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

Regular Model Checking

An Introduction

Parosh Aziz Abdulla

Department of Information Technology
Uppsala University

Received: xx/xx/xx / Revised version: yy/yy/yy

Abstract. *Regular Model Checking* has been studied extensively during recent years as a framework for algorithmic verification of systems with infinite state spaces. We describe the main concepts of the framework, and some of its applications.

1 Introduction

This paper introduces the main concepts of *Regular Model Checking*, a framework that has been used in recent years for algorithmic verification of various classes of systems with infinite state spaces.

Model checking [34,47,35] is one of the most important approaches to program verification. Model checking has achieved spectacular success in the context of *finite-state systems*, where the behavior can be captured by a finite graph. One important factor in this success has been the use of *Binary Decision Diagrams (BDDs)* as an efficient data structure for symbolic representations of large state spaces [29]; and (more recently) the integration of propositional satisfiability solvers (*SAT-solves*) in model checking engines [33,3].

While hardware circuits can be naturally modeled as finite-state systems, there are several aspects in the behaviors of software systems that give rise to infinite state spaces. In fact, there are at least two sources of "infiniteness". First, programs operate usually on *unbounded data domains*, such as unbounded counters, stacks, queues, and clocks. Second, programs may have unbounded control structures. One example of the latter is multi-threaded programs that may spawn unbounded numbers of threads. Another example is *parameterized systems* that contain unbounded numbers of (often identical) components. For instance, a mutual exclusion protocol should work correctly regardless of the number of participating processes, a cache coherence protocol should work correctly regardless of the number of caches, and a security protocol should work correctly regardless of the number of principals. In such cases, we would like to perform *parameterized verification* in which the correctness property is parameterized (and universally quantified) by the number of components inside the system. These applications have led to a large amount of research, directed towards developing model checking algorithms for different classes of infinite-state systems such as push-down systems [20,31,30], timed systems [12], processes communicating through buffers [9], parameterized systems [5], and many other models.

One direction of research has been to design general frameworks for infinite-state model checking that can be instantiated to wide classes of systems. An example of such a framework is that of *well quasi-ordered programs* that was first proposed in [6] (see [2] for a recent survey). The framework has been applied for the verification of Petri nets, lossy channel systems, timed systems, cache coherence protocols, etc. This paper describes *Regular Model Checking (RMC)* which has also been an important framework for infinite-state model checking. In RMC, sets of states are represented by finite-state automata, and transition relations is represented by finite-state transducers, typically over finite or infinite words or tree structures. The framework allows, for instance, to handle models whose configurations can be represented as finite words or trees of arbitrary length over a finite alphabet. This includes parameterized systems consisting of an arbitrary number of homogeneous finite-state processes connected in a linear, ring-formed, or tree-formed topology, and systems that operate on queues, stacks, integers, and other linear (or tree-like) data structures.

Regular model checking was first advocated by Kesten et al. [44] and by Boigelot and Wolper [49], as a uniform framework for analyzing several classes of parame-

Send offprint requests to:

terized and infinite-state systems. The idea is that regular sets will provide an efficient representation of infinite state spaces, and play a role similar to that played by BDDs for symbolic model checking of finite-state systems. An advantage is that one can exploit automata-theoretic algorithms for manipulating regular sets, e.g., algorithms for minimizing and for checking universality and language inclusion for finite automata. Such algorithms have already been successfully implemented, e.g., in the Mona [45] system; and their development is also currently an active area of research [38, 7].

A generic task in symbolic model checking is to compute the set of reachable states (characterize the states that are reachable from the initial state), in order to verify safety properties; and to compute reachability relations (characterize the relation containing pairs of states (s_1, s_2) such that s_2 is reachable from s_1), in order to verify liveness properties. For finite-state systems this is typically done by state-space exploration for which termination is guaranteed. For infinite-state systems this procedure terminates only if there is a bound on the distance (in number of transitions) from the initial configurations to any reachable configuration. An infinite-state system does not have such a bound, and any nontrivial model checking problem is in general undecidable. RMC is a model checking technique, and hence its aim is to verify system properties algorithmically (automatically). As the problem is undecidable, there is no hope to achieve that goal in general. To circumvent this problem, existing approaches adopt either *incomplete methods* or *approximate methods*.

Incomplete methods are not guaranteed to terminate. Naturally, such a method will be not be useful unless it terminates sufficiently often on practical examples. In order to achieve termination, several works adopt *acceleration* operators, the purpose of which is to compute (in one computation step) the effect of arbitrarily long sequences of transitions [1, 14]. In general, the effect of acceleration is not computable. However, computability have been obtained for certain classes [43]. Analogous techniques for computing accelerations have successfully been developed for several classes of parameterized and infinite-state systems, e.g., systems with unbounded FIFO channels [15, 16, 21, 4], systems with stacks [20, 32, 40], and systems with counters [19, 13].

Approximate methods compute an over-approximation of the original transition relation, and then perform verification on the approximated transition system. A safety property that holds for the over-approximation holds also for the original system. Over-approximations are computed either by applying *abstraction functions* [25, 37], or by applying *widening techniques* [46, 27]. Typically, widening is achieved by first generating increasing sequences of approximations of the set of reachable states, then detecting an "increment" in the manner in which the set of detected states grows, and finally extrapolating by allowing an arbitrary repetition of the detected increment.

Outline In the next section, we present the main concepts in the basic framework of RMC. In Section 3, we describe the techniques used for designing verification algorithms. We show some extensions of the basic model in Section 4. Finally, in Section 5, we give an overview of the papers included in this issue of the journal.

