Explicit State Model Checking with Generalized Büchi and Rabin Automata

Vincent Bloemen
University of Twente
Enschede, The Netherlands
v.bloemen@utwente.nl

Alexandre Duret-Lutz
LRDE, EPITA
Kremlin-Bicêtre, France
adl@lrde.epita.fr

Jaco van de Pol
University of Twente
Enschede, The Netherlands
j.c.vandepol@utwente.nl

ABSTRACT
In the automata theoretic approach to explicit state LTL model checking, the synchronized product of the model and an automaton that represents the negated formula is checked for emptiness. In practice, a (transition-based generalized) Büchi automaton (TGBA) is used for this procedure.

This paper investigates whether using a more general form of acceptance, namely transition-based generalized Rabin automata (TGRAs), improves the model checking procedure. TGRAs have significantly fewer states than TGBAs, however the corresponding emptiness checking procedure is more involved. With recent advances in probabilistic model checking and LTL to TGRA translators, it is only natural to ask whether checking a TGRA directly is more advantageous in practice.

We designed a multi-core TGRA checking algorithm and performed experiments on a subset of the models and formulas from the 2015 Model Checking Contest. We observed that our algorithm can be used to replace a TGBA checking algorithm without losing performance. In general, we found little to no improvement by checking TGRAs directly.

CCS CONCEPTS
• Theory of computation → Automata over infinite objects;
  • Mathematics of computing → Automata over infinite objects;
  • Mathematics of computing → Graph algorithms;

KEYWORDS
Model checking, Explicit state, LTL, ω-automata, on-the-fly, Generalized Büchi, Rabin, Multi-core, Parallel

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

1 INTRODUCTION
Model checking. In the automata theoretic approach to LTL model checking, the synchronized product of the negated property and the state-space of the system is combined. The resulting product is checked for emptiness by searching for an accepting cycle, i.e., a reachable cycle that satisfies the accepting condition [28]. The emptiness checking procedure is limited by the well-known state-space explosion problem, where the product automaton becomes too large to handle.

On-the-fly model checking mitigates the state-space memory constraints by only storing the states (not the transitions) encountered during the emptiness check. The search procedure is launched from an initial state. Reachables states are computed via a successor function, and in case a counterexample is detected the search may end well before the entire state-space is explored. A consequence is that in practice emptiness checks rely on depth-first search (DFS) exploration [27].

Another way to reduce the size of the product automaton is to keep the sizes of the system’s state-space and the negated property automaton as small as possible. In particular, smaller property automata can be obtained by using more complex acceptance conditions.

With current hardware systems one can further improve the model checking performance by utilizing multiple cores. This way, the time to model check can be significantly reduced; related work shows that even though the problem is difficult to parallelize, in practice an almost linear improvement with respect to the number of cores can be obtained [6, 14, 17, 26].

Our goal: emptiness checks using generalized Rabin automata. The automata-theoretic approach to LTL model checking is often performed using Büchi automata (BAs), or even transition-based generalized Büchi automata (TGBAs). TGBAs can be linearly more concise than BAs, resulting in smaller products, and can be emptiness checked using algorithm enumerating strongly-connected components (SCCs) at no extra cost compared to SCC-based algorithms on BAs [8].

For probabilistic model checking, working with deterministic automata is important, as otherwise the resulting product automaton might not be a Markov chain [3]. Since it is well known that not all BAs can be determinized, probabilistic model checkers use Rabin automata (RAs) instead. Recently, order-of-magnitude speedups were reported when performing probabilistic model checking using a generalized acceptance condition called transition-based generalized Rabin Automata (TGRAs) [7]. Also, there has been a lot of interest into building tools such as LTLMODA [2] and Rabinizer 3 [13, 19] for translating LTL formulas into small deterministic TGRAs.
Our objective is to study whether the speedups observed with TGRAs in probabilistic model checking also hold for non-probabilistic explicit model checking. There are plenty of algorithms for checking BAs and TGBAs (both sequentially and multi-core) [6, 14, 26, 27], however for Rabin acceptance there is only a recent work on a GPU algorithm for checking (non-generalized) RAs [29] and a TGRA checking algorithm for probabilistic model checking [7].

None of these works address our question: is there any advantage to using transition-based Generalized Rabin Automata (TGRAs) over transition-based Generalized Büchi Automata (TGBAs)? To do so, we introduce a multi-core emptiness-check procedure for TGRAs. We implement it in LTSmin [18], and benchmark several model-checking tasks realized using TGRAs or TGBAs. Note that in our case, the determinism of the automaton is not important.

We should also point out that having an efficient emptiness check for TGRAs has more applications than just model checking, because generalized Rabin acceptance can be thought of as a normal form for any acceptance condition. Such a TGRA emptiness check could therefore be useful to ω-automata libraries such as Spot [10] that work with automata using arbitrary acceptance conditions [1]. Currently, ω-automata are converted into TGBAs before being emptiness-checked. A recent tool is LTL3HOA, which produces automata with an arbitrary acceptance condition.

Overview. The remainder of the paper is structured as follows. We provide preliminaries in Section 2 and present our algorithm in Section 3. We discuss related work in Section 4. Implementation details and experiments are discussed in Section 5 and we conclude in Section 6.

2 PRELIMINARIES

We define ω-automata using acceptance conditions that are positive Boolean formulas over terms like Fin(T) (the transitions in T should be seen finitely often) or Inf(T) (infinitely often). This convention, inspired from the HOA format [1] allows us to express all traditional acceptance conditions, and is similar to the formalism used by Emerson & Lei 30 years ago [12] using state-based acceptance.

