-guided abstraction refinement.

Corresponding author: Tomáš Vojnar
Postal address:
FIT, Brno University of Technology
Božetěchova 2, CZ-61266, Brno, Czech Republic
Tel.: +420541141202, Fax: +420541141270
E-mail: vojnar@fit.vutbr.cz

1 Introduction

Model checking is nowadays widely accepted as a powerful technique for the verification of finite-state systems. However, many real-life systems, especially software systems, exhibit various aspects requiring one to reason about infinite-state models (data manipulation, dynamic creation of objects and threads, etc.). Several approaches extending model checking to be able to deal with them have recently been proposed. One of them is *regular model checking* [22, 30, 12]—a generic, automata-based approach allowing for a uniform modelling and verification of various kinds of infinite-state systems such as pushdown systems, (lossy) FIFO-channel systems, systems with counters, parameterised and dynamic networks of processes, etc.

In regular model checking, configurations of systems are encoded as words over a finite alphabet and transitions are modelled as finite state transducers mapping configurations to configurations. Finite automata can then be naturally used to represent and manipulate (potentially infinite) sets of configurations, and reachability analysis can be performed by computing transitive closures of transducers [21, 15, 3, 9] or images of automata by iteration of transducers [12, 28]—depending on whether dealing with *reachability 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.

A 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 current regular model checking techniques. Typically, these techniques try to calculate the *exact* reachability sets (or relations) independently of the property being verified. However, it would often be enough to only compute an overapproximation of the reachability set (or relation) sufficiently precise to verify the given property. Indeed, this is the way large (or infinite) state spaces are being successfully handled outside the domain of regular model checking using the so-called *abstract-check-refine* paradigm [20, 26, 14, 16] implemented, e.g., in tools for software model checking like Slam [6], Magic [13], or Blast [19]. All these tools use the method of *predicate abstraction*, where a finite set of boolean predicates is used to abstract a concrete system C into an abstract one A by considering equivalent the configurations of C that satisfy the same predicates. If a property is verified in A, it is guaranteed to hold in C too. If a counterexample is found in A, one can check if it is also a counterexample for C. If not, this *spurious* counterexample can be used to *refine* the abstraction such that the new abstract system A' no longer admits the spurious counterexample. In this way, one can construct finer and finer abstractions until a sufficient precision is achieved and the property is verified, or a real counterexample is found.

In this work, we propose a new approach to regular model checking based on the abstract-check-refine paradigm. 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 A to an automaton A' from some finite domain such that A' recognises a superset of the language of A. For the case that the computed overapproximation is too coarse and a spurious counterexample is detected, we give effective principles allowing the abstraction to be refined such that the new abstract computation does not encounter the same counterexample.

We propose two techniques for abstracting automata. 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. However, notice that 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 A to "satisfy" a predicate language L_P if the intersection of L_P with the language $L(A,q)$ 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* 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 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.

We have implemented the different abstraction and refinement schemas in a prototype tool and tested them on a number of examples of various types of systems including parametric networks of processes, pushdown systems, counter automata, systems with queues, and—for the first time in the context of regular model checking—an example of dynamic linked data structures. The experiments show that our techniques are quite powerful in all the considered cases and that they are complementary—different techniques turn out to be the most successful in different scenarios. The results are very promising and compare favourably with other existing tools.

Related Work. In addition to what we have already mentioned above, many other results have been obtained for symbolic model checking of different kinds of infinite state systems, such as pushdown systems [10, 17], systems with counters [5, 18] or queues [4, 2, 11]. These works do not consider abstraction. For parameterised networks of processes, several methods using abstractions have been proposed [7, 24]. Contrary to our approach, these methods do not provide the possibility of refinement of the abstraction. Moreover, they are specialised for parameterised networks whereas our technique is generic.

2 Finite Automata and Transducers

We first provide a simple example that we use to demonstrate the way systems are modelled in (abstract) regular model checking and later to illustrate the verification techniques we propose. Then, we formalise the basic notions of finite automata and transducers and briefly comment on their use in verification by regular model checking.