2 Framework

In this section, we introduce the basic framework of RMC. To do that we introduce how programs are modeled, give some examples of systems, and then state the relevant verification problems.

Model A model checking algorithms usually operates on *transition systems*, each consisting of

- a set of *configurations* (or states), some of which are *initial*, and
- a *transition relation*, which is a binary relation on the set of configurations.

The configurations represent possible "snapshots" of the system state, and the transition relation describes how these can evolve over time. Most work on model checking assumes that the set of configurations is finite, but significant effort is underway to develop model checking techniques for transition systems with infinite sets of configurations. RMC is one such a technique. In its simplest form, the RMC framework represents a transition system as follows.

- A *configuration* (state) of the system is a word over a finite alphabet Σ.
- The set of *initial configurations* is a regular set over Σ.
- The *transition relation* is a regular and length-preserving relation on Σ^*. It is represented by a *finite-state transducer* over $(\Sigma \times \Sigma)$, which accepts all words $(a_1, a_1') \cdots (a_n, a_n')$ such that $(a_1 \cdots a_n\ ,\ a_1' \cdots a_n')$ is in the transition relation.

Formally, let Σ be a finite alphabet of symbols. A deterministic finite-state *transducer* T over Σ is a tuple (Q, q_0, t, F) where Q is the set of states, q_0 is the initial state, $t : (Q \times (\Sigma \times \Sigma)) \mapsto Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states. We use $q_1 \xrightarrow{(a,b)} q_2$ to denote that $t(q_1, (a, b)) = q_2$, and use $L(T)$ to denote the language of T.

The transducer T induces a regular relation R on words over Σ. More precisely, for words $x = a_1 \cdots a_n$ and $y = b_1 \cdots b_n$ in Σ^*, we have $(x, y) \in R$ if $(a_1, b_1) \cdots (a_n, b_n) \in L(T)$. The idea is that R is used to represent the transition relation on the configurations of the system (each of which is a word in Σ). Sometimes, we identify the relation R with the transducer T A generic task in RMC is to compute a representation for the transitive closure of the transducer relation, i.e., to construct a new transducer T^+ where $T^+ = \cup_{i>0} T^i$. The transducer T^+ can then be used for computing the set of reachable states (when verifying safety properties), or to find loops (when verifying livens properties). Due to undecidability, the transitive closure cannot be computed in general. Therefore, some acceleration, abstraction, or widening techniques are needed to compute a representation of T^+ by other means.

Examples There are several classes of systems that can be modeled in the RMC framework. Below, we give some examples. For instance, in RMC we can model parameterized systems with linear or ring-formed topologies (where each component if finite-state). A configuration of the system is represented by a word over a finite alphabet, where each member of the alphabet represents a state of a component. In this manner, each position of the word represents the state of the component at that position. As an example, we consider a simple token passing protocol. The system consists of an arbitrary (but finite) number of components organized in a linear fashion. In each step, the process currently having the token passes it to the right. A configuration of the system is a word over the alphabet $\{t, n\}$, where t represents that the process has the token, and n represents not having it. For instance, the word $nntnn$ represents a configuration of a system with five processes where the third process has the token. The set of initial states is given by the regular expression tn^* (Figure 1). Notice that the set of initial configurations is infinite. The transition relation is represented by the transducer in Figure 1. For instance, the transducer accepts the word $(n,n)(n,n)(t,n)(n,t)(n,n)$, representing the pair ($nntnn$, $nnntn$) of configurations where the token is passed from the third to the fourth process.

As a second example, we consider a system consisting of a finite-state process operating on one unbounded FIFO channel. Let Q be the set of control states of the process, and let M be the (finite) set of messages which can reside inside the channel. A configuration of the system is a word over the alphabet $Q \cup M \cup \{e\}$, where the *padding symbol* e represents an empty position in the channel. For instance the word $q_1 e m_3 m_1 e e$ corresponds to a configuration where the process is in state q_1 and the channel (of length four) contains the messages m_3 and m_1 in this order. The set of configurations of the system can thus be described by the regular expression $Qe^*M^*e^*$. By allowing arbitrarily many padding symbols e, one can model channels of arbitrary but bounded length. As an example, the action where the process sends the message m to the channel and changes state from q_1 to q_2 is modeled by the transducer in Figure 2. In the figure, "M" is used to denote any message in M.

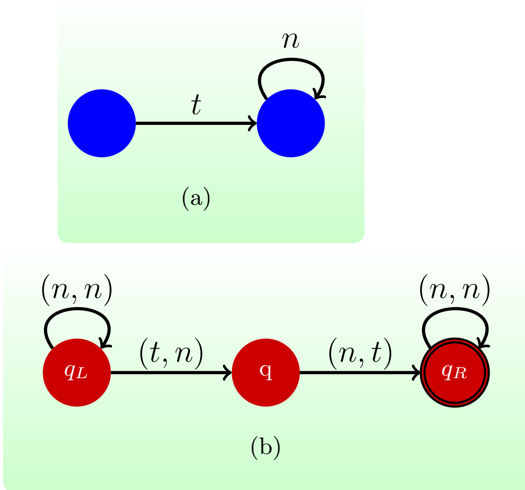

Fig. 1. The token passing protocol. (a) The set of initial states in the token passing protocol. (b) The transducer describing the transition relation.

