Regular Model Checking: Evolution and Perspectives

Parosh Aziz Abdulla(✉)

Uppsala University, Uppsala, Sweden
parosh@it.uu.se

Abstract. We describe the main ideas behind the framework of regular model checking in a tutorial-like manner. First, we recall the original framework, and then describe an over-approximation scheme that we have designed to make the method more scalable. Finally, we point to some directions for future work.

1 Introduction

During the last two decades, a vast research effort has been devoted to extending the applicability of algorithmic verification to infinite-state systems, using approaches based on abstraction, deductive techniques, decision procedures, etc. One primary approach has been to extend the paradigm of symbolic model checking [21] to new classes of models such as timed automata, push-down systems, systems with unbounded communication channels, Petri nets, and systems that operate on integers and reals (e.g., [13,18,22,23]).

Regular Model Checking (Rmc) is one such an extension. In Rmc, regular sets represent sets of states and regular transducers represent transition relations. Such sets and relations are typically defined over finite or infinite words or tree structures. Most initial works considered models whose configurations can be represented as finite words of arbitrary length over a finite alphabet. Such models include parameterized systems consisting of an arbitrary number of homogeneous finite-state processes connected in a linear or ring-formed topology, as well as systems that operate on queues, stacks, integers, and other linear data structures. Regular model checking was first advocated by Kesten et al. [33] and by Boigelot and Wolper [35], as a uniform framework for analyzing several classes of parameterized and infinite-state systems. The idea was that regular sets would provide an efficient representation of infinite-state spaces, and play a role similar to the role that Binary Decision Diagrams (BDDs) used to play for symbolic model checking of finite-state systems. We can then exploit automata-theoretic algorithms for manipulating regular sets. Such algorithms have been successfully implemented, e.g., in the Mona [31] system.

A generic task in symbolic model checking is to verify safety or liveness properties by computing properties of the set of reachable states. For finite-state systems, this is typically done by state-space exploration (which is guaranteed to

E.-R. Olderog et al. (Eds.): Jonsson Festschrift, LNCS 13030, pp. 78–96, 2021.
https://doi.org/10.1007/978-3-030-91384-7_5

terminate). For infinite-state systems, the procedure terminates only if there is a bound on the distance (in number of transitions) from the initial configurations to any reachable configuration. An analogous observation holds if we perform reachability analysis backwards, by iteration-based methods [25,34] from a set of target configurations. A parameterized or infinite-state system does not have such a bound, and the model checking problem for such systems can even be undecidable. In contrast to the deductive application of systems like Mona [14], the goal in regular model checking is to verify system properties algorithmically (automatically). One way to accomplish that is devising so-called *acceleration techniques* that calculate the effect of an arbitrarily long sequence of transitions. This problem has been addressed in regular model checking [11,20,32]. In general, the effect of acceleration is not computable. However, computability have been obtained for certain classes [32]. 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 [2,15,16,19], systems with stacks [18,24,28,30], and systems with counters [17,26].

While RMC, in its pure form, is an elegant and theoretically interesting framework, it became eventually clear that the applicability of the method was limited. The main bottleneck was the automata representation which would not scale beyond small examples. A main research direction has been to find over-approximations that allow more light-weight symbolic representations than the full class of regular languages, while still being sufficiently precise to successfully carry out the verification of non-trivial examples.

In this tutorial, I will use two running examples to explain the main ideas behind the two approaches.

2 Framework

We describe the framework of RMC, using a running example, namely a simple token passing protocol.

2.1 Regular Model Checking

In its simplest form, the RMC framework represents a transition system in the following manner.

- A *configuration* (state) of the system is a word over a finite alphabet Σ.
- Sets of configurations are represented by regular sets over Σ. In particular, this applies to the set of *initial configurations*.
- The *transition relation* is a regular and length-preserving relation on Σ^*. We represent the relation by a finite-state transducer $\mathcal{T}$ over $(\Sigma \times \Sigma)$. The transducer $\mathcal{T}$ accepts all words $(a_1, a'_1) \cdots (a_n, a'_n)$ (of pairs of elements) such that $(a_1 \cdots a_n \ a'_1 \cdots a'_n)$ is in the transition relation. Sometimes, the transition relation is given as a union of a finite number of relations, each of which is called an *action*.

In this paper, we often abuse notation and identify the transducer $\mathcal{T}$ with the relation defined by $\mathcal{T}$. We will apply the transducer relation on (regular) sets of configurations. For a set $\mathcal{C}$ of configurations and a binary relation R on configurations, let $\mathcal{C} \circ R$ denote the set of configurations w such that $w' \; R \; w$ for some $w' \in \mathcal{C}$. Let R^+ denote the transitive closure of R and R^* denote the reflexive transitive closure of R.

The simple instance of Rmc, introduced in the previous paragraphs, is already powerful and can model several interesting classes of systems. On example is *parameterized systems* which consist of arbitrary numbers of linear or ring-shaped collections of processes. We can do this by letting each position in the word represent one process in the system. It is also possible to model programs that operate on linear unbounded data structures such as queues, stacks, integers, etc. For instance, a stack can be modeled by letting each position in the word represent the corresponding position in the stack.

For reachability properties, the requirement of the transducer to be length-preserving is not a restriction. For instance, in the case of parameterized systems, the length-preserving condition implies that we cannot dynamically create new processes. However, the system can initially contain an arbitrary but bounded number of processes which are "statically allocated". We can then faithfully model all finite computations of the system, by initially allocating sufficiently many processes in their configurations. Thus, the restriction to length-preserving transducers introduces no limitations for analyzing safety properties, but may incur restrictions on the ability to specify and verify liveness properties of systems with dynamically allocated data structures. The latter follows from the fact that liveness properties quantify over the set of infinite computations. Therefore, restricting the lengths of the configurations makes it impossible to faithfully model all infinite computations of the system.

