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 (generalized) Büchi automaton (GBA) is constructed and used for this procedure.

This paper investigates whether using a more general form of automaton, a generalized Rabin automaton (GRA), improves the model checking procedure. An advantage of using a GRA is that its number of states can be significantly less than that of a GBA. Unlike a GBA, a GRA can also deterministically represent any LTL formula. However, the corresponding emptiness checking procedure is more involved. With recent advances in probabilistic model checking and LTL to GRA translators, it is only natural to ask whether checking a GRA directly is more advantageous in practice.

We designed a multi-core GRA checking algorithm and performed experiments on a subset of the models and formulas from the 2015 Model Checking Contest. Findings include that our algorithm can be used as a replacement for a GBA checking algorithm without losing performance. We also report advantages and disadvantages of using our algorithm to check TGRAs directly.

Keywords

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

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 [27]. The emptiness checking procedure is limited by the well-known state-space explosion, 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, and in case a counterexample is detected it 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 [26].

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 [17, 14, 25, 6].

Our goal: emptiness checks using generalized Rabin automata.

The automata-theoretic approach to LTL model checking is often performed using Büchi automata (BA), or even transition-based generalized Büchi automata (TGBA). TG-BAs can be linearly more concise than BAs, resulting in smaller products, and can be emptiness checked using SCC-based algorithms 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 have been working with Rabin automata (RA) instead. Recently, order-of-magnitude speedups were reported when performing probabilistic model checking using a generalized acceptance condition called transition-based generalized Rabin Automata (TGRA) [7]. At the same time, there has been a lot of interest into building tools such as LTL3DRA [2] and Rabinizer 3 [19, 13] 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. While there are plenty of algorithms for checking BAs and TG-BAs (both
sequentially and multi-core) [26, 14, 25, 6]. In the case for
Rabin there is only a recent work on a GPU algorithm for
checking (non-generalized) RAs [28] 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 (TGRA) over transition-based Generalized Büchi
Automata (TGBA)? To do so, we introduce a multi-core
emptiness-check procedure for TGRA. We implemented it in
LTSmin [18], and benchmark several model-checking tasks
realized using TGRA or TGBA. Note that in our case, the
determinism of the automaton is not important.

We should also point out that having an efficient empty-
ness check for TGRA has more applications than just model
checking, because the 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 LTL3MBA, which pro-
duces 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. Imple-
mentation 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) (in-
finitely often). This convention, inspired from the HOA for-
matic of Rabin acceptance in the sense that in Rabin
acceptance conditions, one of the two GRPs has to be satisfied. In this case,
both automata are minimal in their number of states and illustrate that allowing Fin
acceptance can reduce the size of the automaton. Moreover, A3 is a deterministic automaton, whereas no equivalent de-
terministic BA exists.

A Transition-based Generalized Büchi Automaton
(TGBA) is a TELA where Acc = Fin(T) ∨ Inf(T) for
some values of n, and i, j, k, . . . . This is a general-
ization of Rabin acceptance in the sense that in Rabin
acceptance conditions, one of the two GRPs has to be satisfied. In this case,
both automata are minimal in their number of states and illustrate that allowing Fin
acceptance can reduce the size of the automaton. Moreover, A3 is a deterministic automaton, whereas no equivalent de-
terministic BA exists.

Since generalized Rabin acceptance is just a disjunction
of GRPs, it can serve as a normal form for any accep-
tance condition. Any acceptance condition can be converted
into generalized Rabin acceptance by distributing ∨ over
∧ to obtain a disjunctive normal form, and then replacing
any conjunctive clause of the form Fin(T) ∧ Fin(T) ∧ . . . ∧ Fin(T)
by the GRP Fin(∪_{i=1}^n Fin(T)) ∧ Fin(T) ∧ . . . ∧ Fin(T). This con-
version can be done without changing the transition structure
of the automaton; its only downside is that it may introduce
an exponential number of GRPs.