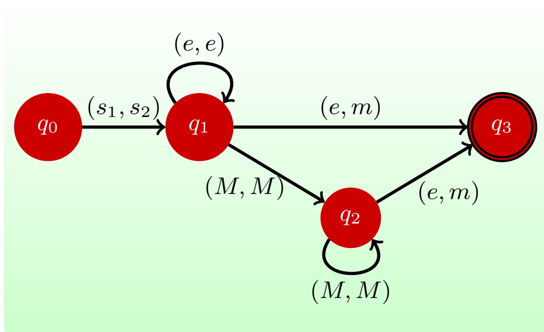

Fig. 2. Sending a message to a queue.

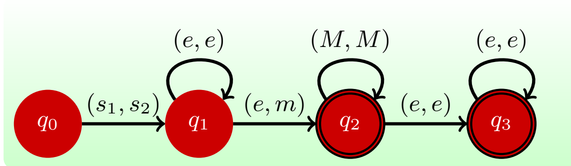

Fig. 3. Pushing a symbol to a stack.

A system consisting of a finite-state process operating on a queue can be modeled in a similar manner. Figure 3 shows the operation of pushing a symbol m to the stack.

Verification Problems Two types of verification problems are usually considered in RMC.

The first problem is verification of *safety properties*. A safety property is of form "bad things do not happen during system execution". A safety property can be verified by solving a *reachability* problem. Formulated in the RMC framework, the corresponding problem is the following: given a regular set of initial configurations I, a regular set of *bad configurations* B and a transition relation specified by a transducer T, does there exist a path from I to B through the transition relation T? This amounts to checking whether $(I \circ T^*) \cap B = \emptyset$. The problem can be solved by computing the set $Inv = I \circ T^*$ and checking whether it intersects B.

The second problem is verification of *liveness properties.* A liveness property is of form "a good thing happens during system execution". Often, liveness properties are verified using fairness requirements on the model, which can state that certain actions must infinitely often be either disabled or executed. Since, by the restriction to length-preserving transducers, any infinite system execution can only visit a finite set of configurations, the verification of a liveness property can be reduced to a *repeated reachability* problem. The repeated reachability problem asks, given a set of initial configurations I, a set of *accepting configurations* F and a transition relation T, whether there exist an infinite computation from I through T that visits F infinitely often? By letting F be the configurations where the fairness requirement is satisfied, and by excluding states where the "good thing" happens from T, the liveness property is satisfied if and only if the repeated reachability problem is answered negatively.

Since the transition relation is length-preserving, and hence each execution can visit only a finite set of configurations, the repeated reachability problem can be solved by checking whether there exists a reachable loop containing some configuration from F. This can be checked by computing $(Inv \cap F)^2 \cap Id$ and checking whether this relation intersects T^+. Here Id is the identity relation on the set of configurations, and $Inv = I \circ T^*$ as before.

3 Algorithms

In Section 2, we stated a verification problem as that of computing a representation of $I \circ T^*$ (or T^+) for some transition relation T and some set of configurations I. In some cases we also have a set of *bad* configurations B and we want to check whether $I \circ T^* \cap B \neq \emptyset$.

Given a transducer T, our goal then is to construct a new transducer that recognizes the relation T^+. As a running illustration, we will consider the problem of computing the transitive closure T^+ for the transducer in Figure 1. A first attempt is to compute T^n, is to take the composition of T with itself n times for $n = 1, 2, 3, \cdots$. For example, T^3 is the transition relation where the token gets passed three positions to the right (its transducer is given in Figure 4). A transducer for T^+ is one where the token gets passed an arbitrary number of times, given in Figure 5. Obviously, the transducer T^+ cannot be constructed naively by simply computing the approximations T^n for $n = 1, 2, 3, \cdots$, since this will not converge. Therefore, different approaches have been proposed for computing T^+. Below, we give overviews of some of them.

Acceleration A solution is proposed in [11], where we derive T^+ in a number of steps as follows. First, starting from T, we can in a straight-forward way construct a transducer for T^+ whose states, called *columns*, are sequences of states in Q, where runs of transitions between columns of length i accept pairs of words in R^i. More precisely, define the *column transducer* for T as the tuple $T^+ = \left(Q^+, q_0^+, \Longrightarrow, F^+\right)$ where

- Q^+ is the set of non-empty sequences of states of T,
- q_0^+ is the set of non-empty sequences of the initial state of T,
- $\Longrightarrow$: $(Q^+ \times (\Sigma \times \Sigma)) \mapsto 2^{Q^+}$ is defined as follows: for any columns $q_1 q_2 \cdots q_m$ and $r_1 r_2 \cdots r_m$, and pair (a, a'), we have $q_1 q_2 \cdots q_m \overset{(a,a')}{\Longrightarrow} r_1 r_2 \cdots r_m$ iff there are $a_0, a_1, \ldots, a_m$ with $a = a_0$ and $a' = a_m$ such that $q_i \overset{(a_{i-1},a_i)}{\longrightarrow} r_i$ for $1 \leq i \leq m$,
- F^+ is the set of non-empty sequences of accepting states of T.

Note that although T is deterministic, T^+ needs not be. It is easy to see that T^+ accepts exactly the relation T^+: runs of transitions from q_0^i to columns in F^i accept transductions in R^i. The problem is that the column transducer has infinitely many states. In order to increase the chances for termination, we present a procedure for incrementally generating a transducer which accepts the same relation as T^+. The procedure starts from T; by successively adding transitions of T^+ we compute a sequence of successively larger (in terms of sets of accepted pairs of words) transducers, all of which underapproximate T^+. Each new approximation is generated through performing a basic step. The step constructs transitions by combining already constructed transitions. In order to that, the algorithm identifies pairs of transitions (of the automaton) and combines them in the following way. When we have a transition from x to x' on (a, b), and a transition from y to y' on (b, c) we add the transition xy to $x'y'$ on (a, c).