2.2 Examples

In Fig. 1 we consider a *token passing protocol*: a simple parameterized system consisting of an arbitrary (but finite) number of processes organized in a linear fashion. Initially, the left-most process has the token. 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 configurations is given by the regular expression tn^* (Fig. 1(a)), i.e., in an initial configuration, the left-most process, and only the left-most process, has the token. The transition relation is represented by the transducer in (Fig. 1(b)). 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

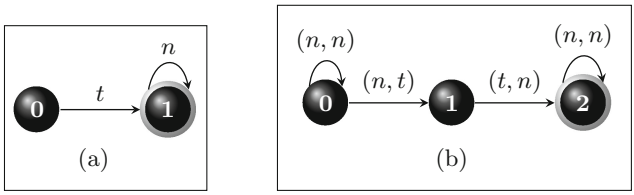


Fig. 1. The token passing protocol: (a) the set of initial configurations, (b) the transducer describing the transition relation.

channel. A configuration of the system is a word over the alphabet $Q \cup M \cup \{\bot\}$, where the *padding symbol* $\bot$ represents an empty position in the channel. For instance the word $q_1 \bot m_3 m_1 \bot \bot$ 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 $Q\bot^* M^* \bot^*$.

By allowing arbitrarily many padding symbols $\bot$, one can model channels of arbitrary but bounded length. As an example, assume that the stack alphabet is the set $\{a, b\}$. Then, the action where the process sends the message m to the channel and changes state from q to q' is modeled by the transducer in Fig. 2.

2.3 Verification Problems

We will consider two types of verification problems in this paper.

The first problem is verification of *safety properties*. A safety property is typically of form "bad things do not happen during system execution". A safety property can be verified by solving a *reachability* problem. Formulated in the regular model checking framework, the corresponding problem is the following: given a 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 exists 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

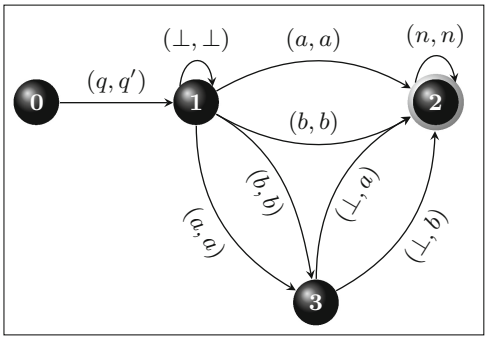


Fig. 2. The push operation in a stack.

"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.

Sets like $I \circ T^*$ and relations like T^+ are in general not regular or even computable (note that T could model the computation steps of a Turing machine). Even if they are regular, they are sometimes not effectively computable. In these cases, the above verification problems cannot be solved by the proposed techniques. Therefore, a main challenge in regular model checking is to design semi-algorithms which successfully compute such sets and relations for as many examples as possible. We will look at this aspect in the next section.

3 Transducers

In Sect. 2, we mentioned that we can carry out verification by computing a representation of $I \circ T^*$ (or T^+) for some transition relation T and some set of configurations I. Given a set of *bad* configurations B, we want to check whether $I \circ T^* \cap B \neq \emptyset$. Algorithms for regular model checking are usually based on starting from I and repeatedly applying T. As a running illustration, we will consider the problem of computing the transitive closure T^+ for the transducer in Fig. 1. A first attempt is to compute T^n, i.e., to compute 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 Fig. 4.

A transducer for T^+ is one whose relation represents that the token gets passed an arbitrary number of times. There (infinitely) many transducers characterizing this relation. One such a transducer is depicted in Fig. 3.

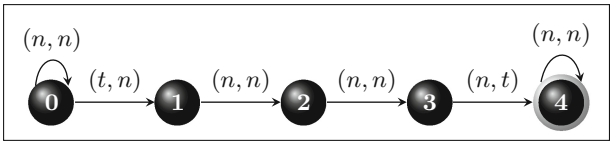


Fig. 3. Applying the token passing transducer relation three times.

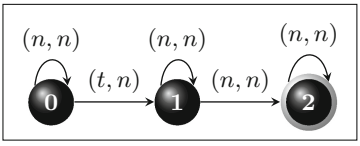


Fig. 4. A transducer characterizing the transitive closure of the token passing transducer.

The challenge is to derive (one of) these transducers *algorithmically*. Obviously, we cannot do this naively by simply computing the approximations T^n for $n = 1, 2, 3, \cdots$, since such a procedure would not converge. We can solve the problem by applying *acceleration* or *widening* techniques that can compute a representation of T^+. Below, we present a technique based on acceleration to illustrate the idea.

Acceleration techniques are usually based on *quotienting* of transducers that represent approximations of T^n for some value(s) of n. This involves finding an *equivalence relation* $\simeq$ on the states of approximations, and to merge equivalent states, obtaining a quotient transducer. For instance, in the transducer that represents T^3 above, we can define the states 1, 2, and 3 to be equivalent. By merging them, we obtain the transducer $T^3/\simeq$ which in this example happens to be equivalent to T^+.