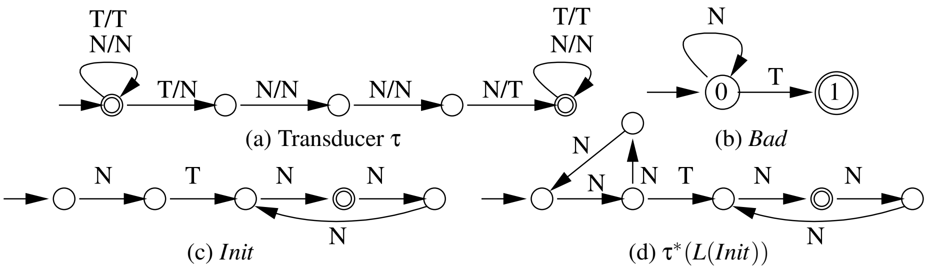


Fig. 1. A transducer modelling a simple token passing, initial, bad and reachable configurations

As a running example, we consider the transducer τ in Fig. 1. It models a simple parameterised system implementing a token passing algorithm. The system consists of a parametric number of processes arranged in an array. Each process either has (T) or does not have (N) the token. Each process can pass the token to its third right neighbour. The transducer includes the identity relation too. In the initial configurations described by the automaton *Init*, 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*.

Formally, a *finite automaton* is a 5-tuple $M = (Q,\Sigma,\delta,q_0,F)$ where Q is a finite set of states, Σ a finite alphabet, $\delta \subseteq Q\times\Sigma\times Q$ a set of transitions, $q_0 \in Q$ an initial state and $F \subseteq Q$ a set of final states. The transition relation $\rightarrow \subseteq Q\times\Sigma^*\times Q$ of M is defined as the smallest relation satisfying: (1) $\forall q \in Q : q \xrightarrow{\varepsilon} q$, (2) if $(q,a,q') \in \delta$, then $q \xrightarrow{a} q'$ and (3) if $q \xrightarrow{w} q'$ and $q' \xrightarrow{a} q''$, then $q \xrightarrow{wa} q''$. Given a finite automaton M and an equivalence relation $\sim$ on its states, $M/\sim$ denotes the *quotient automaton* defined in the usual way.

The language recognised by M from a state $q \in Q$ is defined by $L(M,q) = \{w \mid \exists q' \in F : q \xrightarrow{w} q'\}$. The language $L(M)$ is equal to $L(M,q_0)$. A set $L \subseteq \Sigma^*$ is regular iff there exists a finite automaton M such that $L = L(M)$. We also define the backward language $\overleftarrow{L}(M,q) = \{w \mid q_0 \xrightarrow{w} q\}$ 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.

Let Σ be a finite alphabet and $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$. A *finite transducer* over Σ is a 5-tuple $\tau = (Q,\Sigma_\varepsilon\times\Sigma_\varepsilon,\delta,q_0,F)$ where Q is a finite set of states, $\delta \subseteq Q\times\Sigma_\varepsilon\times\Sigma_\varepsilon\times Q$ a set of transitions, $q_0 \in Q$ an initial state and $F \subseteq Q$ a set of final states. A finite transducer is called a *length-preserving transducer* if its transitions do not contain ε. The transition relation $\rightarrow \subseteq Q\times\Sigma^*\times\Sigma^*\times Q$ is defined as the smallest relation satisfying: (1) $q \xrightarrow{\varepsilon,\varepsilon} q$ for every $q \in Q$, (2) if $(q,a,b,q') \in \delta$, then $q \xrightarrow{a,b} q'$ and (3) if $q \xrightarrow{w,u} q'$ and $q' \xrightarrow{a,b} q''$, then $q \xrightarrow{wa,ub} q''$. Then, by abuse of notation, we identify a transducer τ with the relation $\{(w,u) \mid \exists q' \in F : q_0 \xrightarrow{w,u} q'\}$. For a set $L \subseteq \Sigma^*$ and a relation $R \subseteq \Sigma^*\times\Sigma^*$, we denote $R(L)$ the set $\{w \in \Sigma^* \mid \exists w' \in L : (w',w) \in R\}$. Let $id \subseteq \Sigma^*\times\Sigma^*$ be the identity relation and $\circ$ the composition of relations. We define recursively the relations $\tau^0 = id$, $\tau^{i+1} = \tau\circ\tau^i$ and $\tau^* = \cup_{i=0}^{\infty}\tau^i$. Below, we suppose $id \subseteq \tau$ meaning that $\tau^i \subseteq \tau^{i+1}$ for all $i \geq 0$.