Furthermore, we perform *quotienting* based on an equivalence relation $\simeq$ that we define on the set Q^+ of columns of T^+. During the procedure, we will all the time merge columns of T^+ that are equivalent wrt. $\simeq$; thus hopefully arrive at a finite-state result. We define $\simeq$ as follows. A state in $q \in Q$ is *left-copying* if whenever $q_0 \overset{(a_0,a_0')}{\Longrightarrow} q_1 \overset{(a_1,a_1')}{\Longrightarrow} \cdots \overset{(a_{n-1},a_{n-1}')}{\Longrightarrow} q_n$ with $q_n = q$, then $a_i = a_i'$ for all $i \in \{0, 1, \ldots, n-1\}$. A *right-copying* state is defined in a similar manner. In other words, prefixes of left-copying states only copy input symbols to output symbols, and similarly for suffixes of right-copying states. In our example, the states q_L and q_R are left- and right-copying, respectively. Two columns are *equivalent* if they can be made equal by ignoring repetitions of identical neighbours which are either left- or right-copying. Formally, the equivalence classes of $\simeq$ will be sets denoted by regular expressions of form $e_1 e_2 \cdots e_n$ where each e_i is one of the following:

1. q_L^+, for some left-copying state q_L,
2. q_R^+, for some right-copying state q_R,
3. q, for some state q which is neither left-copying nor right-copying,

and where two consecutive e_i can be identical only if they are neither left-copying nor right-copying. For a column x, let $[x]_\simeq$ denote the equivalence class for x. We will use X, Y, etc. to denote equivalence classes of columns. Define the operator $\star$ as the natural concatenation operator on equivalence classes:

$$[x]_\simeq \star [y]_\simeq = [x \cdot y]_\simeq$$

where $\cdot$ denotes concatenation of columns. It is easy to check that this operation is well-defined. If equivalence classes are represented by their defining regular expressions, this means that $e_1 \cdots e_n \ \star f_1 \cdots f_m$ is $e_1 \cdots e_n \ f_1 \cdots f_m$, except when e_n and f_1 are both q^+ for some left- or right-copying state q, in which case it is $e_1 \cdots e_n \ f_2 \cdots f_m$. For instance the columns $q_L q_L x q_R$ and $q_L x q_R q_R$ are equivalent.

Having defined the equivalence relation $\simeq$ on Q^+, we define the *quotient transducer* $T_\simeq$ as

$$T_\simeq = \left(Q^+/\simeq, \{q_0\}^+, \Longrightarrow_\simeq, F^+/\simeq\right)$$

where

– $Q^+/\simeq$ is the set of equivalence classes of columns,
– q_0^+ is the initial equivalence class (assuming that the initial state is left-copying, this will be one equivalence class of $\simeq$),
– $\Longrightarrow_\simeq$: $((Q^+/\simeq) \times (\Sigma \times \Sigma)) \mapsto 2^{(Q^+/\simeq)}$ is defined in the natural way as follows. For any columns x, x' and symbols a, a':

$$x \stackrel{(a,a')}{\Longrightarrow} x' \quad \Rightarrow \quad [x]_\simeq \stackrel{(a,a')}{\Longrightarrow}_\simeq [x']_\simeq$$

– $F^+/\simeq$ is the partitioning of F^+ with respect to $\simeq$ (if the final states are right-copying then F^+ is a union of equivalence classes).

Our proposed algorithm now builds a sequence $\widetilde{T}_0, \widetilde{T}_1, \widetilde{T}_2, \cdots$ of transducers. The states of each $\widetilde{T}_i$ is $Q^+/\simeq$, and its transition relation will be a subset of $\Longrightarrow_\simeq$. The procedure incrementally adds transitions in $\Longrightarrow_\simeq$ between equivalence classes, and therefore the relations accepted by $\widetilde{T}_0, \widetilde{T}_1, \cdots$ will be successively larger subsets of the relation accepted by $T_\simeq$.

Based on these ideas, here is the **algorithm** for computing a transducer for the transitive closure.

– The initial transducer $\widetilde{T}_0$ is obtained from T by taking all transitions in T and replacing all left- or right-copying states q by q^+.

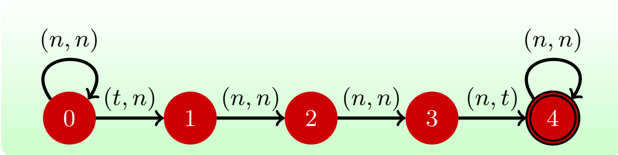

Fig. 4. The transducer T^3.

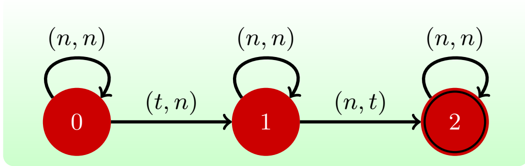

Fig. 5. The transitive closure of T.

– In each step of the procedure, $\Longrightarrow_{i+1}$ is obtained from $\Longrightarrow_i$ by adding transitions of form $X \star X' \stackrel{(a,c)}{\Longrightarrow}_{i+1} Y \star Y'$ such that $X \stackrel{(a,b)}{\Longrightarrow}_i Y$ and $X' \stackrel{(b,c)}{\Longrightarrow}_0 Y'$.
– The algorithm terminates when the relation R^+ is accepted by $\widetilde{T}_i$. This can be tested by checking if the language of $\widetilde{T}_i \circ R$ is included in $\widetilde{T}_i$.

Example (ctd.) Applying this to our example, we get the following transitions.

