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 can 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**; *Shared memory algorithms*; • **Mathematics of computing →** *Graph algorithms*;

KEYWORDS

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

ACM Reference format:
Vincent Bloemen, Alexandre Duret-Lutz, and Jaco van de Pol. 2017. Explicit State Model Checking with Generalized Büchi and Rabin Automata. In *Proceedings of International SPIN Symposium on Model Checking of Software, Santa Barbara, CA, USA, July 2017 (SPIN'17),* 10 pages.
https://doi.org/10.1145/3092282.3092288

SPIN'17, July 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5077-8/17/07…$15.00
https://doi.org/10.1145/3092282.3092288

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 `LTL3DRA` [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 $\mathsf{Fin}(T)$ (the transitions in T should be seen finitely often) or $\mathsf{Inf}(T)$ (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 $\mathcal{A} = (\Sigma, Q, q_0, \delta, Acc)$ where Σ 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, Acc is a positive Boolean function over terms of the form $\mathsf{Fin}(T)$ or $\mathsf{Inf}(T)$ for any subset $T \subseteq \delta$. For a transition $t \in \delta$ we note t^s its source, t^ℓ its label, and t^d its destination: $t = (t^s, t^\ell, t^d)$.

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

$$\mathrm{Runs}(\mathcal{A}) = \{\rho \in \delta^\omega \mid \rho(0)^s = q_0 \wedge \forall i \geq 0 : \rho(i)^d = \rho(i+1)^s\}$$

The acceptance of a run ρ is defined by evaluating the acceptance condition Acc over ρ such that:

- $\mathsf{Fin}(T)$ is true iff all the transitions in T occur finitely often in ρ.
- $\mathsf{Inf}(T)$ is true iff some transitions in T occurs infinitely often in ρ.

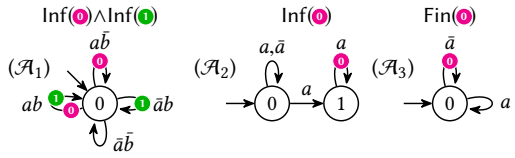

Figure 1: ($\mathcal{A}_1$) a deterministic transition-based generalized Büchi automaton recognizing $\mathsf{GF}a \wedge \mathsf{GF}b$. ($\mathcal{A}_2$) a nondeterministic transition-based Büchi automaton recognizing $\mathsf{FG}a$. ($\mathcal{A}_3$) a deterministic transition-based co-Büchi automaton recognizing $\mathsf{FG}a$.

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

A Transition-based Generalized Büchi Automaton (TGBA) is a TELA where $Acc = \mathsf{Inf}(T_1) \wedge \mathsf{Inf}(T_2) \wedge \ldots \wedge \mathsf{Inf}(T_n)$ 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 $\mathsf{GF}a \wedge \mathsf{GF}b$. Here, transitions are labeled by all possible assignments of a and b, i.e., elements of $\Sigma = \{a\bar{b}, \bar{a}b, a\bar{b}, ab\}$ ($\bar{a}$ denotes the negation of a), and transitions are also *marked* using 0 and 1 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* 0 and 1 infinitely often.

The two automata $\mathcal{A}_2$ and $\mathcal{A}_3$ from Figure 1 represent the formula $\mathsf{FG}a$ 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 Acc has the form $\bigvee_{i=1}^{n}(\mathsf{Fin}(F_i) \wedge \mathsf{Inf}(I_i^1) \wedge \mathsf{Inf}(I_i^2) \wedge \ldots \wedge \mathsf{Inf}(I_i^{\ell_i}))$ for some values of n, and $\ell_1, \ell_2, \ldots, \ell_n$. This is a generalization of Rabin acceptance in the sense that in Rabin acceptance $\ell_i = 1$ for all i. Each conjunctive clause of the form $\mathsf{Fin}(F_i) \wedge \mathsf{Inf}(I_i^1) \wedge \mathsf{Inf}(I_i^2) \wedge \ldots \wedge \mathsf{Inf}(I_i^{\ell_i})$ is called a transition-based generalized Rabin pair (TGRP).