Definition 2 (SCC). Given a TELA of the form A =
(Σ, Q, q0, δ, Acc), a partial Strongly Connected Component (partial SCC) in C is a pair C := (C, C) ∈ 2^Q × 2^Q
such that any ordered pair of states of C is connected by
a sequence of consecutive transitions from C. We say

![Figure 1: (A₁) a deterministic transition-based
generalized Büchi automaton recognizing GFa ∨ GFb, (A₂) a non-deterministic transition-based Büchi automaton recognizing FGa, (A₃) a deterministic transition-based co-Büchi automaton recognizing FGa.](attachment:image.png)
that \( C \) is a maximal SCC if \( C \) is maximal with respect to inclusion, thus the case where both \( C_Q \) and \( C_B \) cannot be extended. An SCC is called trivial if \( C_B = \emptyset \), and hence \( C_Q \) consists of a single state.

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

An accepting run on a GRP with \( Acc \) of the form \( \text{Inf}(F) \land \text{Inf}(I^1) \land \text{Inf}(I^2) \land \ldots \land \text{Inf}(I^k) \) is found by searching for a trace from the initial state to a reachable partial SCC \( C \) for which \( \forall 1 \leq i \leq k : I_i \cap C_B \neq \emptyset \) and \( F \cap C_B = \emptyset \) hold, i.e., a partial SCC that contains a transition from every \( \text{Inf} \) set and no transition from the \( Fin \) set.

Note that it is always valid in a TGBA to take \( C_B \) and \( C_Q \) to be maximal with respect to inclusion. In the case for a TGRA this is not always the possible since \( Fin \) transitions must be prevented in \( C_B \). In case a \( Fin \) transition cannot be avoided in \( C_B \), then \( C \) cannot be an accepting SCC.

3. **Algorithm for TGRA Emptiness**

In this section we present an algorithm for checking emptiness on TGRAs. We start by splitting up the TGRAs into individual generalized Rabin pairs, and show how these can be checked efficiently.

### 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 in the GRPAcc, that sub-procedure should report \( Acc \) and exit. Thus, in case none of the GRPAcc sub-procedures report acceptance, the algorithm returns with \( No\ Acc \). We assume that prior to each GRPAcc call, we have no knowledge on the individual GRPs and therefore treat them equally and separately. In theory, this assumption may lead to missed opportunities. For example, consider the case where GRP_1 = GRP_2, or even an overlap in the \( Fin \) and/or \( Inf \) fragments of the GRPs could be a reason for combining gained information.

**Parallel GRA checking.**

After a GRPAcc 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 \( \frac{n}{P} \) workers.

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.

A disadvantage is that the data structures used have to be copied for every Rabin pair search. This means that by checking all pairs in parallel, approximately \( n \) times more memory is required.

### 3.2 GRP checking algorithm

Throughout this section we consider checking a GRP with acceptance of the form \( Acc = (F,I) = (I^1,\ldots,I^k) \).

**Abstract idea of the algorithm.**

The general idea of the algorithm, which we present in Algorithm 2, is to perform an SCC computation on the graph without allowing any \( F \) transitions from being part of the \( SCCs \). As a result, we obtain \( SCCs \) that contain all edges except for the \( F \) transition ones. Formally, we have that each \( SCC \) \( C \) is a partial SCC that is maximal on the GRP \( A_{\delta,F} := (\Sigma,Q,q_0,\delta^F,Acc) \). \( C \) is an accepting SCC if \( \forall 1 \leq i \leq k : I_i \cap \delta \neq \emptyset \), i.e., every \( I^i \) set intersects the transitions of \( C \). By definition of \( A_{\delta,F} \), we have that \( C \cap \delta = \emptyset \). If \( C \) is also reachable from \( q_0 \) via transitions from \( \delta \) (including \( F \) transitions), it can be reported as a counterexample.

**Preventing \( F \) transitions to be considered.**

The algorithm detects the above-mentioned ‘constrained’ SCCs in linear time and in an on-the-fly setting, without relying on visiting states multiple times. It does so by performing a constrained SCC decomposition of \( A \) from \( q_0 \). Once a transition \( t = (t^*,t^d) \in F \) is encountered, state \( t^d \) is stored in a so-called \( Fstates \) set and \( t \) is further disregarded since \( t \) cannot appear in any accepting cycle.