– First, we take all transitions in the original transducer replacing q_L by q_L^+, and replacing q_R by q_R^+.
– $q_L^+ \stackrel{(t,n)}{\Longrightarrow} q$, $q_L^+ \stackrel{(n,n)}{\Longrightarrow} q_L^+$ gives $q_L^+ \stackrel{(t,n)}{\Longrightarrow} q q_L^+$.
– $q \stackrel{(n,t)}{\Longrightarrow} q_R^+$, $q_L^+ \stackrel{(t,n)}{\Longrightarrow} q$ gives $q q_L^+ \stackrel{(n,n)}{\Longrightarrow} q_R^+ q$.
– $q_R^+ \stackrel{(n,n)}{\Longrightarrow} q_R^+$, $q \stackrel{(n,t)}{\Longrightarrow} q_R^+$ gives $q_R^+ q \stackrel{(n,t)}{\Longrightarrow} q_R^+$.
– $q q_L^+ \stackrel{(n,n)}{\Longrightarrow} q_R^+ q$, $q_L^+ \stackrel{(n,n)}{\Longrightarrow} q_L^+$ gives $q q_L^+ \stackrel{(n,n)}{\Longrightarrow} q_R^+ q q_L^+$.
– $q_R^+ q \stackrel{(n,t)}{\Longrightarrow} q_R^+$, $q_L^+ \stackrel{(t,n)}{\Longrightarrow} q$ give us $q_R^+ q q_L^+ \stackrel{(n,n)}{\Longrightarrow} q_R^+ q$.
– $q_R^+ q q_L^+ \stackrel{(n,n)}{\Longrightarrow} q_R^+ q$, $q_L^+ \stackrel{(n,n)}{\Longrightarrow} q_L^+$ gives $q_R^+ q q_L^+ \stackrel{(n,n)}{\Longrightarrow} q_R^+ q q_L^+$.

At the last point, the termination test succeeds, implying that the transducer indeed accepts the transitive closure of the original relation (the one shown in Figure 5). The new transducer thus becomes the one shown in Figure 6.

Abstraction In [23], *abstraction* techniques are applied to automata that arise in the iterative computation of $I \circ T^*$. When computing the sequence $I, I \circ T, I \circ T^2, I \circ T^3, \cdots$ the automata that arise in the computation may all be different or may be very large and contain information that is not relevant for checking whether $I \circ T^*$ has a nonempty intersection with the set of bad configurations B. Therefore, each iterate $I \circ T^n$ is abstracted by quotienting under some equivalence relation $\simeq$. In contrast to the techniques of [11,24,36,10], the abstraction does not need to preserve the language accepted, i.e., $(I \circ T^n)/\simeq$ can be any over-approximation of $I \circ T^n$ or even of $I \circ T^*$. The procedure calculates the sequence of

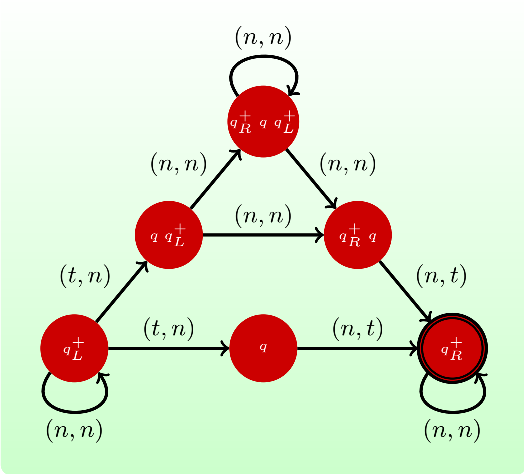

Fig. 6. The transitive closure of T as computed by the algorithm.

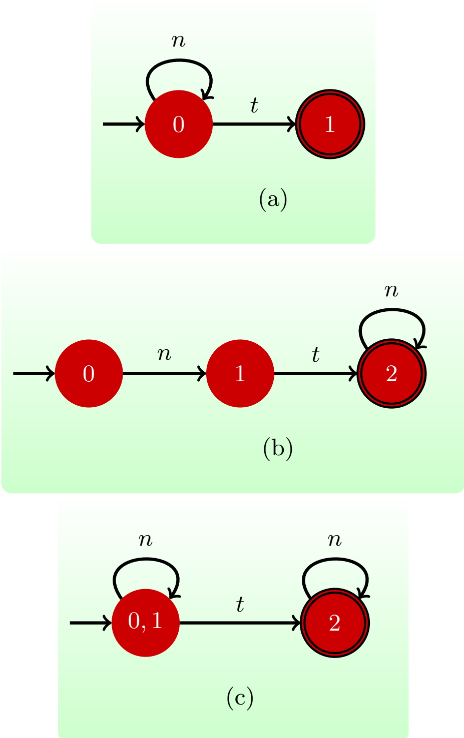

Fig. 7. (a) Automaton for B. (b) An automaton A. (c) Abstraction of A.

approximations of form $(((I \circ T)/\simeq) \circ T)/\simeq \cdots$. Convergence to a limit T^{lim} can be ensured by choosing $\simeq$ to have finite index.

If now $T^{lim} \cap B = \emptyset$, we can conclude (by $L((I \circ T^*)) \subseteq L(T^{lim})$) that $I \circ T^*$ has an empty intersection with B. Otherwise, we try to trace back the computation from B to I. If this succeeds, a counterexample has been found, otherwise the abstraction must be refined by using a finer equivalence relation, from which a more exact approximation T^{lim} can be calculated, etc.