The properties we want to check are primarily reachability properties. Given a system with a transition relation modelled as a transducer τ, a regular set of initial configurations given by an automaton *Init*, and a set of "bad" configurations given by an automaton *Bad*, we want to check $\tau^*(L(Init)) \cap L(Bad) = \emptyset$. We transform more complicated properties into reachability by composing the appropriate property automaton with the system being checked. In this way, even liveness properties may be handled if the transition relation is instrumented to allow for loop detection as briefly mentioned later. For our running example, $\tau^*(L(Init))$ is shown in Fig. 1(d), and the property of interest clearly holds. Notice, however, that in general, $\tau^*(L(Init))$ is neither guaranteed to be regular nor computable. In the following, the verification task is to find a regular overapproximation $L \supseteq \tau^*(L(Init))$ such that $L \cap L(Bad) = \emptyset$.

3 Abstract Regular Model Checking

In this section, we describe the general approach of abstract regular model checking and propose a common 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. We concentrate on the use of abstract regular model checking for dealing with reachability sets. However, the techniques we propose may be applied to dealing with reachability relations too—though in the context of length-preserving transducers only. (Indeed, length-preserving transducers over an alphabet Σ can be seen as finite-state automata over $\Sigma \times \Sigma$.) We illustrate the applicability of the method to dealing with reachability relations by one of the experiments presented in Section 6.

3.1 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, i.e. for some $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$, $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ 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, we introduce the *abstract transition function* τ_α for a transition relation expressed as a transducer τ over Σ and an automata abstraction function 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 $\tau(L(M))$. Now, we may iteratively compute the sequence $(\tau_\alpha^i(M))_{i\geq 0}$. Since we suppose $id \subseteq \tau$, it is clear that if α is finitary, there exists $k \geq 0$ such that $\tau_\alpha^{k+1}(M) = \tau_\alpha^k(M)$. 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 $\tau^*(L(M))$.

3.2 Refining Automata Abstractions

We call an automata abstraction function α' a *refinement* of α iff $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. Moreover, we call α' a *true refinement* iff it yields a smaller overapproximation in at least one case—formally, iff $\exists M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subset L(\alpha(M))$.

A need to refine α arises when a situation depicted in Fig. 2 happens. Suppose we are checking whether no configuration from the set described by some automaton *Bad* is

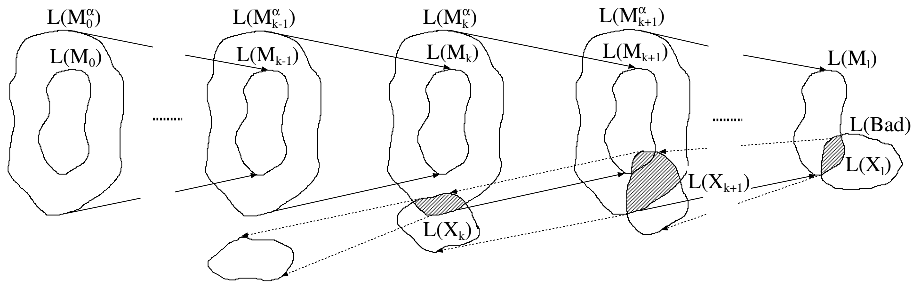


Fig. 2. A spurious counterexample in an abstract regular fixpoint computation

reachable. We start with the initial configuration set 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 disjunct 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, the 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.

3.3 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 $\mathbb{E}$ such that $\forall M \in \mathbb{M}_\Sigma : \alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$. 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 basic principles: (1) comparing states wrt. the intersections of their forward/backward languages with some *predicate languages* (represented by the appropriate *predicate automata*) and (2) comparing states wrt. their forward/backward behaviours up to a certain *bounded length*.