One problem is that quotienting in general increases the language accepted by a transducer: $L(T^n) \subseteq L(T^n/\simeq)$, usually with strict inclusion. This problem was resolved in [10,11,20,27] by characterizing equivalence relations $\simeq$ such that T^+ is equivalent to $(T/\simeq)^+$ for any transducer T, i.e., the quotienting does not increase the transitive closure of the transducer. To explain the idea, let us first build explicitly a transducer for T^+ as the union of transducers T^n for $n = 1, 2, 3, \cdots$. Each state of T^n is labeled with a sequence of states from T, resulting from the product construction using n copies of T. The result is called the *history transducer*. The history transducer corresponding to the token passing protocol is shown in Fig. 5. Recall minimization algorithms for automata. They are based on building a *forward* bisimulation $\simeq_F$ on the states, and then carry out minimization by quotienting. For instance, in the history transducer of Fig. 5, all states with names of form 2^i1 for any $i \geq 0$ are forward bisimilar. Analogously, we can find a backward bisimulation $\simeq_B$. For instance, all states with names of form 10^i, $i \geq 0$, are backward bisimilar. Dams et al. [27] showed how to combine a forward $\simeq_F$ and a backward bisimulation $\simeq_B$ into an equivalence relation $\simeq$

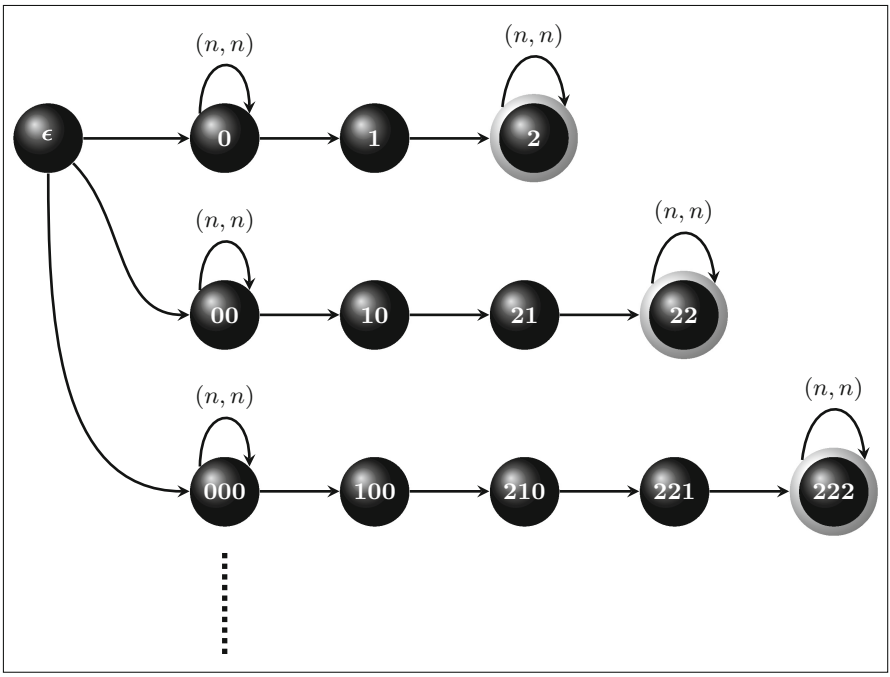


Fig. 5. The history transducer for the token passing protocol.

which preserves the transitive closure of the transducer. In [12], this result was generalized to consider *simulations* instead of bisimulations. The simulations can be obtained by computing properties of the original automaton T (as in [11,12]), or on successive approximations of T^n (as in [27]).

From the results in [12] it follows for the history transducer that the states with names in 2^i1 can be merged for $i \geq 1$, and the same holds for 10^i. The equivalence classes for that transducer would be 2^+, 0^+, 10^+, 2^+1 and 2^+10^+. Hence, it can be quotiented to the transducer depicted in Fig. 6, which, in turn, can be minimized to the three-state representation shown in Fig. 4.

4 Monotonic Abstraction

In this section, we present an approach that avoids using the full power of regular languages and transducers. Instead, we compute an over-approximation of the set of reachable configurations through a particular technique which we call *monotonic abstraction*. We will instantiate the framework to a special class of parameterized systems. In this section, a *parameterized system* consists of an arbitrary number of identical processes each of which is a finite-state process. The processes are organized as a linear array. In each step in the execution of the system, one process, called the *active process*, changes state. The rest of

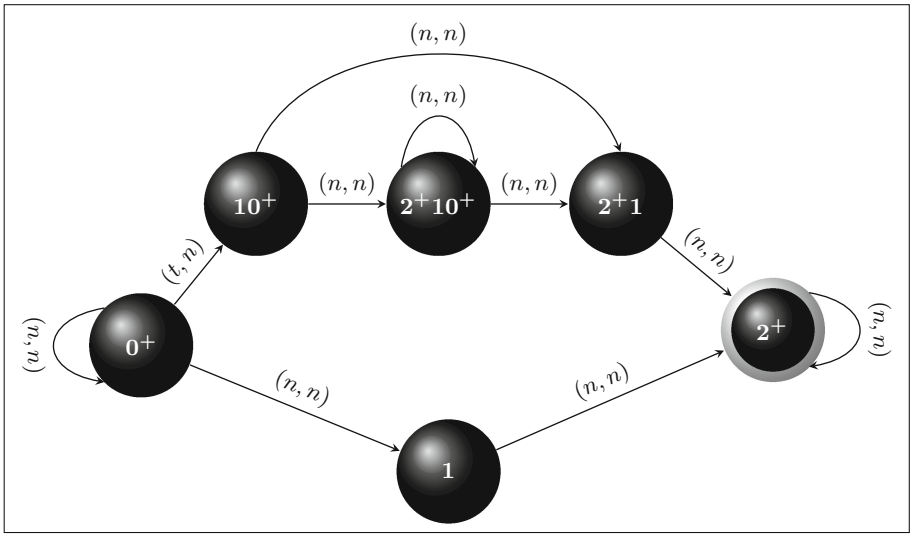


Fig. 6. The history transducer for the token passing protocol.