The technique relies on defining suitable equivalence relations. One way is to use the automaton for B. We illustrate this on the token passing example. Suppose that B is given by the automaton in Fig 7, denoting that the last process has the token. Each state q in an automaton A has a *post language* $L(A, q)$ which is the set of words accepted starting from that state. For example, in the automaton for B we have $L(B, 0) = n^*t$ and $L(B, 1) = \{\epsilon\}$. The post languages are used to define $\simeq$, such that $q \simeq q'$ holds if for all states r of B we have $L(A, q) \cap L(B, r) = \emptyset$ exactly when $L(A, q') \cap L(B, r) = \emptyset$. Each equivalence class of $\simeq$ can be represented by a Boolean vector indexed by states of B, which is true on position s exactly when the equivalence class members have nonempty intersection with $L(B, s)$. This is one way to get a finite index equivalence relation.

We show an example of an automaton A in Fig 7 with its corresponding abstract version. Considering the states of A, we observe that the post languages of states 0 and 1 both have a nonempty intersection with the post language n^*t and an empty intersection with the post language containing the empty string. The post language of state 2 have an empty intersection with n^*t and an nonempty intersection with the post language containing the empty string.

If a *spurious* counterexample is found, i.e. a counterexample occurring when quotienting with an equivalence $\simeq$, but not in the original system, we need to refine the equivalence and start again. Automata representing parts of the counterexample can be used, in the same way as the automaton B above, to define an equivalence. In [23], the equivalence is refined by using *both* B *and* automata representing parts of the counterexample. This prevents the same counterexample from occurring twice. Using abstraction can potentially greatly reduce the execution time, since we only need to verify that we cannot reach B and therefore it may be that less information about the structure of $I \circ T^*$ needs to be stored.

Widening Another technique for calculating $I \circ T^*$ is to speed up the iterative computation by *widening* (*extrapolation*) techniques that try to guess the limit. The idea is to detect a repeating pattern – a regular growth – in the iterations, from which one guesses the effect of arbitrarily many iterations. The guess may be exactly the limit, or an approximation of it.

In [24, 48], the extrapolation is formulated in terms of rules for guessing $I \circ T^*$ from observed growth patterns among the approximations $I, I \circ T, I \circ T^2, \cdots$. Following [24], if I is a regular expression ρ which is a concatenation of form $\rho = \rho_1 \cdot \rho_2$, and in the successive approximations we observe a growth of form $(\rho_1 \cdot \rho_2) \circ T = \rho_1 \cdot \Lambda \cdot \rho_2$ for some regular expression Λ, then the guess for the limit $\rho \circ T^*$ is $\rho_1 \cdot \Lambda^* \cdot \rho_2$. In [48] this approach is extended to more general situations. One of these is when

ρ is a concatenation of form $\rho_1 \cdot \ldots \cdot \rho_n$ and

$$(\rho_1 \cdot \ldots \cdot \rho_n) \circ T = \bigcup_{i=1}^{n-1} \rho_1 \cdot \ldots \cdot \rho_i \cdot \Lambda_i \cdot \rho_{i+1} \cdot \ldots \cdot \rho_n$$

The guess for the limit $\rho \circ T^*$ is in this case

$$\rho_1 \cdot \Lambda_1^* \cdot \rho_2 \cdot \Lambda_2^* \cdot \ldots \cdot \Lambda_{n-1}^* \cdot \rho_n$$

For example, if $\rho = a^*ba^*$ and T is a relation which changes an a to a c, then $\rho \circ T$ is $a^*ca^*ba^* \cup a^*ba^*ca^*$ (i.e., each step adds either ca^* to the left of b or a^*c to the right). The above rule guesses the limit $\rho \circ T^*$ to be $a^*(ca^*)^*b(a^*c)^*a^*$.

Having formed a guess ρ' for the limit, we apply a *convergence test* which checks whether $\rho' = (\rho' \circ T) \cup \rho$. If it succeeds, we can conclude that $\rho \circ T^* \subseteq \rho'$. The work in [24] and [48] also provide results which state that under some additional conditions, we can in fact conclude that $\rho \circ T^* = \rho'$, i.e., that ρ' is the exact limit.

The paper [17] extends the above techniques by considering growth patterns for subsequences of $I, I \circ T, I \circ T^2, \cdots$, consisting of infinite sequences of *sample points*, noting that the union of the approximations in any such subsequence is equal to the union of the approximations in the full sequence. This idea is applied by iteratinig a special case of relations, *arithmetic transducers*, which operate on binary encodings of integers, and give a sufficient criterion for exact extrapolation.

We illustrate these approaches, using our token passing example. From the initial set $\rho_I = tn^*$, we get $\rho_I \circ T = ntn^*$, $\rho_I \circ T^2 = nntn^*$, $\rho_I \circ T^3 = nnntn^*$, and so on. The methods above detect the growth $\rho_I \circ T = n \cdot \rho_I$, and guess that the limit is n^*tn^*. In this case, the completeness results of [24, 48] allow to conclude that the guessed limit is exact.