Definition 2.1 (TELA). A transition-based Emerson-Lei automaton (TELA) is a tuple \( \mathcal{A} = (\Sigma, q_0, \delta, \text{Acc}) \) where \( \Sigma \) is an alphabet, \( Q \) is a finite set of states, \( q_0 \in Q \) is the initial state, \( \delta \subseteq Q \times \Sigma \times Q \) is a transition relation, \( \text{Acc} \) is a positive Boolean function over terms of the form \( \text{Fin}(T) \) or \( \text{Inf}(T) \) for any subset \( T \subseteq \delta \). For a transition \( t \in \delta \) we note \( t^\ell \) its source, \( t^d \) its label, and \( t^d \) its destination: \( t = (t^s, t^d, t^d) \).

Runs of \( \mathcal{A} \) are infinite sequences of consecutive transitions:

\[
\text{Runs}(\mathcal{A}) = \{ \rho \in \delta^\omega \mid \rho(0)^n = q_0 \land \forall i \geq 0 : \rho(i)^d = \rho(i + 1)^s \}
\]

The acceptance of a run \( \rho \) is defined by evaluating the acceptance condition \( \text{Acc} \) over \( \rho \) such that:

- \( \text{Fin}(T) \) is true iif all the transitions in \( T \) occur finitely often in \( \rho \).
- \( \text{Inf}(T) \) is true iif some transitions in \( T \) occurs infinitely often in \( \rho \).

}\[ \text{inf}(\mathcal{A}_1) \text{ a deterministic transition-based generalized Büchi automaton recognizing } \text{GF}a \land \text{GF}b, (\mathcal{A}_2) \text{ a non-deterministic transition-based Büchi automaton recognizing } \text{FG}a. (\mathcal{A}_3) \text{ a deterministic transition-based co-Büchi automaton recognizing } \text{FG}a. \]

Let \( \rho^f \in \Sigma^\omega \) be the word recognized by a run \( \rho \in \text{Runs}(\mathcal{A}) \) defined by \( \rho^f(i) = \rho(i)^d \) for all \( i \geq 0 \). The language of \( \mathcal{A} \) is the set of all words \( \rho^f \) recognized by some accepting run \( \rho \).

A Transition-based Generalized Büchi Automaton (TGBA) is a TELA where \( \text{Acc} = \text{Inf}(T_1) \land \text{Inf}(T_2) \land \ldots \land \text{Inf}(T_n) \) for some \( n \), meaning that any accepting run has to visit infinitely often one transition from each set \( T_i \).

As an example, automaton \( \mathcal{A}_1 \) from Figure 1 represents a TGBA for the formula \( \text{GF}a \land \text{GF}b \). Here, transitions are labeled by all possible assignments of \( a \) and \( b \), i.e., elements of \( \Sigma = \{ \bar{a}, a, \bar{b}, b, \bar{ab}, a\bar{b}, ab \} \) (\( \bar{a} \) denotes the negation of \( a \)), and transitions are also marked using \( \bullet \) and \( \circ \) to denote their membership to the sets used in the acceptance condition. A run of \( \mathcal{A}_1 \) is accepted if it visits the two acceptance marks \( \bullet \) and \( \circ \) infinitely often.

The two automata \( \mathcal{A}_2 \) and \( \mathcal{A}_3 \) from Figure 1 represent the formula \( \text{FG}a \) using the alphabet \( \Sigma = \{ \bar{a}, a \} \) and different acceptance conditions: \( \mathcal{A}_2 \) is a Transition-based Büchi Automaton while \( \mathcal{A}_3 \) is a Transition-based co-Büchi Automaton. Both automata are minimal in their number of states and illustrate that allowing Fin acceptance can reduce the size of the automaton. Moreover, \( \mathcal{A}_3 \) is a deterministic automaton, whereas no equivalent deterministic BA exists.

A Transition-based Generalized Rabin Automaton (TGRA) is a TELA where \( \text{Acc} \) has the form \( \bigwedge_{i=1}^n (\text{Fin}(F_i) \land \text{Inf}(l^1_i) \land \text{Inf}(l^2_i) \land \ldots \land \text{Inf}(l^{t_i}_i)) \) for some values of \( n \) and \( t_1, t_2, \ldots, t_n \). This is a generalization of Rabin acceptance in the sense that in Rabin acceptance \( t_i = 1 \) for all \( i \). Each conjunctive clause of the form \( \text{Fin}(F_i) \land \text{Inf}(l^1_i) \land \text{Inf}(l^2_i) \land \ldots \land \text{Inf}(l^{t_i}_i) \) is called a transition-based generalized Rabin pair (TGRP).

Figure 2 depicts a deterministic TGRA (\( \mathcal{A}_4 \)) and a non-deterministic TGBA (\( \mathcal{A}_5 \)), both representing the property \( \text{FG}(Fa \lor Fb) \). \( \mathcal{A}_4 \) is accepting if either \( \bullet \) is visited infinitely often without visiting \( \circ \) infinitely often, or if \( \circ \) is visited finitely often and both \( \bullet \) and \( \circ \) are visited infinitely often. Only one of the two TGRPs has to be satisfied. In this case, by comparing \( \mathcal{A}_4 \) and \( \mathcal{A}_5 \) we can (again) observe that Fin acceptance aids in both reducing the size of the automaton and causing the automata to be deterministic.

Since generalized Rabin acceptance is just a disjunction of TGRPs, it can serve as a normal form for any acceptance condition. Any acceptance condition can be converted into generalized Rabin acceptance by distributing \( \land \) over \( \lor \) to obtain a disjunctive normal form, and then replacing any conjunctive clause of the form \( \text{Fin}(F^1) \land \text{Fin}(F^2) \land \ldots \land \text{Fin}(F^n) \land \text{Inf}(l^1) \land \text{Inf}(l^2) \land \ldots \land \text{Inf}(l^{t_i}) \) by the TGRP \( \text{Fin}(\bigcup_{i=1}^n F^i) \land \text{Inf}(l^1) \land \text{Inf}(l^2) \land \ldots \land \text{Inf}(l^{t_i}) \).
conversion can be done without changing the transition structure of the automaton; its only downside is that it may introduce an exponential number of TGRPs.