---

1. All global (shared) data structures have to be copied for the \( n \) pairs, but the memory overhead for the local data structures remains the same.
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.
we must not allow this transition to form a cycle as it would contain the ♣ mark. Since the search from u is initiated after
the search from q0 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 C1 ∪ C2 are marked Dead it may
in the worst case only add redundant explorations. This is
because 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 ♣ mark.

Algorithm 2: Algorithm for checking a GRP

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

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

Data structures.

To represent the Fin and Inf fragments of a GRP, we use
a set of accepting marks per transition. We assign a unique
marking to each F and I* set, for \(1 \leq i \leq l\), and refer to these
markings with \(F_M\) and \(I_M\) and \((\emptyset)_{\leq l} I_M\). The
complete set \(M\) is thus defined as \(M := \{F_M, I^1_M, \ldots, I^l_M\} \).
Each transition \(t\) is combined with an acceptance set \(acc \subseteq M\),
indicating whether \(t \in F\text{ or } t \in I^i\) for \(1 \leq i \leq l\).

We define \(S\) as a mapping from states to pairs, consisting
of a set of states and an acceptance set. Thus \(S(s) = (\text{states}, acc)\)
and formally, \(S : Q \rightarrow 2^Q \times 2^M\). By
implementing \(S\) with a union-find structure, we can main-
tain 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. $S$ pairs can be combined using a `Unite` function. We use an example to illustrate $S$ and the `Unite` function. Let $S(u) := \{(u, w), \{F_M, I_M^1\}\}$ and $S(v) := \{(v, \{F_M, I_M^1\}\}$, we can use the `Unite` function to combine the two structures. After calling `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 `AddAcc(S, v, \{I_M^1, I_M^2\})` will ensure that $S(v).acc$ becomes $\{F_M, I_M^1, I_M^2\}$.

The $Fstates$ structure is implemented as a cyclic list that contains all states added to the list (by means of $Fstates.addState$). $Fstates.pickState$ returns a state from the list, in case the list is nonempty. Finally, states are removed from the list by calling $Fstates.removeState$. 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.

**Detailed algorithm.**

The sequential algorithm for checking a GRP 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 `GRPAccRecur` procedure. After this procedure 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 GRP and the algorithm returns.

In the `GRPAccRecur` procedure, state $s$ is marked as visited and pushed on top of the $R$ stack, along with the accompanying acceptance set $acc_s$ (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 15-28. For each successor $t$ we consider three cases:

- $t \in Dead$ (line 16), this implies that $t$ has already been completely explored and can thus be disregarded.
- $t$ is unvisited (lines 17-20), 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 21-28). This implies that there is some state $r'$ on the $R$ stack such that $t \in S(r').states$ and hence a cycle can be formed. The algorithm then continuously takes the top two states from the $R$ stack and unites them together (and adds the acceptance mark) until $S(s)$ and $S(t)$ are the same. Finally, the acceptance marks from $acc_t$ are added. At line 27, $S(s)$ contains all states on 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 29), $s$ 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 `GRPAcc` algorithm decomposes the GRP 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 $I^i$ sets, for $1 \leq i \leq \ell$, and no transition from $F$. Due to the conditions of line 18 and 21, for a transition with $acc \cap F \neq \emptyset$ it is not possible to start a recursive call with $acc_2$ (thus $acc_2$ never appears on the $R$ stack) nor is it possible to call `AddAcc` with $acc_2$ 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.

**Complexity.**

One can observe that every state and transition is visited at most once in the algorithm. The `GRPAccRecur` 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 [5] we have 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 GRP.

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 `GRPAccRecur` 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 the TGBA checking algorithm from [6].