4 Automata State Equivalences Based on Predicate Languages

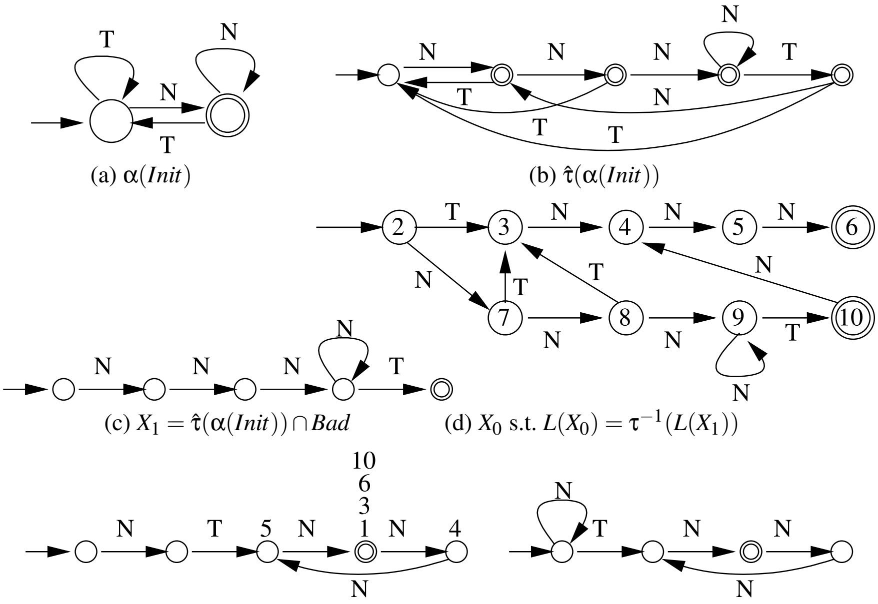
The two automata state equivalence schemas we introduce in this section—$\mathbb{F}_{\mathcal{P}}$ based on forward languages of states and $\mathbb{B}_{\mathcal{P}}$ based on backward languages—are both defined wrt. a finite set of *predicate automata* $\mathcal{P}$. They compare two states of a given automaton according to the intersections of their forward/backward languages with the languages of the predicates. Below, we first introduce the basic principles of the schemas and then add some implementation and optimisation notes.

4.1 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. 1, if we take as $\mathcal{P}$ the automata of the languages of the states of *Bad*, we obtain the automaton in Fig. 3(a) as the abstraction of *Init* from Fig. 1(c). This is because all states of *Init* except the final one become equivalent. Then, the intersection of $\tau(\alpha(\mathit{Init}))$ with the bad configurations—shown in Fig. 3(c)—is not empty, and we have to refine the abstraction.



(a) $\alpha(Init)$ (b) $\hat{\tau}(\alpha(Init))$

(c) $X_1 = \hat{\tau}(\alpha(Init)) \cap Bad$ (d) X_0 s.t. $L(X_0) = \tau^{-1}(L(X_1))$

(e) *Init* labelled with states of X_0 and *Bad* (f) The final result

Fig. 3. An example using abstraction based on predicate languages

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. 2. Theorem 1 proved in Appendix A.1 shows that this prevents abstractions of languages disjunct with $L(X_k)$, such as—but not only—$L(M_k)$, from intersecting with $L(X_k)$. Thus, as already mentioned, a repetition of the same faulty computation is excluded, and we are prevented from reaching the set of bad configurations unless there is another reason for this than overapproximating by subsets of $L(X_k)$.

Theorem 1. *Let us have any two finite automata* $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ *and* $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ *and a finite set of predicate automata* $\mathcal{P}$ *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.*

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. 3(d). Fig. 3(e) then indicates for each state q of *Init*, the predicates corresponding to the states of *Bad* and X_0 whose languages have a non-empty intersection with the language of q. The first two states of *Init* are equivalent and are collapsed to obtain the automaton from Fig. 3(f), which is a fixpoint showing that the property is verified. Notice that it is an overapproximation of the set of reachable configurations from Fig. 1(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 in Sect. 4.3, 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 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 in Sect. 4.3.

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

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

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. 2. As stated in Theorem 2, the effect is the same as for $\mathbb{F}_{\mathcal{P}}$.