Strongly-connected components (SCCs) are usually defined as maximal with respect to inclusion, but this extra constraint is not always desirable in an emptiness check, where we are just looking for one accepting cycle. We therefore use the terms partial SCC and maximal SCC when we need to be specific.

**Definition 2.2 (SCC).** Given a TELA of the form $\mathcal{A} = (\Sigma, Q, q_0, \delta, \text{Acc})$, a partial Strongly Connected Component (partial SCC) is a pair $C = (C_Q, C_\delta) \in 2^Q \times 2^\delta$ with $C_Q \neq \emptyset$ such that any ordered pair of states of $C_Q$ can be connected by a sequence of consecutive transitions from $C_\delta$. We say that $C$ is a maximal SCC if $C$ is maximal with respect to inclusion, thus the case where both $C_Q$ and $C_\delta$ cannot be extended. An SCC is called trivial if $C_\delta = \emptyset$, and hence $C_Q$ consists of a single state.

In a TGBA with $n$ acceptance sets of the form $\text{Inf}(T_1) \land \ldots \land \text{Inf}(T_n)$, finding an accepting run boils down to searching for a trace from the initial state to a reachable partial SCC $C$ for which $\forall 1 \leq i \leq n : T_i \cap C_\delta \neq \emptyset$ holds, i.e., a partial SCC that intersects each acceptance set.

In a TGRA, an accepting run has to satisfy one TGRP. We say that a TGRP $\text{Fin}(\mathbb{F}) \land \text{Inf}(I^1) \land \text{Inf}(I^2) \land \ldots \land \text{Inf}(I^n)$ has an accepting run if there is a trace from the initial state to a reachable partial SCC $C$ with $\forall 1 \leq i \leq n : I_i \cap C_\delta \neq \emptyset$ and $F \cap C_\delta = \emptyset$, i.e., a partial SCC that contains a transition from every Inf set and no transition from the Fin set.

Note that in the case of a TGBA, it is always valid to replace the search of a partial SCC intersecting all acceptance sets by the search of a maximal SCC intersecting these sets. However this cannot be done when the acceptance condition uses Fin sets. For instance consider the automaton $\mathcal{A}_4$ in Figure 2 checked against the TGRP $\text{Fin}(\mathbb{O}) \land \text{Inf}(\mathbb{O})$: the automaton is a unique maximal SCC that does not satisfy $\emptyset \cap C_\delta \neq \emptyset$ and $\emptyset \cap C_\delta = \emptyset$. However, those constraints hold on the partial SCC that consists of state 0 and the loop above it. For this reason our algorithm will build partial SCCs that do not include transitions labeled by Fin sets.

### 3.1 Checking each Rabin pair separately

Checking TGRAs can be achieved by checking each Rabin pair separately, as shown in Algorithm 1. In case an accepting cycle is found by TGRPAcc, that sub-procedure should report Acc and exit. Thus, in case none of the TGRPAcc sub-procedures report acceptance, the algorithm returns with No_Acc. We assume that prior to each TGRPAcc call, we have no knowledge on the individual TGRPs and therefore treat them equally and separately. In theory, this assumption may lead to missed opportunities. For example, consider the case where TGRP$_1 = $ TGRP$_2$, or even an overlap in the Fin and/or Inf fragments of the TGRPs could be a reason for combining gained information.

**Algorithm 1:** Checking TGRA by checking each TGRP

```plaintext
function TGRACheck(Q, q, TGRA = (TGRP$_1$, ..., TGRP$_n$))
for all $i \in \{1, \ldots, n\}$
   TGRPAcc(Q, q, TGRP$_i$)
return No_Acc // No TGRPAcc call reported Acc
```

**Parallel TGRA checking.** After a TGRPAcc call has finished, the next Rabin pair is selected and a new sub-procedure is started, until all $n$ pairs have been checked. Since we are working in a multi-core environment, we can assign different worker instances to different Rabin pairs. Suppose there are $P$ workers available, we can choose to either use all $P$ workers for checking a single Rabin pair, or we can distribute the workers over the different pairs. By distributing the workers evenly, for $n$ Rabin pairs, each pair is checked by $\frac{P}{n}$ workers.

A disadvantage of this setup is that each of the $n$ groups of $\frac{P}{n}$ workers processing the same TGRA needs its own copy of the shared data structure. This means that by checking all pairs in parallel, approximately $n$ times more memory is required.

However one advantage of checking each pair in parallel is that the total workload can be better spread out over the available workers, i.e., there is less contention in the data structures since the probability of interfering with different search instances is reduced. Another advantage could be that counterexamples may be detected faster in this setting; suppose for example that only the $n$th pair contains a counterexample, assuming that the counterexample is detected by visiting only part of the state-space, this prevents the complete state-space from being searched $n - 1$ times.

### 3.2 TGRP checking algorithm

Throughout this section we consider checking a TGRP with acceptance of the form $\text{Acc} = (F, I = \{I^1, \ldots, I^n\})$.

**Abstract idea of the algorithm.** The general idea of the algorithm, which we present in Algorithm 2, is to perform an SCC decomposition of the graph without allowing any $F$ transitions from being part...
of the SCCs. As a result, we obtain SCCs that contain all edges except those in \( F \). Formally, we have that each SCC \( C \) is a partial SCC of \( \mathcal{A} \) that is maximal on the TGRP \( \mathcal{A}_{\mathcal{S},F} := (\Sigma, Q, q_0, \delta \setminus F, \text{Acc}) \). \( C \) is an accepting SCC if \( \bigvee_{1 \leq i \leq L} : C_\delta \cap \delta^i \neq \emptyset \), i.e., \( C \) contains transitions such that every \( \delta^i \) can be visited infinitely often. By definition of \( \mathcal{A}_{\mathcal{S},F} \), we have that \( C_\delta \cap \delta = \emptyset \). If \( C \) is also reachable from \( q_{0} \) via transitions from \( \delta \) (including \( F \) transitions), it can be reported that a counterexample exists.