The \texttt{Fstates} 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 \texttt{Fstates} as a cyclic list becomes clear in the next example. Suppose the \texttt{Fstates} list contains two states, \texttt{u} and \texttt{v}. To avoid contention, the algorithm attempts to divide the workload by assigning half of the workers to search from \texttt{u} and the other half to search from \texttt{v}. Now, assume that \texttt{u} does not have any successors and a large part of the state-space is reached from \texttt{v}. If the search from \texttt{u} completes, we ideally want to let the workers aid in the search from \texttt{v}. By maintaining \texttt{Fstates} as a cyclic list, without much effort we can check which searches have not been completed yet. The \texttt{Fstates} 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 Rabin pairs are checked simultaneously, the global data structures have to be copied. As a result, \(n\) times more memory is required for these structures, in case there are \(n\) Rabin pairs.

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 [26]. 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. This technique 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. [25] 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 GRA conditions in the context of (offline) probabilistic model checking. The idea is to consider each generalized Rabin pair \((F, (I_1, \ldots, I_l))\) 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 \(I_j\), for \(j = 1, \ldots, l\). These sets are then used for computing maximal probabilities. The paper reports significant improvements over check-

ing a degeneralized variant of deterministic GRAs. 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 [28] 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. He uses a BFS-based search technique, in particular a variation on the heuristic piggybacking search [17, 15]. 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 \(\text{Fin}(I) \lor \text{Inf}(F)\) instances (or equivalent, \(\text{Inf}(I) \Rightarrow \text{Inf}(F)\)), 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 64 cores. There is a total of 512GB memory available. We performed all experiments using 16 cores.

implementation.

The TGRA checking algorithm is implemented in the LTSMin toolset [18]. We used several external tools and libraries for generating and parsing the automata:

- Spot v2.3 [10], we used the spot tools \texttt{ltl2tgba} for generating TGBAs and \texttt{autilf} for translating automata to TGRAs (using \texttt{autilf --generalized-rabin}).
- cpphoafparser v0.99.2\(^4\), we used this library to parse a HOA automaton and create an internal representation for representing automata in LTSMin.
- Rabinizer v3.1 [19, 13], we used this tool to generate deterministic transition-based generalized Rabin automata.

\(^4\)Available on \url{http://automata.tools/hoa/cpphoafparser}. 
• LTL3DRA v0.2.4 [2], we used this tool to also generate deterministic automata with transition-based generalized Rabin acceptance, but LTL3DRA only supports a subset of LTL, called LTL\GUX in [4], which is slightly stricter than the set of LTL formulas where no until (U) operator may occur in the scope of any always (G) operator.

• LTL3HOA v1.0.1\(^5\), we used this tool to generate non-deterministic automata with an arbitrary complex transition-based acceptance. We used autfilt --generalized-rabin to translate 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 LTSmin. The algorithms make use of 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 \(2^{28}\) states.

**Experiments.**

We took models and LTL formulas from the 2015 Model Checking Contest [20]. We restricted this set of over 44,000 pairs of models and formulas to those that do not describe obligation properties [24] because using non-Büchi acceptance cannot help producing smaller automata on this class. This selection is further reduced by selecting only the instances where the ‘TGRA generators’ (LTL3DRA, Rabinizer 3 and LTL3HOA) create TGRAs with at least one non-empty Fin set. Otherwise, a GRP is the same as a TGBA, and hence the TGBA emptiness check could be used instead. For this selection, we report results on the experiments (80 in total) for which the time to model check using the TGBA checking algorithm is between 1 second and 10 minutes. We remark that this selection is in favor of the TGBA checking algorithm, since all timeouts and memory errors were filtered out automatically.

For each pair of model \(M\) and formula \(\varphi\) we solved the model checking task \(\mathcal{L}(M \otimes A_{\varphi}) = \emptyset\) using 5 configurations that were repeated 10 times. The configurations were: ltl2tgba using the TGBA checking algorithm, LTL3DRA, Rabinizer 3, LTL3HOA translated to TGRA, and ltl2tgba translated to a TGRA, where the latter four cases used the TGRA checking algorithm introduced in this paper. Every task was run with a timeout of 10 minutes. In total the experiments took approximately 5 days to complete.

All our results and means to reproduce the results are available on https://github.com/utwente-fmt/Rabin-SPIN2017.