Theorem 2. *Let us have any two finite automata $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ and $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ and a finite set of predicate automata $\mathcal{P}$ 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.*

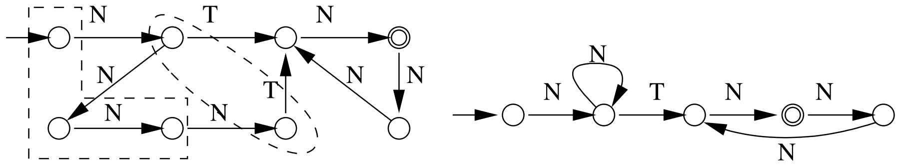
4.3 Implementing and Optimising Collapsing Based on $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$

The abstraction of an automaton M wrt. the automata state equivalence schema $\mathbb{F}_{\mathcal{P}}$ may 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 disjunct.) 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 may 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. As stated in Theorem 3 (proved in Appendix A.1), the effect of such a refinement corresponds to the weaker way of refining automata abstraction functions described in Section 3.2. 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.

Theorem 3. *Let M and X be any finite automata over Σ and $Y = (Q_Y, \Sigma, \delta_Y, q_0^Y, F_Y)$ the important tail/head part of X wrt. $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ and M. If $\mathcal{P}'$ is such that $\forall q_Y \in Q_Y\ \exists P \in \mathcal{P}' : L(Y, q_Y)/\overleftarrow{L}(Y, q_Y) = L(P)$ and $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{F}_{\mathcal{P}'}/\mathbb{B}_{\mathcal{P}'}}(M)) \cap L(X) = \emptyset$.*



(a) $\hat{\tau}(Init)$ (b) The final result

Fig. 4. An example using abstraction based on languages of words up to length n (for $n = 2$)

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 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].

5 Automata State Equivalences Based on Finite-Length Languages

We now present the possibility of defining automata state equivalence schemas 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 a 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, as we work with minimal deterministic automata, we may achieve the weaker type of refinement described in Section 3.2. Such an effect is achieved when n is increased to be equal or bigger than the number of states in M_k from Fig. 2 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. 4, we apply $\mathbb{F}_n^L$ to the example from Fig. 1. We choose $n = 2$. In this case, the abstraction of the *Init* automaton is *Init* itself. Fig. 4(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. 4(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 its fraction (e.g., one half), increase n by the number of states in X_k (or its fraction), or increase n just by one.

In such cases, 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, we may also think of *backward languages of words* of a limited length and forward or backward *languages of traces* with a limited length. The automata equivalence schemas $\mathbb{B}_n^L$, $\mathbb{F}_n^T$, and $\mathbb{B}_n^T$ 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. 4(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 the schemas based on traces are quite successful in practice.

6 Experiments

We have implemented the ideas proposed in the paper in a prototype tool written in YAP Prolog using the FSA library [29]. 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.

6.1 The Types of Systems Verified

Parameterised networks of processes We considered several slightly idealised *mutual exclusion algorithms* for an arbitrary number of processes (namely the Bakery, Burns, Dijkstra, and Szymanski algorithms in versions similar to [23]). In most of these systems, particular processes are finite-state. We encode global configurations of such systems 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 point of time) and the current configuration being further modified. 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 [17]. 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 case, general (non-length-preserving) transducers were used to encode transitions of the systems.

Petri Nets, 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/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. (Using this encoding, a marking of a net with places p and q, two tokens in p, and four in q would be encoded as $q|q|pq|pq$.) In some other examples of systems with counters (such as the Bakery algorithm for two processes and unboundend counters), we also successfully experimented with a binary encoding of the counters like in NDDs [30].

Dynamic Linked Data Structures We considered verification of a *procedure for reversing lists* shown in Fig. 5. As according to our knowledge, it is for the first time that regular model checking has been applied to such a task, let us now spend a bit more time with this experiment.

```
x := 0;
while (list → next) {
    y := list → next;
    list → next := x;
    x := list;  list := y;
}
list → next := x;
```