Preventing \( F \) transitions from being considered. The algorithm detects the aforementioned 'constrained' SCCs in linear time and in an on-the-fly setting, without relying on visiting states multiple times\(^2\). It does so by performing a constrained SCC decomposition of \( \mathcal{A} \) from \( q_0 \). Once a transition \( t = (t^1, t^2, t^3) \in F \) is encountered, state \( t^d \) is stored in a so-called \( F \) states set and \( t \) is further disregarded since \( t \) cannot appear in any accepting cycle. Once this search is finished, all states are marked as Dead and all SCCs are decomposed on the subgraph \( \mathcal{A}' \), which is formed by a reachability from \( q_{0} \) over the transitions \( \delta \setminus F \). In case a non-trivial SCC contains transitions from all \( \delta^i \) sets we have detected a counterexample. Otherwise, we pick a state \( s \) from the \( F \) states set and repeat the same procedure (using \( q_0 := s \)). For this state \( s \) there are two cases:

1. \( s \) is marked Dead, meaning that it was added to \( F \) states but it was also reachable in \( \mathcal{A}' \) (without taking any \( F \) transitions). Thus, we have already explored this state and can ignore it.
2. \( s \) is not marked Dead, meaning that \( s \) is not part of \( \mathcal{A}' \) and we launch a new SCC decomposition from \( s \).

The search procedure is illustrated in Figure 3. Here, two TGRPs are checked separately. Note that due to the way how \( \bullet \) and \( \otimes \) are located in the automaton, the initial searches for the first and second pair lead to different components. The search for pair 1 detects \( \mathcal{A}'_1 = C_1 \cup C_3 \) (avoiding \( \bullet \)) and the search for pair 2 detects \( \mathcal{A}'_2 = C_1 \cup C_2 \) (avoiding \( \otimes \)). Consider the search for pair 1, after the initial search it found the \( F \) states and \( t \). Both have not been explored so suppose that \( t \) is arbitrarily chosen as a 'new' initial state. We assume that the search from \( t \) visits all states in \( C_1 \). If now the edge is found from \( v \) to \( r \), thus from \( C_2 \) to \( C_1 \), we must not allow this transition to form a cycle as it would contain the \( \bullet \) mark. Since the search from \( t \) is initiated after the search from \( q_0 \) is complete, \( r \) is already marked Dead and thus ignored. In fact, even when we allow the search from \( u \) to start before all states in \( C_1 \cup C_3 \) are marked Dead it may in the worst case only add redundant explorations. This is because from the edge from \( s \) to \( u \) is never really considered as an edge in the SCC decomposition and hence no cycle can be formed with a \( \bullet \) mark.

Data structures. To represent the Fin and Inf fragments of a TGRP, we use a set of accepting marks per transition. We assign a unique mark to each \( L \) and \( \delta^i \) set, for \( 1 \leq i \leq L \), and refer to these marks with \( F_M \) and \( I_M^i \) (and \( I_M \) for \( \bigcup_{1 \leq i \leq L} I_M^i \)). The complete

\(^2\)The parallel search is based on swarmed verification, making it unlikely that states are visited only once in practice, but in theory and in a sequential setting this is not necessary for correctness.

\(^3\)Consider for example what happens when there is no path from \( u \) to \( t \). After the search for \( u \) ends, all reachable states from \( u \) are marked Dead and the search from \( t \) is started. Once it observes a Dead state it will not continue searching that state, hence no redundant states are explored.

Figure 3: Example of running an emptiness check on an TGRA with two pairs. \( C_1, C_2, \) and \( C_3 \) represent components (not necessarily strongly connected) that do not contain any transition in the sets \( \bullet \) or \( \otimes \). Workers doing the emptiness check for the first pair will first explore \( C_1 \cup C_3 \), attempting to find a cycle satisfying \( \text{Inf}(\bullet) \lor \text{Inf}(\otimes) \) without crossing the \( \bullet \)-transitions leading to \( u \) and \( t \). If no accepting cycles are found, they will continue their exploration in \( C_2 \), starting in states \( u \) and \( v \), and ignoring all transitions going back to \( C_1 \cup C_3 \). Workers doing the emptiness for the second pair, will similarly first look for cycles satisfying \( \text{Inf}(\bullet) \lor \text{Inf}(\otimes) \) in \( C_1 \cup C_2 \), postponing the exploration of \( C_3 \) that is only accessible via a \( \otimes \)-transition.

set \( M \) is thus defined as \( M := \{ F_M, I_M^0, \cdots, I_M^L \} \). Each transition \( t \) is associated to a set of acceptance marks \( \text{acc} \subseteq M \), indicating whether \( t \in F \) or \( t \in \delta^i \) for \( 1 \leq i \leq L \).