the processes, called the *passive processes*, do not change states. We call the passive processes to the left of the active process the *left context* of the active process. The *right context* is defined analogously. The active process may perform a *local transition* in which it changes its state independently of the states of the passive processes. The active process may also perform a global transition in which it checks the states of the passive processes. A global transition is either *universally* or *existentially* quantified. An example of a universal condition is that *all* processes in the left context of the active process should be in certain states. In an existential transition we require that *some* (rather than all) processes should be in certain states.

We use a running example of a mutual exclusion protocol, among an array of processes, where each process is of the form depicted in Fig. 7. The process has four local states, namely the green, black, blue, and red states. We represent these states by colored balls, , , and . Sometimes, when no confusion arises, we refer to a process in a configuration by its state, so we say e.g. "the red process" rather than "the process in its red state".

Initially, all the processes are green (they are idle). When a process becomes interested in accessing the critical section (which corresponds to the red state), it declares its interest by moving to the black state. This is described by the global universal transition rule t_1 in which the move is allowed only if all the other processes are in their green or black states. The universal quantifier labeling t_1 encodes the condition that all other processes (whether in the left or the right context – hence the index LR of the quantifier) of the active process should be green or black. In the black state, the process may move to the blue state through the local transition t_2 (in which the process does not need to check the

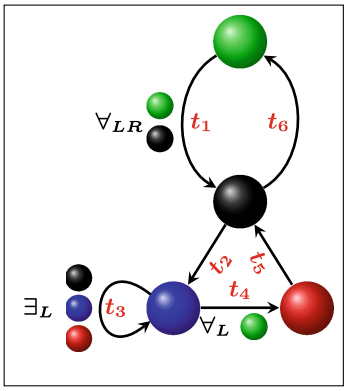


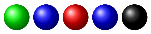
Fig. 7. One process in the mutual exclusion protocol (Color figure online)

states of the other processes). Notice that any number of processes my cross from the initial (green) state to the black state. However, once the first process has crossed to the blue state, it "closes the door" on the processes which are still in their green states. These processes will no longer be able to leave their green states until the door is opened again (when no process is blue or red). From the set of processes which have declared interest in accessing the critical section (those which have left their green states and are now black or blue) the leftmost process has the highest priority. This is encoded by the global universal transition t_4 where a process may move from its blue state to its red state only subject to the universal condition that all processes in its left context are green (the index L of the quantifier stands for "Left"). If the process finds out, through the existential global condition, that there are other processes that are black, blue, or red, then it loops back to the blue state through the existential transition t_3. Once the process leaves the critical section, it will return back to the black state through the local transition t_5. In the black state, the process chooses either to try to reach the critical section again, or to become idle (through the local transition t_6).

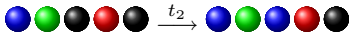
Formally, we represent a parameterized systems P by a pair $\langle Q, T\rangle$, where Q is the set of the local states of the processes, and T is the set of transition rules which define the behaviour of each process. In the above example, the set Q consists of four states (green, black, blue, and red), while the set T consists of six rules, namely three local rules (t_2, t_5, and t_6), two universal rules (t_1 and t_4), and one existential rule (t_3).

4.1 Transition System

A parameterized system $P = \langle Q, T\rangle$ induces a transition systems $T = \langle \mathcal{C}, \longrightarrow\rangle$, where $\mathcal{C}$ is the set of configurations and $\longrightarrow$ is a transition relation on $\mathcal{C}$. A configuration is a word in Q^*, where each element of the word represents the local state of one process.

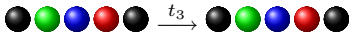
Let us consider the example of Fig. 7. The word  represents a configuration in an instance of the system with five processes that are in their green, blue, red, blue, and black states, in that order. Since there is no bound on the configuration sizes, the set of configurations is infinite. We define the transition relation $\longrightarrow := \cup_{t \in T} \xrightarrow{t}$, where $\xrightarrow{t}$ is a relation on configurations that captures the effect of the transition rule t. The definition of $\longrightarrow$ depends on the type of t (whether it is local, existential, or universal). We will consider three transition rules from Fig. 7 to illustrate the idea.

The local rule t_2 induces transitions of the form



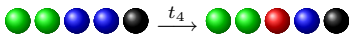
Here, the active process changes its local state from black to blue.

The existential rule t_3 induces transitions of the form



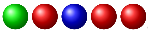
The blue process can perform the transition since there is a black process in its left context. However, the transition is not enabled from the configuration  , since there are no red, blue, or black processes in the left context of the process trying to perform the transition.

The universal rule t_4 induces transitions of the form t_4



The active process  can perform the transition since all the processes in its left con- text are green. On the other hand, neither of the blue processes can perform the transition form the configuration  since, for each one of them, there is at least one process in its left context which is not green. As usual, we use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. For sets C_1 and C_2 of configurations, we use $C_1 \xrightarrow{*} C_2$ to denote that there are configurations $c_1 \in C_1$ and $c_2 \in C_2$ such that $c_1 \xrightarrow{*} c_2$.