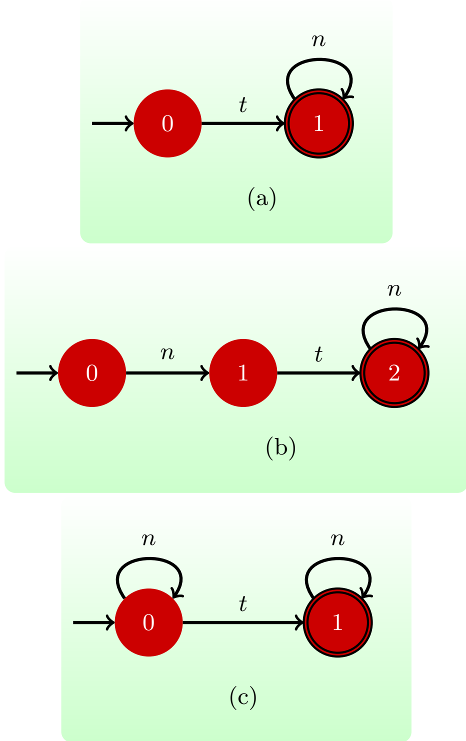

Fig. 8. (a) Automaton for ρ_I. (b) Automaton for $\rho_I \circ T$. (c) Extrapolated automaton.

4 Extensions

In the previous sections, we presented the basic concepts in RMC, where configurations are represented as finite words, and the transition relation is represented by length-preserving transducers. Below, we give an overview of some extensions of the basic framework.

Non-Length-Preserving Transducers Lifting the restriction of length-preservation from transducers allows to model more easily dynamic data structures and parameterized systems of processes with dynamic process creation. The techniques have been extended, see, e.g., [36, 17].

Infinite Words The natural extension to modeling systems by infinite words has been considered in [18], having the application to real arithmetic in mind. Regular sets and transducers must then be represented by Büchi automata. To avoid the high complexity of some operations on Büchi automata, the approach is restricted to sets that can be defined by weak deterministic Büchi automata.

Finite Trees Regular sets of trees can in principle be analyzed in the same way as regular sets of words, as was observed also in [18]. Configurations will now be finite trees; sets of configurations will be encoded by tree automata, while the transition relation will be represented by a tree transducer. With some complications, similar techniques can be used for symbolic verification [10, 26]. Some techniques have been implemented and used to verify mutual exclusion algorithms [10], to perform dataflow analysis for parallel programs with procedures [26], or to verify pointer-manipulating programs [22].

Light-Weight Techniques Several approaches to RMC have been recently proposed to avoid the use of the full class of regular languages. An example of such an approach is that of *monotonic abstraction* [8]. The main idea is to use over-approximations that allow the application of the framework of well-quasi-ordered programs [6] on the abstract system. This results in methods that are on the one hand sufficiently general to handle most existing benchmarks, and on the other hand sufficiently simple to allow efficient verification algorithms.

5 Overview of Papers

The papers included in this issue cover large parts of the topics mentioned above:

- The paper [46] describes a widening technique for computing the transitive closure of a relation induced by a transducer. This is done by deriving a sequence of automata that represent successive approximations of the transitive closure. The method works for general transducers. It does not rely on the particular relation the transducer represents. For instance, the method has been applied both to perform verification and to derive automata that represent the convex hull of a set of integers.
- The paper [27] applies widening in the case of tree-based RMC. More precisely, it extends RMC from the context of words to that of trees. Sets of configurations are now modeled by tree automata, while transition relations are represented by tree transducers. The paper is based on principles similar to those in [46], namely iteratively computing transitive closures of tree transducers, while enhancing the iterations by a widening operator. The method is applied to perform different tasks on various classes of systems such as the verification of parameterized tree networks, and data-flow analysis of multithreaded programs.
- The paper [25] describes how to apply abstraction in RMC. The goal of abstraction here is twofold, namely (i) it accelerates the computation of the transitive closure and hence increases the chances of termination, and (ii) the sizes of the generated automata are much smaller thus limiting the state explosion problem which is often the limiting factor in the application of RMC. The paper shows the application of the method to programs operating on unbounded counters, queues, stacks, and parameters. Furthermore, it shows how to extend the techniques to the case of trees.
- The paper [14] describes another technique to achieve acceleration in RMC. The acceleration operator is associated to cycles in the transition graph of the program. The acceleration is achieved by computing (in a single step) the set of all states that can be reached by iterating a cycle arbitrarily many times. In contrast to the previous techniques, the algorithm for computing the acceleration operator depends on the type of the data domains that are manipulated by the program, and on the symbolic representation chosen for representing sets of states. The paper describes acceleration operators designed for systems operating on FIFO communication channels, and for programs operating on integer- and real-valued variables.
- The paper [37] describes a lightweight approach to RMC that tries to achieve efficient solutions by avoiding the use of the full power of finite-state automata. More precisely, it introduces *monotonic abstraction* that only uses sets of states that are upward closed with respect to a certain ordering on the state space. This allows to use the framework of *well quasi-ordered programs* [6] that is in some cases much more efficient than transducer-based methods.
- The paper [1] introduces a specification formalism that combines the classical languages of Linear Temporal Logic (LTL) and Monadic Second-Order Logic (MSO). The formalism can be used to describe both safety and liveness properties. The paper describes a technique for model checking LTL(MSO) which is adapted from the automata-theoretic approach by translating a formula to a Büchi regular transition system with a regular set of accepting states. Then, it uses RMC techniques to perform searching.

References

1. Parosh A. Abdulla, Bengt Jonsson, Marcus Nilsson, Juline d'Orso, and Mayank Saksena. Regular model checking for S1S + LTL, 2012. In this volume.
2. Parosh Aziz Abdulla. Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic*, 16(4):457–515, 2010.
3. Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on sat-solvers. In Graf and Schwartzbach [41], pages 411–425.
4. Parosh Aziz Abdulla, Ahmed Bouajjani, and Bengt Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In Hu and Vardi [42], pages 305–318.
5. Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling global conditions in parameterized system verification. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 134–145. Springer, 1999.
6. Parosh Aziz Abdulla, Karlis Cerans, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *LICS*, pages 313–321, 1996.
7. Parosh Aziz Abdulla, Yu-Fang Chen, Lukás Holík, Richard Mayr, and Tomáš Vojnar. When simulation meets antichains. In Javier Esparza and Rupak Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 158–174. Springer, 2010.
8. Parosh Aziz Abdulla, Giorgio Delzanno, Noomene Ben Henda, and Ahmed Rezine. Regular model checking without transducers (on efficient verification of parameterized systems). In Orna Grumberg and Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 721–736. Springer, 2007.
9. Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *LICS*, pages 160–170. IEEE Computer Society, 1993.
10. Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In Brinksma and Larsen [28], pages 555–568.

11. Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, and Julien d'Orso. Regular model checking made simple and efficient. In Lubos Brim, Petr Jancar, Mojmír Kretínský, and Antonín Kucera, editors, *CONCUR*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130. Springer, 2002.
12. Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-checking for real-time systems. In *LICS*, pages 414–425. IEEE Computer Society, 1990.
13. Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Ph. Schnoebelen. Flat acceleration in symbolic model checking. In Doron Peled and Yih-Kuen Tsay, editors, *ATVA*, volume 3707 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2005.
14. Bernard Boigelot. Domain-specific regular accelaration, 2012. In this volume.
15. Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds (extended abstract). In Rajeev Alur and Thomas A. Henzinger, editors, *CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1996.
16. Bernard Boigelot, Patrice Godefroid, Bernard Willems, and Pierre Wolper. The power of qdds (extended abstract). In Pascal Van Hentenryck, editor, *SAS*, volume 1302 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 1997.
17. Bernard Boigelot, Axel Legay, and Pierre Wolper. Iterating transducers in the large (extended abstract). In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235. Springer, 2003.
18. Bernard Boigelot, Axel Legay, and Pierre Wolper. Omega-regular model checking. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 561–575. Springer, 2004.
19. Bernard Boigelot and Pierre Wolper. Symbolic verification with periodic sets. In David L. Dill, editor, *CAV*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 1994.
20. Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *CONCUR*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.
21. Ahmed Bouajjani and Peter Habermehl. Symbolic reachability analysis of fifo channel systems with nonregular sets of configurations (extended abstract). In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *ICALP*, volume 1256 of *Lecture Notes in Computer Science*, pages 560–570. Springer, 1997.
22. Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz, and Tomáš Vojnar. Abstract regular tree model checking of complex dynamic data structures. In Kwangkeun Yi, editor, *SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 52–70. Springer, 2006.
23. Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Abstract regular model checking. In Rajeev Alur and Doron Peled, editors, *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer, 2004.
24. Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In Emerson and Sistla [39], pages 403–418.
25. Ahmed Bouajjani, Adam Rogalewicz Peter Habermehl, and Tomáš Vojnar. Abstract regular (tree) model checking, 2012. In this volume.
26. Ahmed Bouajjani and Tayssir Touili. Extrapolating tree transformations. In Brinksma and Larsen [28], pages 539–554.
27. Ahmed Bouajjani and Tayssir Touili. Widening techniques for regular tree model checking, 2012. In this volume.
28. Ed Brinksma and Kim Guldstrand Larsen, editors. *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*. Springer, 2002.
29. Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.
30. Olaf Burkart and Bernhard Steffen. Model checking for context-free processes. In Rance Cleaveland, editor, *CONCUR*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer, 1992.
31. Olaf Burkart and Bernhard Steffen. Pushdown processes: Parallel composition and model checking. In Bengt Jonsson and Joachim Parrow, editors, *CONCUR*, volume 836 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 1994.
32. Didier Caucal. On the regular structure of prefix rewriting. *Theor. Comput. Sci.*, 106(1):61–86, 1992.
33. Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
34. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dexter Kozen, editor, *Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.
35. Rance Cleaveland. Pragmatics of model checking: An sttt special section. *STTT*, 2(3):208–218, 1999.
36. Dennis Dams, Yassine Lakhnech, and Martin Steffen. Iterating transducers. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 286–297. Springer, 2001.
37. Giorgio Delzanno and Ahmed Rezine. A lightweight regular model checking approach for parameterized systems, 2012. In this volume.
38. Laurent Doyen and Jean-François Raskin. Antichains for the automata-based approach to model-checking. *Logical Methods in Computer Science*, 5(1), 2009.
39. E. Allen Emerson and A. Prasad Sistla, editors. *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.
40. Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In Emerson and Sistla [39], pages 232–247.

41. Susanne Graf and Michael I. Schwartzbach, editors. *Tools and Algorithms for Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, volume 1785 of *Lecture Notes in Computer Science*. Springer, 2000.
42. Alan J. Hu and Moshe Y. Vardi, editors. *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*. Springer, 1998.
43. Bengt Jonsson and Marcus Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In Graf and Schwartzbach [41], pages 220–234.
44. Yonit Kesten, Oded Maler, Monica Marcus, Amir Pnueli, and Elad Shahar. Symbolic model checking with rich assertional languages. *Theor. Comput. Sci.*, 256(1-2):93–112, 2001.
45. Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. Mona implementation secrets. *Int. J. Found. Comput. Sci.*, 13(4):571–586, 2002.
46. Axel Legay. Extrapolating (omega-)regular model checking, 2012. In this volume.
47. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.
48. Tayssir Touili. Regular model checking using widening techniques. *Electr. Notes Theor. Comput. Sci.*, 50(4):342–356, 2001.
49. Pierre Wolper and Bernard Boigelot. Verifying systems with infinite but regular state spaces. In Hu and Vardi [42], pages 88–97.