We define \( S \) as a mapping from states to pairs, consisting of a set of states and a set of marks. Thus \( S(s) = (\text{states}, \text{acc}) \) and formally, \( S : Q \rightarrow 2^Q \times 2^M \). By implementing \( S \) with a union-find structure, we can maintain the following invariant at all times:

\[ \forall u, v \in Q : u \in S(v).\text{states} \Leftrightarrow S(v) = S(u) \]

This further implies that every state is part of exactly one set of states. In the algorithm, we use \( S \) to associate each state \( u \) to its partial SCC that contains the states \( S(u).\text{states} \) and visits all the marks in \( S(u).\text{acc} \). \( S \) pairs can be combined using a Unite function. We use an example to illustrate \( S \) and the \( \text{Unite} \) function. Let \( S(u) := \{ \{ u, w \}, \{ F_M \} \} \) and \( S(v) := \{ \{ v \}, \{ F_M, I_M^1 \} \} \); we can use the \( \text{Unite} \) function to combine the two structures. After calling \( \text{Unite}(S, u, v) \) we have \( S(u) = S(v) = \{ \{ u, v, w \}, \{ F_M, I_M^1 \} \} \), while keeping all other mappings the same. For more details on this structure, we refer to Bloemen et al. \[5\]. We use an additional function AddAcc to 'add' (the union of) acceptance marks to the set, thus \( \text{AddAcc}(S, t, \{ I_M^1, I_M^2 \}) \) will ensure that \( S(v).\text{acc} \) becomes \( \{ F_M, I_M^1, I_M^2 \} \).

The \( F \) states structure is implemented as a cyclic list that contains all states added to the list (by means of \( F \) states. addState). \( F \) states. pickState returns a state from the list, in case the list is nonempty. Finally, states are removed from the list by calling \( F \) states. removeState. For efficient list containment and to avoid duplicated states from being added, we store the list on top of an array, in which the elements point to each other.
Algorithm 2: Algorithm for checking a TGRP

1. function TGRPAcc \((Q, q_0, \text{TGRP} = (F_M, I_M))\)
2. \[\forall s \in Q : S(s) := ([s], \emptyset) \] // Initialize map
3. \(\text{Visited} := \text{Dead} := \emptyset \) // Sets
4. \(R := \emptyset \) // Roots stack of \((\text{acc}, \text{state})\) pairs
5. \(\text{Fstates} := \{q_0\} \) // Cyclic list of init states
6. while \(\neg\text{Fstates}.isEmpty()\) do
7. \(s := \text{Fstates}.\text{pickState}()\)
8. if \(s \notin \text{Dead}\) then TGRPAccRecur\((\emptyset, s)\)
9. \(\text{Fstates}.\text{removeState}(s)\)
10. return // No Acc got reported in the search

11. function TGRPAccRecur \((\text{acc}, s)\)
12. \(\text{Visited} := \text{Visited} \cup \{s\}\)
13. \(R.\text{push}(\text{acc}, s)\)
14. for all \((\text{acc}, t) \in \text{succ}(s)\) do
15. if \(t \in \text{Dead}\) then continue // Explored
16. else if \(t \notin \text{Visited}\) then // ‘New’ state
17. if \(\text{acc} \cap F_M \neq \emptyset\) then // Avoid \(F\)
18. \(\text{Fstates}.\text{addState}(t)\)
19. else TGRPAccRecur\((\text{acc}, t)\)
20. else if \(\text{acc} \cap F_M = \emptyset\) then // Cycle
21. while \(S(s) \neq S(t)\) do
22. \((\text{acc}_r, r) := R.\text{pop}()\)
23. \(\text{Unite}(S(s), r.\text{top}().\text{state})\)
24. \(\text{AddAcc}(S(s), r, \text{acc}_r)\)
25. \(\text{AddAcc}(S(s), \text{acc}_r) \) // Add \text{acc}_r to \(S(s)\)
26. if \(I_M = S(s).\text{acc} \) then // Acc. cycle
27. \(\text{report Acc and exit}\)
28. if \(s = R.\text{top}()\) then // Completed SCC
29. \(\text{Dead} := \text{Dead} \cup S(s).\text{states}\)
30. \(R.\text{pop}()\)

Detailed algorithm. The sequential algorithm for checking a TGRP is presented in Algorithm 2. First, all data structures are initialized in lines 2-5. Then, the algorithm continuously picks a state \(s\) (initially \(q_0\)) and calls the TGRPAccRecur procedure. This procedure performs an SCC decomposition, similar to Dijkstra’s algorithm [9]. After the TGRPAccRecur is finished, \(s\) is removed from the Fstates list and a new state is picked from the list. If the list is empty, we assume that the complete state-space has been visited and since no Acc was reported, we can conclude that no counterexample exists for this TGRP.

In the TGRPAccRecur procedure, state \(s\) is marked as visited and pushed on top of the \(R\) stack, along with the accompanying acceptance set \(\text{acc}_r\) (note that since there is no transition to the initial state, the empty set is given in line 8). The \(R\) stack can be regarded as an extension to the roots stack from Dijkstra’s SCC algorithm [9]. All successors of \(s\) are considered in lines 14-27. For each successor \(t\) we consider three cases:

- \(t \in \text{Dead}\) (line 15), this implies that \(t\) has already been completely explored and can thus be disregarded.
- \(t\) is unvisited (lines 16-19), meaning that \(t\) has not been encountered yet. If \(t\) is part of the Fin set, we add it to the Fstates list and ignore it for the current search. Otherwise, we recursively search \(t\).
- \(t\) is not Dead but it has been visited before (lines 20-27).

This implies that there is some state \(r^\prime\) on the \(R\) stack such that \(t \in S(r^\prime).\text{states}\) and hence a cycle can be formed. The algorithm then continuously takes the top two states from the \(R\) stack and unites them (and adds the acceptance mark) until \(S(s)\) and \(R(t)\) are the same. Finally, the acceptance marks from \(\text{acc}_r\) are added. At line 26, \(S(s)\) contains all states in the cycle from \(s\) to \(t\) and forms a partial SCC. \(S(s)\) is then checked if it contains all Inf acceptance marks. If so, an accepting SCC is found and is reported.

After all successors are explored, the algorithm backtracks. In case \(s\) equals the top of the \(R\) stack (line 28), \(s\) is the last state of the SCC and the entire SCC is marked as being fully explored by marking it as Dead.