Fig. 5. Reversing a linear list

When abstracting the memory manipulated by the procedure, we focus on the cases where in the first n memory cells (we take the biggest n possible) 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 represent configurations of the procedure as words over the following alphabet: list items are represented by symbols $\underline{i}$, left/right pointers by $</>$, pointer variables are represented by their names (*list* is shortened to l), and $\underline{o}$ is used to represent the memory outside the list. Moreover, we use symbols $\underline{iv}$ (resp. $\underline{ov}$) to denote that v points to i (resp. outside the list). We use $|$ to separate the ascending and descending lists. Pointer variables pointing to null are not present in the configuration representations. A typical abstraction of the memory may then look 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. Provided such an abstraction is used for the memory contents (prefixed with the current control line), it is not difficult to associate transducers to 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 does not fit the above described form, we abstract it to a single word with the "don't know" meaning. However, when we start from a configuration like $1\ \underline{il} > \underline{i} > \underline{i}\ \underline{o}$ or $1\ \underline{i} < \underline{i} < \underline{il}\ \underline{o}$, the verification shows that such a situation does not happen. Via a symmetry argument exploiting the fact that the procedure never refers to concrete addresses, the results of the verification may then easily be generalised 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 is a reversion of the input one (modulo the fact that we consider a finite number of different items). To speed up the computation, the reachability relation was restricted to the initial configurations, i.e. to $id_{Init} \circ \tau^*$.

6.2 A Summary of the Results

Our method has been quite successful in all the described experiments. The best results (corresponding to a certain automata state equivalence used, its initialisation, and the way of refinement) for the particular examples obtained from our prototype tool were mostly under 1 sec. on an Intel Pentium 4 processor at 1.7 GHz. The only exceptions were checking individual liveness in the Bakery example where we needed about 9 sec., the Readers/Writers example where we needed about 6 sec., and the example of reversing lists where on the level of working with the reachability relation, we needed about 22 sec. (A more detailed description of the results may be found in Appendix A.2.) Taking into account that the tool used was an early prototype written in Prolog, the results are very encouraging. For example, on the same computer, the Uppsala Regular Model Checker [3] in version 0.10 took from about 8 to 11 seconds when applied to the Burns, Szymanski, and Dijkstra examples over whose comparable encoding we needed in the most successful scenarios from 0.06 to 0.73 sec.

The results we obtained from our experiments also showed 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 clearly faster. The latter approach is faster, e.g., in the Dijkstra and Readers/Writers examples whereas the former, e.g., in the cases of reversing lists or checking the individual liveness in the Bakery example. This experimentally justifies our interest in both of the techniques.

Let us note that for some of the classes of systems we considered, there exist various special purpose verification approaches, and the appropriate verification problems are sometimes even decidable (as, e.g., for push-down systems [10, 17, 27] or lossy channel systems [4, 2, 1]). However, we wanted to show that our approach is generic and can be uniformly applied to all these systems. Moreover, in the future, with a new version of our tool, we would like to compare the performance of abstract regular model checking with the specialised approaches on large systems. We believe that while we can hardly outperform these algorithms in general, in some cases of systems with complex state spaces, our approach could turn out to be quite competitive due to not working with the exact representation of the state spaces, but their potentially much simpler approximations in which many details not important for the property being checked are ignored.

7 Conclusions

We have proposed a new technique for verification of parameterised and infinite-state systems. The technique called *abstract regular model checking* combines the approach of regular model checking with the abstract-check-refine paradigm. We have described the general framework of the method as well as several concrete strategies for abstracting automata or transducers, which are used in the framework for representing reachability sets and relations, by collapsing their states. As we have illustrated on a number of experiments we did, including a novel application of automata-based techniques to verification of systems with dynamic linked structures, the technique is very broadly applicable. Moreover, compared to the other techniques of regular model checking, it is also quite efficient due to working with approximations of reachability sets or relations with precision found iteratively by eliminating spurious counterexamples and sufficient just to verify the property of interest.

In the future, we plan to implement the techniques we proposed in the paper in a more efficient way and test them on larger examples. An interesting theoretical question is then whether some guarantees of termination can be obtained at least for some classes of systems. Further, the way of dealing with liveness properties within abstract regular model checking can be put on a more systematic basis allowing for a better automation. Similarly, we would like to extend our results on verifying systems with linear (linearisable) dynamic linked data structures and see how automata-based verification techniques compare with static analysis techniques like shape analysis [25]. Finally, we believe that there is a wide space for generalising the method to working with non-regular reachability sets and/or systems with non-linear (tree-like or in general graph-like) structure of states. The latter generalisation is primarily motivated by our interest in verifying multithreaded programs with recursive procedures and/or programs with dynamically allocated memory structures that cannot be easily linearised.