**5.2 Main results**

The main results of the experiments are presented in Figure 4 and are summarized in Table 1. One thing to note is that the results are presented on a log-log scale. The (16-core) experiments for the TGBA checking algorithm are provided on the x-axis and the results for the four GRP checking experiments are given on the y-axis. The time for each experiment was repeated 10 times and averaged. All TGRAs are checked by considering each GRP sequentially, i.e., all workers are assigned to the first GRP and continue to the second pair (if there is one) when the first GRP is fully explored.

We encountered a couple of errors in the experiments. There were two instances that resulted in a memory error, meaning that too much memory was allocated during the model checking procedure. These errors only occurred for the TGRA checks and were caused by the additional allocation of the Fstates data structure. There are also two instances that resulted in timeouts for some of the configurations. These both contain counterexamples and suggest that having Fin acceptance instead of only Inf can sometimes lead to bad performance for the TGRA checking algorithm.

**Comparison with Rabinizer 3 and LTL3DRA.**

We observe that most of the results for Rabinizer 3 and LTL3DRA (and to some extent also LTL3HOA) are similar to each other. This could be explained by the fact that both translators produce (deterministic) GRAs that likely do not differ much. We observe that on average, in Table 1, the TGBA checking algorithm performs 37% faster when compared to Rabinizer 3 and 49% faster when compared to LTL3DRA. We highlight a couple of instances.

Arguable the worst performing model is the one at \((x,y)\) position (119,503) in the top-right scatter plot, meaning that the GRA checking algorithm took 503 seconds to complete, while the TGBA checking algorithm performed 4.2 times faster. The corresponding TGBA consists of 1 acceptance set and the GRA is a single pair with a nonempty Fin set and no Inf sets (i.e., a co-Büchi automaton). On further analysis we find that the GRA even contains fewer transitions, namely \(1.05 \cdot 10^{6}\) compared to \(1.23 \cdot 10^{6}\) of the TGBA. However, over 15% of the transitions in the GRA are part of the Fin set. As a result, the performance deficit is likely caused by the overhead of maintaining the Fstates in the algorithm. This instance can be found in the comparisons with Rabinizer 3 and LTL3HOA as well, with similar results.

A better instance is the one at \((9.2.6.5)\) in the top-right scatter plot. In this case, we also have a TGBA with 1 acceptance set and a GRA that equals a co-Büchi automaton. While we again have that the number of Fstates forms a significant part of the total number of transitions (15%), the difference here is that the total number of transitions is much smaller. In total, the TGBA has \(1.4 \cdot 10^{6}\) transitions and the GRA has \(6 \cdot 10^{5}\) transitions. This significant difference with the previous instance is explained by the combination of a costly successor function and the checking times are also reduced by at least a factor of 10. We argue that the GRA checking algorithm takes advantage of the reduced state-space in this instance to outperform the TGBA checking algorithm.

**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. We observed that Fin sets are more often found in TGBAs created from the LTL3HOA instances compared to those from the LTL-to- TGRA translators, hence there are more experiments shown in the scatter plot.

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

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 times from experiments that were checked in all configurations (25 in total, of which 2 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>8.40 (5.42)</td>
<td>8.04 (5.19)</td>
<td>2.83 (1.83)</td>
<td>1.63 (1.05)</td>
<td>1.55</td>
</tr>
<tr>
<td>No counterexample</td>
<td>12.91 (1.33)</td>
<td>12.99 (1.33)</td>
<td>12.96 (1.33)</td>
<td>10.08 (1.04)</td>
<td>9.73</td>
</tr>
<tr>
<td>Total</td>
<td>12.47 (1.48)</td>
<td>12.50 (1.49)</td>
<td>11.48 (1.37)</td>
<td>8.71 (1.04)</td>
<td>8.40</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.

faster by the GRA checking algorithm when the LTL3HOA automaton is used.