An initial configuration is one in which all processes are in their initial (green) states. In this section, we use *Init* to denote the set of initial configurations. Examples of initial configurations are  and  corresponding to instances of the system with two and four processes respectively. Notice that there is an infinite set of initial configurations, namely one for each size of the system.

As mentioned above, the protocol is intended to observe mutual exclusion. In other words, we are interested in verifying a safety property. To do this we characterize the set *Bad* of configurations: all configurations which contain at least two red processes. Examples of configurations in *Bad* are  , and  . Showing the safety property amounts to proving that the protocol, starting from an initial configuration, will never reach a bad configuration. In other words, we want to answer the question whether $Init \xrightarrow{*} Bad$.

4.2 Ordering

We define an ordering on configurations, which we use to define bad sets of configurations, and hence also to formulate the class of safety properties which we consider. For configurations c_1 and c_2, we use $c_1 \preceq c_2$ to denote that c_1 is a (not necessarily contiguous) subword of c_2. For instance, we have  $\preceq$  . A set U of configurations is said to be *upward-closed*, if whenever $c \in U$ and $c \preceq c'$ then $c' \in U$. For a configuration c, we use $\widehat{c}$ denote the upward-closed set $U := \{c' \mid c \preceq c'\}$, i.e., $\widehat{c}$ contains all configurations which are larger than c w.r.t. the ordering $\preceq$. In such a case, we call c the generator of U.

We are interested in upward-closed sets for two reasons. First, all sets of bad configurations which we work with are upward-closed. For instance, in the above example, the set *Bad* of configurations violating mutual exclusion are those which contain at least two red processes. The set is upward-closed since whenever a configuration contains two red processes then any larger configuration will also contain (at least) two red processes. The second reason why we are interested in upward-closed sets is that they have an efficient symbolic representation. In fact, it can be shown that each upward-closed set can be characterized by a finite set of generators. More precisely, for an upward-closed set U, there are configurations $c_1, \ldots, c_n$ with $U = \widehat{c_1} \cup \cdots \cup \widehat{c_n}$. For instance, the set *Bad* above has a single generator, namely  . Thus, operations which manipulate upward-closed sets can be translated into operations which manipulate words. In this manner we avoid using the full power of regular languages, when performing reachability analysis. This makes monotonic abstraction more efficient in practice compared to the automata-based methods such as the one we described in Sect. 3.

We will check safety properties using backward reachability analysis. For a set C of configurations, we define $Pre\,(C) := \{c \mid \exists c' \in \mathcal{C}.\, c \longrightarrow c'\}$. In other words, the set contains exactly all configurations from which a configuration in C can be reached through a single application of the transition relation.

To solve the safety problem, we present a scheme for backward reachability analysis. The scheme is an instantiation of the framework of *well-structured systems* [3,29]. We start with the set *Bad* of bad configurations which is upward-closed. Then, we apply the function *Pre* repeatedly generating a sequence $U_0, U_1, U_2, \ldots$ of sets of configurations, where $U_0 = Bad$, and $U_{i+1} = U_i \cup Pre\,(U_i)$, for $i \geq 0$. We observe that the set U_i characterizes the set of configurations from which the set *Bad* is reachable within i steps. We would like the sets U_i to be upward-closed (so that we can represent them by their finite sets of generators). In order to achieve that, we introduce a sufficient condition, namely that of *monotonicity*. Monotonicity implies that $Pre\,(U)$ is upward-closed whenever U itself is upward-closed. Since U_0 is upward-closed by definition, monotonicity would imply that all the sets U_i are upward-closed.

A transition system is said to be *monotone* if $\preceq$ forms a simulation on the set $\mathcal{C}$ of configurations. In other words, for all configurations c_1, c_2, c_3, whenever $c_1 \longrightarrow c_2$ and $c_1 \preceq c_3$ then $c_2 \longrightarrow c_4$ for some c_4 with $c_3 \preceq c_4$.

Monotonicity implies that upward-closedness is preserved through the application of *Pre*. The reasoning goes as follows. Consider an upward-closed set

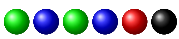
U. Let $c_1 \in \mathit{Pre}(U)$ and let $c_2 \succeq c_1$. We will show that $c_2 \in \mathit{Pre}(U)$. Since $c_1 \in \mathit{Pre}(U)$, we know by definition that there is a $c_3 \in U$ such that $c_1 \longrightarrow c_3$. By monotonicity it follows that there is a c_4 such that $c_3 \preceq c_4$ and $c_2 \longrightarrow c_4$. From $c_3 \in U$ and $c_3 \preceq c_4$ it follows that $c_4 \in U$. This means that we have found a configuration $c_4 \in U$ such that $c_2 \preceq c_4$, which implies that $c_2 \in \mathit{Pre}(U)$.

4.3 Abstraction

We define an abstraction that generates an over-approximation of the transition system. The abstract transition system is monotone, thus allowing to work with upward-closed sets. We first show that local and existential transitions are monotone, and hence need not be approximated. Therefore, we only provide an over-approximation for universal transitions. Consider the transition

$c_1 =$  $\xrightarrow{t_2}$  $= c_3$

in which a process changes state from black to blue. Consider the configuration $c_2 =$  that is larger than c_1. Clearly, c_2 can perform the local transition

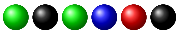
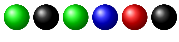
$c_2 =$  $\xrightarrow{t_2}_A$  $= c_4 \succeq c_3$

In general, local transitions are monotone, since the active process in the small configuration (the black process in c_1) also exists in the larger configuration (i.e., c_2). A local transition does not check or change the states of the passive processes; and hence the larger configuration c_2 is also able to perform the transition, while maintaining the ordering $c_3 \preceq c_4$.