Figure 2 depicts a deterministic TGRA ($\mathcal{A}_4$) and a nondeterministic TGBA ($\mathcal{A}_5$), both representing the property $\mathsf{FG}(\mathsf{F}a \,\mathsf{U}\, b)$. $\mathcal{A}_4$ is accepting if either 1 is visited infinitely often without visiting 0 infinitely often, or if 2 is visited finitely often and both 1 and 3 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 $\wedge$ over $\vee$ to obtain a disjunctive normal form, and then replacing any conjunctive clause of the form $\mathsf{Fin}(F^1) \wedge \mathsf{Fin}(F^2) \wedge \ldots \wedge \mathsf{Fin}(F^m) \wedge \mathsf{Inf}(I^1) \wedge \mathsf{Inf}(I^2) \wedge \ldots \wedge \mathsf{Inf}(I^\ell)$ by the TGRP $\mathsf{Fin}(\bigcup_{i=1}^{n} F^i) \wedge \mathsf{Inf}(I^1) \wedge \mathsf{Inf}(I^2) \wedge \ldots \wedge \mathsf{Inf}(I^\ell)$. This

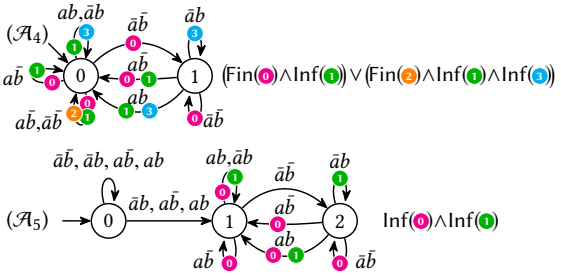

Figure 2: Two automata recognizing FG(Fa U b)**. ($\mathcal{A}_4$) a deterministic transition-based generalized-rabin automaton with two pairs. ($\mathcal{A}_5$) a non-deterministic transition-based generalized Büchi automaton.**

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 $\mathcal{A} = (\Sigma, Q, q_0, \delta, Acc)$, 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_δ. 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_δ 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 $\mathsf{Inf}(T_1) \wedge \ldots \wedge \mathsf{Inf}(T_n)$, 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 $\mathsf{Fin}(F) \wedge \mathsf{Inf}(I^1) \wedge \mathsf{Inf}(I^2) \wedge \ldots \wedge \mathsf{Inf}(I^\ell)$ has an accepting run if there is a trace from the initial state to a reachable partial SCC C with $\forall_{1 \leq i \leq \ell} : I_i \cap C_\delta \neq \emptyset$ 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 $\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)$: the automaton is a unique maximal SCC that does not satisfy $1 \cap C_\delta \neq \emptyset \wedge 0 \cap C_\delta = \emptyset$. 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 ALGORITHM FOR TGRA EMPTINESS