One remarkable instance is the one at (48,8) in the top-left scatter plot. The corresponding TGBA is a single-state automaton with one acceptance set, and the GRA is a single-state automaton with two pairs; one pair with a nonempty Fin set, and the other pair is equal to the TGBA. The GRA checking algorithm detects the counterexample while still searching in the first pair (the second pair is never considered), thus the co-Buchi acceptance leads to a 6 time improvement. The TGRA algorithm visits $59 \cdot 10^6$ unique transitions, which is 12% of what the TGBA algorithm explores. Only 3% of the transitions were part of the Fin set.

The result at (1,19) in the top-left scatter plot is an instance where the TGRA algorithm performs significantly worse compared to the TGBA checking algorithm. This is (also) an instance where the TGRA equals a TcBA. One difference with the previous instance is that here, 30% of the transitions are part of the Fin set. This result suggests that Fin sets are only advantageous when a small fraction of the transitions in the product are part of a Fin set.
TABLE 2: Geometric mean sizes of the automata and products. \(|\text{Aut}|\) denotes the number of states in the LTL automaton, \(|\text{Pairs}|\) the number of GRPs in the TGRA, and \(|\text{States}|\) and \(|\text{Trans}|\) provide the sizes of the product automaton. We only used the data from experiments that did not contain a counterexample and were checked in all configurations (23 in total).

<table>
<thead>
<tr>
<th></th>
<th>(\text{Aut})</th>
<th>(\text{Pairs})</th>
<th>(\text{States})</th>
<th>(\text{Trans})</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTL3HOA</td>
<td>1.06</td>
<td>1.00</td>
<td>1.80·10^6</td>
<td>8.72·10^6</td>
</tr>
<tr>
<td>LTL3ORA</td>
<td>1.05</td>
<td>1.03</td>
<td>1.79·10^6</td>
<td>8.74·10^6</td>
</tr>
<tr>
<td>Rabinizer 3</td>
<td>1.64</td>
<td>1.03</td>
<td>1.79·10^6</td>
<td>8.74·10^6</td>
</tr>
<tr>
<td>TGBA-TGRA</td>
<td>2.08</td>
<td>1.00</td>
<td>2.26·10^6</td>
<td>11.88·10^6</td>
</tr>
<tr>
<td>TGBA</td>
<td>2.08</td>
<td>1.00</td>
<td>2.26·10^6</td>
<td>11.88·10^6</td>
</tr>
</tbody>
</table>

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 one memory error (caused by unnecessarily allocating the data structure for \(\text{Fstates}\), since there are no \(\text{Fin}\) 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 \(\text{Fstates}\) as there are no \(\text{Fin}\) 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 (and should) avoid allocating memory for the \(\text{Fstates}\) data structure in case there are no \(\text{Fin}\) sets in the GGRCA. 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 in the automata is, on average, smaller for all GRAs (disregarding TGBA-TGRA) compared to TGBA and the automaton is approximately half the size of that of a TGBA for LTL3HOA and LTL3ORA. If we look at the mean state-space of the product, the sizes for the three GRAs are practically equal. From the Rabinizer 3 case we observe that a larger \(|\text{Aut}|\) does not necessarily result in a larger number of states in the product. Also, the number of GRPs per GRA is almost always equal to one.

We further notice that the product size differs significantly from the GRAs and TGBA. The TGBA products contain approximately 26% more states and 35% more transitions. These numbers suggest that if the GRA 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 TGAs.

Checking GRPs in parallel.
In a number of cases we observe that the GRA consists of 2 GRPs (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 GRPs sequentially. Future work that checks more complicated GRAs may suggest reasons for choosing the alternative approach.

6. CONCLUSION
We introduced a multi-core, on-the-fly algorithm for explicit checking of emptiness on TGAs. 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 \(\text{Fin}\) 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 \(\text{Fin}\) set for the TGRA.
Our experiments do suggest that using TGAs for emptiness checks is advantageous in some scenarios. The product state-space for a TGRA is on average smaller than that of a TGBA. 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 \(\text{Fin}\) 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 to a TGRA and when on a TGBA.

Acknowledgements.
This work is supported by the 3TU.BSR project.

7. REFERENCES