Consider the existential transition

$c_1 =$  $\xrightarrow{t_3}_A$  $= c_3$

We can divide the configuration c_1 to three parts: the *active process*  , the *left context*  , and the *right context*  . Furthermore, the left context contains a *witness*  which enables the transition. Consider the configuration $c_2 =$  that is larger than c_1. Also, the configuration c_2 comprises three parts: the active process  , the left context  , and the right context  . The left context of c_2 is larger than the left context of c_1, and hence the former will also contain the witness  , which means c_2 can perform the same transition

$c_2 =$  $\xrightarrow{t_3}_A$  $= c_4 \succeq c_2$

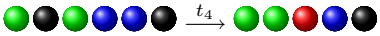
While local and existential transitions are monotone, universal transitions are *not*. To see the reason, we consider the transition

$c_1 =$  $\xrightarrow{t_4}$  $= c_3$

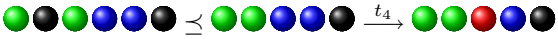
The transition is enabled since all processes in the left context of the active process satisfy the condition of the transition (they are green). Consider the

configuration $c_2 =$  . Although $c_1 \preceq c_2$, the transition t_4 is not enabled from c_2 since the left context of the active process contains processes that violate the condition of the transition. This means that universal transitions are not monotone.

In order to deal with non-monotonicity of universal transitions, we will work with an abstract transition relation $\longrightarrow_A$ that is an over-approximation of the concrete transition relation $\longrightarrow$. We call $\longrightarrow_A$ the *monotonic abstraction* of $\longrightarrow$. We let $\stackrel{t}{\longrightarrow}_A$ coincide $\stackrel{t}{\longrightarrow}$ when t is a local or an existential transition. The reason is that, in these two cases, the relation is monotone and hence no over-approximation is needed. For the case when t is universal, we let $c_1 \stackrel{t}{\longrightarrow}_A c_2$ if there is a $c_1' \preceq c_1$ with $c_1' \stackrel{t}{\longrightarrow}_A c_2$. In other words, we allow c_1 to first "transform" to a smaller configuration from which it can perform the transition. For instance



since



The abstract transition relation $\longrightarrow_A$ is monotone also w.r.t. universal transitions, since for configurations c_1, c_2, c_3, and a transition t, if $c_1 \preceq c_2$ and $c_1 \stackrel{t}{\longrightarrow}_A c_3$ then, by definition $c_2 \stackrel{t}{\longrightarrow}_A c_3$. Notice that the over-approximation essentially deletes those processes in the configuration that violate the condition of the universal transition. Since $\longrightarrow_A$ is an over-approximation of the original transition relation $\longrightarrow$, it follows that if a safety property holds in the abstract model, then it will also hold in the original model.

4.4 Backward Reachability

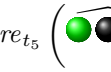
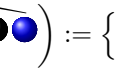
We present a backward algorithm for approximated reachability analysis. Here, we compute the function *Pre* w.r.t. the abstract relation $\longrightarrow_A$ rather than the concrete relation $\longrightarrow$. This means that we can work with upward-closed sets in the scheme for backward reachability analysis that we presented earlier. Recall that we generate a sequence $U_0, U_1, U_2, \ldots$ of sets of configurations where $U_0 = \mathit{Bad}$, and $U_{i+1} = U_i \cup \mathit{Pre}\,(U_i)$, for $i \geq 0$. Since U_0 is upward-closed by definition, and $\longrightarrow_A$ is monotone, all the sets U_i are upward-closed.

Recall that each set can be represented by its finite set of generators. Given a configuration c, we show below how to compute the set of generators for the set $\mathit{Pre}\,(\widehat{c})$. This means that we only need to work with generators (configurations) as a symbolic representation of the sets which arise in the algorithm.

Now, we show that the algorithm is guaranteed to terminate. Suppose that the algorithm, during its execution, produces two generators c_1, c_2 such that $c_1 \preceq c_2$. Since $\widehat{c_2} \subseteq \widehat{c_1}$, we can safely discard c_2 from the analysis without the loss of precision. In such a case, we say that c_2 is *subsumed* by c_1. Discarding configurations in this manner makes it possible to apply the well-structured framework [3,29]. According to the framework, termination of the algorithm is guaranteed since $\preceq$ is a *well quasi-ordering*. That $\preceq$ is a well quasi-ordering

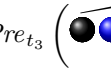
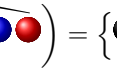
means that for any infinite sequence $c_0, c_1, c_2, \ldots$ of configurations, there are $i < j$ such that $c_i \preceq c_j$.

It remains to show that we can compute the generators of $Pre\,(\widehat{c})$ for any configuration c. We define $Pre\,(\widehat{c}) := \cup_{t \in T} Pre_t\,(\widehat{c})$ where $Pre_t\,(\widehat{c})$ gives the generators of the set of configurations from which we can reach $\widehat{c}$ through one application of the transition rule t. The definition of Pre_t depends on the type of t (whether it is local, existential, or universal). We will consider the different transition rules in Fig. 7 to illustrate how to compute Pre_t. For the local rule t_5, we have

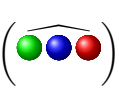
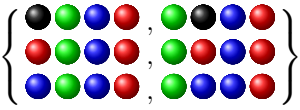
Pre_{t_5}  $:=$ 

In other words, the predecessor set is characterized by one generator, namely  . Strictly speaking, the set contains also a number of other configurations such as  . However such configurations are subsumed by the original configuration, and therefore we will not include them in the set.