Acknowledgement. We would like to thank Andreas Podelski for fruitful discussions.

References

1. P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic Verification of Lossy Channel Systems: Application to the Bounded Retransmission Protocol. In *Proc. of TACAS*, volume 1579 of *LNCS*. Springer, 1999.
2. P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly Analysis of Systems with Unbounded, Lossy Fifo Channels. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998.
3. P.A. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Algorithmic improvements in regular model checking. In *Proc. of CAV'03*, volume 2725 of *LNCS*. Springer, 2003.

4. B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Proc. of CAV'96*, volume 1102 of *LNCS*. Springer, 1996.
5. B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. of CAV'94*, volume 818 of *LNCS*, pages 55–67. Springer, 1994.
6. T. Ball and S. K. Rajamani. The SLAM toolkit. In *Proc. CAV'01*, LNCS. Springer, 2001.
7. K. Baukus, S. Bensalem, Y. Lakhnech, and K. Stahl. Abstracting WS1S systems to verify parameterized networks. In *Proc. of TACAS*, volume 1785 of *LNCS*. Springer, 2000.
8. S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proc. of CAV'98*, LNCS. Springer, 1998.
9. B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *Proc. CAV'03*, volume 2725 of *LNCS*. Springer, 2003.
10. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *Proc. of CONCUR'97*, LNCS. Springer, 1997.
11. A. Bouajjani and P. Habermehl. Symbolic Reachability Analysis of Fifo-Channel Systems with Nonregular Sets of Configurations. *Theoretical Computer Science*, 221(1-2):211–250, 1999.
12. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.
13. S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent c programs. *Formal Methods in System Design*, 2004. To appear.
14. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV'00*, volume 1855 of *LNCS*. Springer, 2000.
15. D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In *Proc. CAV'01*, volume 2102 of *LNCS*. Springer, 2001.
16. S. Das and D.L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*, 2002.
17. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.
18. A. Finkel and J. Leroux. How to compose presburger-accelerations: Applications to broadcast protocols. In *Proc. of FST&TCS'02*, volume 2556 of *LNCS*. Springer, 2002.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proc. of 10th SPIN Workshop*, volume 2648 of *LNCS*. Springer, 2003.
20. T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL'02*. ACM Press, 2002.
21. B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. of TACAS'00*, volume 1785 of *LNCS*. Springer, 2000.
22. Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256(1–2), 2001.
23. M. Nilsson. Regular Model Checking. Licentiate Thesis, Uppsala University, Sweden, 2000.
24. A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.
25. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3), 2002.
26. H. Saïdi. Model checking guided abstraction and analysis. In *Proc. of SAS'00*, volume 1824 of *LNCS*. Springer, 2000.
27. Stefan Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, Technische Universität München, 2002.
28. T. Touili. Widening techniques for regular model checking. *ENTCS*, 50, 2001.
29. G. van Noord. FSA6.2, 2004. `http://odur.let.rug.nl/~vannoord/Fsa/`.
30. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. of CAV'98*, volume 1427, 1998.

A Appendix

A.1 Proofs of the Theorems

Theorem 1. *Let us have any two finite automata $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ and $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ and a finite set of predicate automata $\mathcal{P}$ 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 for accepting a word from $L(\alpha_{\mathbb{F}_\mathcal{P}}(M)) \cap L(X)$ in M is i, $i > 0$, 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 accepted (possibly with jumps) just before the jump from q_1 to q_2 (cf. Fig. 6). 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 accepting 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. $\square$

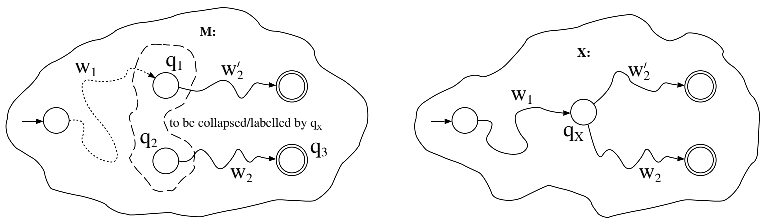


Fig. 6. An illustration of the proof of Theorem 1

Theorem 2. *Let us have any two finite automata $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ and $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ and a finite set of predicate automata $\mathcal{P}$ 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. (Sketch) 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. $\square$