Outline of correctness. We argue that the TGRPAcc algorithm decomposes the TGRP automaton in maximal SCCs when defined over the transitions \(\delta \setminus F\) and that it correctly reports accepting cycles; it reports Acc when a reachable SCC contains a transition from each \(I^\prime\) sets, for \(1 \leq i \leq \ell\), and no transition from \(F\). Due to the conditions of line 17 and 20, for a transition with \(\text{acc}_r \cap F = \emptyset\) it is not possible to start a recursive call with \(\text{acc}_r\) (thus \(\text{acc}_r\) never appears on the \(R\) stack) nor is it possible to call AddAcc with \(\text{acc}_r\) as an argument. All such transitions are ‘avoided’ and unvisited successors are added to Fstates. We thus conclude that no \(F\) transition can be contained in any formed SCC.

Because we do allow and explore all other (non-\(F\)) transitions during the search, assuming a correct SCC algorithm, the acceptance set of each SCC cannot be further extended without also having to include an \(F\) transition.

Since all states that did not get visited were added to the Fstates list, and each state from this list is eventually picked as an initial state, we argue that the complete state-space has been explored after the algorithm terminates on line 10.

Complexity. One can observe that every state and transition is visited at most once in the algorithm. The TGRPAccRecur procedure will mark a state as visited and will never be called twice for the same state. The bottleneck of the algorithm becomes maintaining the \(S\) structure. From previous work [5] we know that the union-find structure (without tracking acceptance marks) causes the complete algorithm to operate in quasi-linear time. By assuming that \(|M| = 1 + \ell\) is a small constant (which can be assumed in practice), tracking the acceptance can be achieved in constant time per modification to the structure, hence the total time complexity is upper bounded by \(O(|M| \cdot |\delta| \cdot \log(|\delta|))\) for each TGRP.

The space complexity is limited by the sizes of the \(R, S,\) and Fstates structures. \(R\) may contain up to \(|Q|\) states and acceptance marks in the worst case (by visiting every state in a single path). \(S\) can be implemented as an array of length \(|Q|\) of structs that are of constant size, plus \(|M|\) bits for tracking acceptance, and Fstates can be implemented as an array of \(|Q|\) elements. In total \(O(|Q| \cdot |M|)\) memory is used.
Parallel implementation. Algorithm 2 can be parallelized by *swarming* the search instances; by starting multiple worker instances from the initial state and using a randomized successor function to steer the workers towards different parts of the state-space. The TGRPACCReCUR function can be seen as an extension to the multi-core SCC algorithm from Bloemen et al. [5, 6]. The key to this algorithm is to globally communicate locally detected cycles. This way, multiple workers can cooperatively decompose SCCs. Additionally, (partly) unexplored states in an SCC are tracked globally and once a worker fully explores a state, none of the other workers have to explore this state again. Once all states of an SCC are fully explored, the entire SCC must be fully explored and thus can be marked Dead.

During UniTe procedures, the involved parts of the union-find structure are briefly locked to guarantee correctness. During this locking phase, the acceptance set can be updated atomically without interfering with other parts. This is also implemented in our existing TGBA checking algorithm [6].

The $F$ states list is implemented by using a fine-grained locking mechanism to add states to the list, such that all states remain on the cycle. The reason for implementing $F$ states as a cyclic list becomes clear in the next example. Suppose the $F$ states list contains two states, $u$ and $v$. To avoid contention, the algorithm attempts to divide the workload by assigning half of the workers to search from $u$ and the other half to search from $v$. Now, assume that $u$ does not have any successors and a large part of the state-space is reached from $v$. If the search from $u$ completes, we ideally want to let the workers aid in the search from $v$. By maintaining $F$ states as a cyclic list, without much effort we can check which searches have not been completed yet. The $F$ states list is implemented similarly as the cyclic list in the union-find structure, which is discussed in [5].

The time complexity of the algorithm is in the worst case increased by a factor $P$, for $P$ workers, since the algorithm tracks a bit per worker instance in the union-find structure. However, in practice we observe an improvement over the sequential implementation. For the same reason the memory complexity is also increased by $P$, and additionally every worker contains its own $R$ stack. Moreover, as mentioned in Section 3.1, if all TGRPs are checked simultaneously, a copy of the global data structures has to be made for each group of worker processing a different TGRP. As a result, $n$ times more memory is required for these structures in case there are $n$ TGRPs.

4 RELATED WORK

Related work on checking Büchi automata. Explicit state on-the-fly algorithms for checking can be distinguished in two classes, namely Nested Depth-First Search (NDFS) and SCC-based algorithms. Schwoon and Esparza provide a great overview on these techniques [27]. The advantage of SCC-based algorithms over NDFS is that they can handle generalized Büchi automata efficiently.

In a multi-core setting, we consider the CNDFS algorithm [14] to be the state-of-the-art NDFS-like algorithm. It is based on swarm verification [16] and operates by spawning multiple NDFS instances and globally communicating ‘completed’ parts of the state-space.

For state-of-the-art multi-core SCC-based algorithms, in prior work we showed that the algorithm from Bloemen et al. [6] outperforms other techniques and performs similar to the CNDFS algorithm. The algorithm is also based on swarmed searches, and detected partial SCCs are communicated globally and maintained in a shared structure. Notable related multi-core SCC algorithms are the ones from Renault et al. [26] and Lowe [23].

Related work on checking Rabin automata. As mentioned in Section 1, when checking LTL properties for probabilistic systems, the automaton needs to be deterministic [3]. Chatterjee et al. [7] present an algorithm to check deterministic TGRA conditions in the context of (offline) probabilistic model checking. The idea is to consider each generalized Rabin pair $(F_i, (I^1_i, \ldots, I^l_i))$ separately and for each pair: (1) remove the set of states $F_i$ from the state space, (2) compute the maximal end-component (MEC) decomposition, and (3) check which MECs have a non-empty intersection with every $I^j_i$, for $j = 1, \ldots, l_i$. These sets are then used for computing maximal probabilities. The paper reports significant improvements over checking a degeneralized variant of deterministic TGTRAs. They also present improvements for computing a winning strategy in LTL($F, G$) games by using a fixpoint algorithm for generalized Rabin pairs. Our algorithm is different in that it operates on-the-fly and in a multi-core setting.