For existential transitions, there are two cases depending on whether a witness exists or not in the configuration. Consider the existential rule t_3 in Fig. 7. We have

Pre_{t_3}  $=$ 

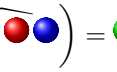
In this case, there is a witness, namely,  in the left context of the active process  . On the other hand, we have

Pre_{t_3}  $:=$ 

In this case there is no witness available in the left context of the active process. Therefore, we add a witness explicitly in each possible state ( ,  , or ), and each possible place in the left context of the active process. Notice that the sizes of the new generators (four processes) is larger than the size of the original configuration (three processes). This means that the sizes of the configurations generated by the backward algorithm may increase, and hence there is a priori no bound on the sizes of the configurations. However, termination is still guaranteed due to the well quasi-ordering of $\preceq$. For universal conditions, let us consider the universal rule t_4 in Fig. 7. We have

Pre_{t_4}  $= \emptyset$

since there is a black process in the left context of the potential active process (which is in state ). On the other hand

Pre_{t_4} $=$ 

since all processes in the left context of the active process are in their green states.

4.5 Example

We show how the backward reachability algorithm runs on our example. We start by the generator

$g_0 =$ 

of the set of bad configuration. The only transition which can be enabled backwards from a red state, is the one induced by the rule t_4. From the two red processes in g_0, only the left one can perform t_4 backwards (the right process cannot perform t_4 backwards since its left context contains a process not satisfying the condition of the quantifier):

$$Pre_{t_4}(g_0) = \left\{ g_1 = \text{  } \right\}$$

From g_1, two rules are enabled backwards (both from the blue process): the local rule t_2

$$Pre_{t_2}(g_1) = \left\{ g_2 = \text{  } \right\}$$

and the existential rule t_3

$$Pre_{t_3}(g_1) = \left\{ \text{  } \right\}$$

Since a witness is missing in the left context, we add it explicitly. All the three generators in $Pre_{t_3}(g_1)$ are subsumed by g_1. One rule is enabled backwards from g_2, namely the local rule t_5 from the black process

$$Pre_{t_5}(g_1) = \left\{ g_0 = \text{  } \right\}$$

Notice that the universal transition t_1 is not enabled from the black process, since there is another process (the red process) in the configuration that violates the condition of the quantifier. At this point, the algorithm terminates, since it is not possible to provide any new generators which are not subsumed by the existing ones.

Since there is no initial configuration (with only green processes) in $\widehat{g_0} \cup \widehat{g_1} \cup \widehat{g_2}$, the set of bad configurations is not reachable from the set of initial configurations in the abstract semantics. Therefore, we can conclude that the set of bad configurations is not reachable from the set of initial configurations in the concrete semantics, either.

5 Perspective and Future Work

Since its introduction [33,35], RMC has played an important role in the development of verification techniques for infinite-state systems.

In addition to the basic techniques we describe in this tutorial, the framework has been developed in many directions [4]. We mention some of these extensions in this paragraph. A *broadcast* transition is initiated by a process, called the

initiator. Together with the initiator, an arbitrary number of processes change state simultaneously. In *binary* communication two processes perform a rendezvous changing state simultaneously.

We have also considered parameterized systems where the individual processes operate on numerical variables over the natural numbers [5]. The conditions on the numerical variables are stated as *gap-order constraints*: a logical formalism which can express simple relations such as lower and upper bounds on the values of individual variables; and equality, and gaps (minimal differences) between values of pairs of variables.

Furthermore, we have studied abstraction techniques that approximate the set of *forward-reachable* configurations [7] (rather than the set of backward-reachable configurations as was the case with monotonic abstraction). The framework is based on establishing a *cut-off theorem*. More precisely, it needs to inspect only a small number of processes in order to show correctness of the whole system. It relies on an abstraction function that views the system from the perspective of a fixed number of processes. The abstraction is used during the verification procedure in order to dynamically detect cut-off points beyond which the search of the state space need not continue.

Interesting directions for future work include:

- Parameterized timed systems [6].
- Applying symbolic partial order techniques [9] to increase efficiency.
- Applying RMC to concurrent programs that operate on weak consistency models such as the release-acquire semantics [1].
- Refining the granularity of quantified transitions [8]

Acknowledgement. Bengt Jonsson introduced me to the world of research in computer science. Since those early days, he has been my colleague, friend, and mentor. He was a leader and influential in developing the frameworks of regular model checking and well-structured systems. Many thanks, Bengt, for your support and for being an inspiration throughout the years.

References

1. Abdulla, P.A., Arora, J., Atig, M.F., Krishna, S.N.: Verification of programs under the release-acquire semantics. In: McKinley, K.S., Fisher, K. (Eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, 22–26 June 2019, pp. 1117–1132. ACM (2019)
2. Abdulla, P.A., Bouajjani, A., Jonsson, B.: On-the-fly analysis of systems with unbounded, lossy FIFO channels. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification, CAV 1998. LNCS, vol. 1427, pp. 305–318. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028754
3. Abdulla, P.A., Čerāns, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: Proceedings of the LICS 1996 11^{th} IEEE International Symposium on Logic in Computer Science, pp. 313–321 (1996)

4. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71209-1_56
5. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized verification of infinite-state processes with global conditions. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73368-3_17
6. Abdulla, P.A., Deneux, J., Mahata, P.: Multi-clock timed networks. In: 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14–17 July 2004, Turku, Finland, Proceedings, pp. 345–354. IEEE Computer Society (2004)
7. Abdulla, P.A., Haziza, F., Holík, L.: Parameterized verification through view abstraction. Int. J. Softw. Tools Technol. Transfer **18**(5), 495–516 (2015). https://doi.org/10.1007/s10009-015-0406-x
8. Abdulla, P.A., Ben Henda, N., Delzanno, G., Rezine, A.: Handling parameterized systems with non-atomic global conditions. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) Verification, Model Checking, and Abstract Interpretation, VMCAI 2008. LNCS, vol. 4905, pp. 22–36. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78163-9_7
9. Abdulla, P.A., Jonsson, B., Kindahl, M., Peled, D.: A general approach to partial order reductions in symbolic verification. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification, CAV 1998. LNCS, vol. 1427, pp. 379–390. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028760
10. Abdulla, P.A., Jonsson, B., Mahata, P., d'Orso, J.: Regular tree model checking. In: Brinksma, Ed., Larsen, K.G. (eds.) Computer Aided Verification, CAV 2002. LNCS, vol. 2404, pp. 555–568. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45657-0_47
11. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Regular model checking made simple and effcient*. In: Brim, L., Křetínský, M., Kučera, A., Jančar, P. (eds.) CONCUR 2002—Concurrency Theory, CONCUR 2002. LNCS, vol. 2421, pp. 116–131. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45694-5_9
12. Abdulla, P.A., Jonsson, B., Nilsson, M., d'Orso, J.: Algorithmic improvements in regular model checking. In: Hunt, W.A., Somenzi, F. (eds.) Computer Aided Verification, CAV 2003. LNCS, vol. 2725, pp. 236–248. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45069-6_25
13. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking for real-time systems. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS 1990), Philadelphia, Pennsylvania, USA, 4–7 June 1990, pp. 414–425. IEEE Computer Society (1990)
14. Basin, D.A., Klarlund, N.: Automata based symbolic reasoning in hardware verification. Formal Methods Syst. Des. **13**(3), 255–288 (1998)
15. Boigelot, B., Godefroid, P.: Symbolic verification of communication protocols with infinite state spaces using QDDs. In: Alur, R., Henzinger, T.A. (eds.) Computer Aided Verification, CAV 1996. LNCS, vol. 1102, pp. 1–12. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61474-5_53
16. Boigelot, B., Godefroid, P., Willems, B., Wolper, P.: The power of QDDs (extended abstract). In: Van Hentenryck, P. (ed.) Static Analysis, SAS 1997. LNCS, vol. 1302, pp. 172–186. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0032741

17. Boigelot, B., Wolper, P.: Symbolic verification with periodic sets. In: Dill, D.L. (ed.) Computer Aided Verification, CAV 1994. LNCS, vol. 818, pp. 55–67. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58179-0_43
18. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model checking. In: Proceedings of the International Conference on Concurrency Theory (CONCUR 1997). LNCS 1243 (1997)
19. Bouajjani, A., Habermehl, P.: Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) Automata, Languages and Programming, ICALP 1997. LNCS, vol. 1256, pp. 560–570. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63165-8_211
20. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification, CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_31
21. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98**, 142–170 (1992)
22. Burkart, O., Steffen, B.: Model checking for context-free processes. In: Cleaveland, W.R. (ed.) CONCUR 1992, CONCUR 1992. LNCS, vol. 630, pp. 123–137. Springer, Heidelberg (1992). https://doi.org/10.1007/BFb0084787
23. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. Theor. Comput. Sci. **221**(1–2), 251–270 (1999)
24. Caucal, D.: On the regular structure of prefix rewriting. Theoret. Comput. Sci. **106**(1), 61–86 (1992)
25. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specification. ACM Trans. Program. Lang. Syst. **8**(2), 244–263 (1986)
26. Comon, H., Jurski, Y.: Multiple counters automata, safety analysis and presburger arithmetic. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification, CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028751
27. Dams, D., Lakhnech, Y., Steffen, M.: Iterating transducers. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification, vol. 2102. Lecture Notes in Computer Science (2001)
28. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: Berry, G., Comon, H., Finkel, A. (eds.) Computer Aided Verification, CAV 2001. LNCS, vol. 2102, pp. 324–336. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44585-4_30
29. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere. Tech. Rep. LSV-98-4, Ecole Normale Supérieure de Cachan (1998)
30. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems (extended abstract). In: Proceedings of the Infinity 1997, Electronic Notes in Theoretical Computer Science, Bologna, August 1997
31. Henriksen, J.G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Proceedings of the TACAS 1995, 1^{th} International Confererence on Tools and Algorithms for the Construction and Analysis of Systems, vol. 1019, Lecture Notes in Computer Science (1996)

32. Jonsson, B., Nilsson, M.: Transitive closures of regular relations for verifying infinite-state systems. In: Graf, S., Schwartzbach, M. (eds.) Proceedings of the TACAS 1900, 6^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems, vol. 1785, Lecture Notes in Computer Science (2000)
33. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. Theoret. Comput. Sci. **256**, 93–112 (2001)
34. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) International Symposium on Programming, Programming 1982. LNCS, vol. 137, pp. 337–351. Springer, Heidelberg (1982). https://doi.org/10.1007/3-540-11494-7_22
35. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: Hu, A.J., Vardi, M.Y. (eds.) Computer Aided Verification, CAV 1998. LNCS, vol. 1427, pp. 88–97. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0028736