Theorem 3. *Let M and X be any finite automata over Σ and $Y = (Q_Y, \Sigma, \delta_Y, q_0^Y, F_Y)$ the important tail/head part of X wrt. $\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$ and M. If $\mathcal{P}'$ is such that $\forall q_Y \in Q_Y\ \exists P \in \mathcal{P}' : L(Y, q_Y)/\overleftarrow{L}(Y, q_Y) = L(P)$ and $L(M) \cap L(X) = \emptyset$, $L(\alpha_{\mathbb{F}_{\mathcal{P}'}/\mathbb{B}_{\mathcal{P}'}}(M)) \cap L(X) = \emptyset$.*

Proof. 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. □

A.2 Some Concrete Experimental 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.[3] We first say whether it was in a forward or backward computation (i.e. starting from the initial configurations or the "bad" configurations), 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 quite high, and we tried to iteratively find a value of n for which it was the best.

Table 1. Some results of experimenting with abstract regular model checking

Experiment	$\mathbb{F}_n^L/\mathbb{F}_n^T/\mathbb{B}_n^L/\mathbb{B}_n^T$	$T_{best}^{\mathbb{F}_n^L/\mathbb{F}_n^T/\mathbb{B}_n^L/\mathbb{B}_n^T}$	$\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}$	$T_{best}^{\mathbb{F}_{\mathcal{P}}/\mathbb{B}_{\mathcal{P}}}$
Bakery	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert/2$	0.02	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$	0.02
Bakery/comm. liv.	Fw, $\mathbb{F}_n^T$, $\lvert Q_{Bad}\rvert$	0.14	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad\lvert Grd]$	0.13
Bakery/ind. liv.	Fw, $\mathbb{F}_n^T$, 1	8.66	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$, Key St.	19.41
Bakery – counters	Bw, $\mathbb{B}_n^L$, $\lvert Q_{Bad}\rvert$	0.08	Bw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\lvert Grd]$	0.09
ABP	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.32	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init\lvert Grd]$	0.68
Burns	Fw, $\mathbb{B}_n^T$, 1	0.31	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.06
Dijkstra	Fw, $\mathbb{F}_n^T$, 1	1.75	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad]$	0.73
PDS	Bw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert/2$	0.02	Bw, $\mathbb{F}_{\mathcal{P}}$, $[Bad]$	0.02
Petri net/Read. Wr.	Fw, $\mathbb{B}_n^T$, special n	21.07	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\lvert Grd]$	5.86
Faulty PN/Rd. Wr.	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Bad}\rvert$	0.73	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init\lvert Grd]$	0.81
Szymanski	Fw, $\mathbb{B}_n^T$, 1	0.25	Fw, $\mathbb{F}_{\mathcal{P}}$, $[Init\lvert Grd]$	0.55
Rev. Lists	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Init}\rvert/2+\lvert Q_{X_k}\rvert/2$	0.61	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Bad\lvert Grd\lvert Act]$	1.29
Rev. Lists/Transd.	Fw, $\mathbb{F}_n^L$, $\lvert Q_{Init}\rvert/2$	21.79	Fw, $\mathbb{B}_{\mathcal{P}}$, $[Init\lvert Grd\lvert Act]$	42.60

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

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

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 1. The *heuristic optimisation* of the refinements described in Section 4.3, 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 Table 1 are in seconds and were obtained on a computer based on an Intel Pentium 4 processor at 1.7 GHz. 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 Prolog, the results seem to be quite encouraging. For example, on the same computer, the Uppsala Regular Model Checker [3] in version 0.10[4] took from about 8 to 11 seconds when applied to a comparable encoding of the Burns, Szymanski, and Dijkstra examples. Finally, Table 1 also shows 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.

[4] The authors are aware of the fact that this tool as well as other tools for parametric verification are a subject of an intensive development, and new versions may be soon available.