Wijs [29] recently presented a on-the-fly GPU algorithm for checking LTL properties for non-generalized deterministic Rabin automata. Here, the choice for (deterministic) Rabin automata, instead of non-deterministic Büchi automata, is motivated by the observations that it can speed up the successor construction and that it can reduce the state space of the cross-product. In that paper, a BFS-based search is used, in particular a variation on the heuristic piggybacking search [15, 17]. This approach is incomplete due to situations referred to as shadowing and blocking, but these cases can be detected and resolved with a depth-bounded DFS. Our approach differs in that we allow (generalized) TGRAs and do not require repair procedures.

Related work on checking different automata. Emerson and Lei [12] show that the emptiness check of an $\omega$-automaton with arbitrary acceptance condition is NP-complete. They also present a polynomial algorithm for the case where the acceptance condition is provided as a disjunction of Streett acceptance conditions. Streett acceptance is the negation of Rabin acceptance, a conjunction of Fin($l$) $\lor$ Inf($F$) instances (or equivalent, Inf($l$) $\Rightarrow$ Inf($F$)), and Streett acceptance closely relates to fairness checking.

Duret-Lutz et al. [11] present a sequential algorithm for checking Streett objectives by performing an SCC decomposition and tracking thresholds to prevent ‘rejecting’ cycles from occurring in the SCCs. In a multi-core setting, the algorithm by Liu et al. [22] performs an initial SCC decomposition and for every SCC a new instance is launched in parallel that ignores certain transitions.

5 EXPERIMENTS

5.1 Experimental Setup

All experiments were performed on a machine with 4 AMD Opteron™ 6376 processors, each with 16 cores, forming a total of
We used the algorithm from Bloemen et al. [6] to model check \( \text{Fin} \) with at least one non-empty (116 in total) for which the time to model check using the TGBA \( \text{LTSmin} \) translators, and used the algorithm presented in this paper for checking total the experiments took approximately 5 days to complete.

Every task was run with a timeout of 10 minutes. In the latter four cases used the TGRA checking algorithm introduced in this paper. Every task was run with a timeout of 10 minutes. In these cases, the TGRA checking algorithm performed 4.6 times faster. The corresponding algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

We used this tool to generate deterministic transition-based generalized Rabin automata.

- **LTL3DRA** v0.2.4 [2]. We used this tool to also generate deterministic transition-based automata with transition-based generalized Rabin acceptance, but LTL3DRA only supports a subset of LTL, called LTL\(_\{\neg \text{GUX}\} \) in [4], which is slightly stricter than the set of LTL formulas where no \text{until} (U) operator may occur in the scope of any \text{always} (G) operator.
- **LTL3HOA** v1.0.1 [24]. We used this tool to generate nondeterministic automata with an arbitrarily complex transition-based acceptance. We then used Spot’s \text{autfilt} \--\text{generalized-rabin} to convert these automata to TGRAs.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

Experiments. We used Spot’s \text{autfilt} \--\text{generalized-rabin} to convert these automata to TGRAs.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.

We used the algorithm from Bloemen et al. [6] to model check TGBAs, and used the algorithm presented in this paper for checking TGRAs, both are implemented in \( \text{LTSmin} \). The algorithms make use of \( \text{LTSmin} \)‘s internal shared hash tables [21], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to \( 2^{32} \) states.
Table 1: Comparison of the geometric mean execution times (in seconds). The numbers between parentheses denote how many times faster the TGBA checking algorithm is compared to the other configuration. We only used the experiments that were checked in all configurations (51 in total, of which 5 counterexamples).

<table>
<thead>
<tr>
<th></th>
<th>LTL3HOA</th>
<th>LTL3DRA</th>
<th>Rabinizer 3</th>
<th>TGBA−TGRA</th>
<th>TGBA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counterexample</td>
<td>4.19 (0.48)</td>
<td>10.06 (1.14)</td>
<td>11.38 (1.29)</td>
<td>8.53 (0.97)</td>
<td>8.80</td>
</tr>
<tr>
<td>No counterexample</td>
<td>10.68 (1.50)</td>
<td>8.30 (1.17)</td>
<td>8.38 (1.18)</td>
<td>7.14 (1.00)</td>
<td>7.12</td>
</tr>
<tr>
<td>Total</td>
<td>9.74 (1.34)</td>
<td>8.46 (1.16)</td>
<td>8.64 (1.19)</td>
<td>7.27 (1.00)</td>
<td>7.27</td>
</tr>
</tbody>
</table>

Figure 4: Time (in seconds) comparisons of the TGBA (x-axis) and the TGRA emptiness checks (y-axis), for various LTL to TGRA translations. Each point represents the time to perform an emptiness check using 16 cores, averaged over 10 runs. The TGRA algorithm performed faster for instances below the \( x=y \) line.

Comparison with LTL3HOA. We consider a comparison with the automata produced by LTL3HOA different from the previous two discussed configurations, since LTL3HOA produces automata with a generic acceptance that are not necessarily deterministic.

The results show instances that perform very poorly compared to the TGBA checking algorithm, but there are also cases, especially counterexamples, that are solved much faster by the TGRA checking algorithm when the LTL3HOA translator is used.

One remarkable instance is the one at (13.2,1.7) in the top-left scatter plot. The corresponding TGBA is a single-state automaton with one acceptance set, and the TGRA is a single-state automaton with two pairs; one pair with a nonempty \( \text{Fin} \) set, and the other pair is equal to the TGBA. The TGRA checking algorithm detects the counterexample while still searching in the first pair (the second pair is never considered), thus the co-Büchi acceptance leads to an almost 8 times improvement. The TGRA algorithm visits on average \( 62 \cdot 10^3 \) unique transitions, while the TGBA version visited \( 110 \cdot 10^6 \) transitions, more than a 1,000\( \times \) difference. For the TGRA, about half of the transitions were part of the \( \text{Fin} \) set.

Cross-validation with TGBA seen as TGRA. Since any TGBA can be trivially rewritten into a TGRA with an empty \( \text{Fin} \) set, we can use this to cross-validate our algorithm. The bottom-right scatter plot of Figure 4 shows the results for this comparison. Aside from
Table 2: Geometric mean sizes of the automata and products. 

<table>
<thead>
<tr>
<th>Aut</th>
<th>Pairs</th>
<th>States</th>
<th>Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTL3HO</td>
<td>1.03</td>
<td>1.41</td>
<td>1.11·10^6</td>
</tr>
<tr>
<td>LTL3DRA</td>
<td>1.02</td>
<td>1.02</td>
<td>0.78·10^6</td>
</tr>
<tr>
<td>Rabinizer 3</td>
<td>1.53</td>
<td>1.03</td>
<td>0.80·10^6</td>
</tr>
<tr>
<td>TGBA-TGRA</td>
<td>1.44</td>
<td>1.00</td>
<td>0.88·10^6</td>
</tr>
<tr>
<td>TGBA</td>
<td>1.44</td>
<td>1.00</td>
<td>0.88·10^6</td>
</tr>
</tbody>
</table>

one memory error (caused by unnecessarily allocating the data structure for Fstates, since there are no Fin sets), there are hardly any differences in the model checking times. It is not too surprising that the results are almost equal, since the TGRA checking algorithm does not have to track any Fstates as there are no Fin transitions in a TGBA. This means that the algorithm reduces to an SCC decomposition that tracks the acceptance marks, which is almost equal to the TGBA checking algorithm that we used to compare with.

We can avoid allocating memory for the Fstates data structure in case there are no Fin sets in the TGRA. Then, this TGRA emptiness check can be used instead of the TGBA emptiness check as there is no reason to keep both algorithms if they perform equally.

5.3 Additional results

Sizes of the automata and products. Table 2 summarizes information on the state-spaces from the experiments that do not contain a counterexample (thus the complete state-space is explored). One can see that the number of states and transitions in the product automata is, on average, smaller for the LTL3DRA and Rabinizer 3 versions compared to TGBA.

We observe that while the Rabinizer 3 TGRAs generally tend to have a larger |Aut| compared to TGBAs, but the product automata are smaller on average, indicating that determinism can help to reduce the state-space of the product automata. Interestingly, while LTL3DRA also produces deterministic automata, it produces both the smallest LTL automata and the smallest product automata.

The product automata from LTL3DRA and Rabinizer 3 are generally smaller than those from TGBAs. If the TGRA checking algorithm would be improved to be (almost) as efficient as a TGBA checking one, there would be no reason to keep using TGBAs instead of TGRAs.

Checking TGRPs in parallel. In a number of cases we observe that the TGRA consists of 2 TGRPs (we have not encountered an instance that contained more than two pairs). In Section 3.1 we suggested that these pairs could be checked in parallel instead of sequentially. We performed experiments to compare the two. In the case for products without counterexamples, there was no observable difference. In case there were counterexamples, the results varied more, but there does not seem to be a clear winner. Because the ‘parallel’ version does allocate significantly more memory (the memory consumption was almost doubled), we prefer checking the TGRPs sequentially. Future work that checks more complicated TGRAs may suggest reasons for choosing the alternative approach.

Scalability. Our existing TGBA checking algorithm [5, 6] achieves good scalability when increasing the number of workers, at least up to 64 cores. Initial experiments for the TGRA checking algorithm showed similar improvements, but the performance improvement starts to drop when increasing beyond 16 cores. The bottleneck of the algorithm is most likely caused by inserting and selecting states from the Fstates list. Future work could investigate whether the Fstates list can be further improved, or point out whether the bottleneck is a structural problem in the algorithm.

6 CONCLUSION

We introduced a multi-core, on-the-fly algorithm for explicit checking of emptiness on TGRAs. We showed that the algorithm is efficient in the sense that every state and transition only has to be visited once and reduces to an SCC decomposition in case there are no Fin sets in the TGRA.

Experiments show that, in general, a TGBA checking algorithm outperforms our new algorithm. This seems to be true in particular for cases where a large proportion of the product state-space is part of a Fin set for the TGRA. In general we conclude that using TGRAs is not advantageous over TGBAs for checking emptiness, when using our algorithms.

Our experiments do suggest that using TGRAs for emptiness checks is advantageous in some scenarios. The product automaton for a TGRA is on average smaller than that of a TGBA, in particular when deterministic LTL to TGRA translators are used (LTL3DRA and Rabinizer 3). The results also suggest, presumably as a consequence, that our algorithm can outperform a TGBA checking algorithm if the successor computation procedure is a costly operation. The TGRA checking algorithm also seems beneficial in instances where only a small fraction of the state-space is part of a Fin set. Finally, the TGRA checking algorithm can be used as a replacement for a TGBA checking one, since the performance on checking TGBAs is practically equal.

Future work includes further improving the TGRA checking algorithm (there are several variations possible), performing additional experiments, and comparing this technique (in different contexts) with related work. Perhaps a preprocessing step could suggest when the algorithm should be applied on a TGRA and when on a TGBA. Another direction for future work is to investigate a variation of the proposed algorithm to check fairness or Streett automata.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work is supported by the 3TU.BSR project.

REFERENCES