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

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 $\mathsf{TGRP}_1 = \mathsf{TGRP}_2$, 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

```
1 function TGRACheck (Q, q0, TGRA = {TGRP_1, ..., TGRP_n})
2     forall i ∈ {1, ..., n} do
3         TGRPAcc(Q, q0, TGRP_i)
4     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 TGRP 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[1].

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 nth 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 $Acc = (F, \mathcal{I} = \{I^1, \ldots, I^\ell\})$.

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

[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.

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 $\mathcal{A}_{\delta \setminus F} := (\Sigma, Q, q_0, \delta \setminus F, Acc)$. C is an accepting SCC if $\forall_{1 \leq i \leq \ell} : C_\delta \cap I^i \neq \emptyset$, i.e., C contains transitions such that every I^i can be visited infinitely often. By definition of $\mathcal{A}_{\delta \setminus F}$, we have that $C_\delta \cap F = \emptyset$. If C is also reachable from q_0 via transitions from δ (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 $t = (t^s, t^\ell, t^d) \in F$ 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. 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 I^i sets we have detected a counterexample. Otherwise, we pick a state s from the `Fstates` 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 `Fstates` 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 0 and 3 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 0) and the search for pair 2 detects $\mathcal{A}'_2 = C_1 \cup C_2$ (avoiding 3). Consider the search for pair 1, after the initial search it found the `Fstates` u and t. Both have not been explored so suppose that u is arbitrarily chosen as a 'new' initial state. We assume that the search from u visits all states in C_2[3]. 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 0 mark. Since the search from u 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 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 0 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 F and I^i set, for $1 \leq i \leq \ell$, and refer to these marks with F_M and I^i_M (and $\mathcal{I}_M$ for $\bigcup_{1 \leq i \leq \ell} I^i_M$). 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.

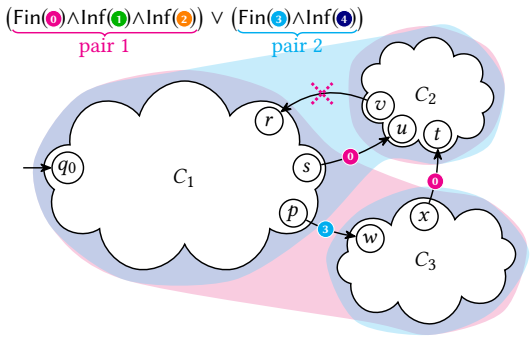

Figure 3: Example of running an emptiness check on an TGRA with two pairs. C_1, C_2, and C_3 represent components (not necessarily strongly connected) that do not contain any transition in the sets 0 or 3. Workers doing the emptiness check for the first pair will first explore $C_1 \cup C_3$, attempting to find a cycle satisfying Inf(1)∧Inf(2) without crossing the 0-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 Inf(4) in $C_1 \cup C_2$, postponing the exploration of C_3 that is only accessible via a 3-transition.

set M is thus defined as $M := \{F_M, I^0_M, \ldots, I^\ell_M\}$. Each transition t is associated to a set of acceptance marks $\mathtt{acc} \subseteq M$, indicating whether $t \in F$ or $t \in I^i$ for $1 \leq i \leq \ell$.

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

$$\forall u, v \in Q : u \in \mathcal{S}(v).states \Leftrightarrow \mathcal{S}(v) = \mathcal{S}(u)$$

This further implies that every state is part of exactly one set of states. In the algorithm, we use $\mathcal{S}$ to associate each state u to its partial SCC that contains the states $\mathcal{S}(u).states$ and visits all the marks in $\mathcal{S}(u).acc$. $\mathcal{S}$ pairs can be combined using a `Unite` function. We use an example to illustrate $\mathcal{S}$ and the `Unite` function. Let $\mathcal{S}(u) := \{\{u, w\}, \{F_M\}\}$ and $\mathcal{S}(v) := \{\{v\}, \{F_M, I^1_M\}\}$, we can use the `Unite` function to combine the two structures. After calling $\mathtt{Unite}(\mathcal{S}, u, v)$ we have $\mathcal{S}(u) = \mathcal{S}(v) = \{\{u, v, w\}, \{F_M, I^1_M\}\}$, 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 $\mathtt{AddAcc}(\mathcal{S}, v, \{I^1_M, I^2_M\})$ will ensure that $\mathcal{S}(v).acc$ becomes $\{F_M, I^1_M, I^2_M\}$.

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.

Algorithm 2: Algorithm for checking a TGRP

```
1  function TGRPAcc (Q, q_0, TGRP = (F_M, I_M))
2      ∀s ∈ Q : S(s) := ({s}, ∅) // Initialize map
3      Visited := Dead := ∅ // Sets
4      R := ∅ // Roots stack of (acc, state) pairs
5      Fstates := {q_0} // Cyclic list of init states
6      while ¬Fstates.isEmpty() do
7          s := Fstates.pickState()
8          if s ∉ Dead then TGRPAccRecur(∅, s)
9          Fstates.removeState(s)
10     return // No Acc got reported in the search
11     function TGRPAccRecur (acc_s, s)
12         Visited := Visited ∪ {s}
13         R.push(acc_s, s)
14         forall (acc_t, t) ∈ suc(s) do
15             if t ∈ Dead then continue // Explored
16             else if t ∉ Visited then // 'New' state
17                 if acc_t ∩ F_M ≠ ∅ then // Avoid F
18                     Fstates.addState(t)
19                 else TGRPAccRecur(acc_t, t)
20             else if acc_t ∩ F_M = ∅ then // Cycle
21                 while S(s) ≠ S(t) do
22                     (acc_r, r) := R.pop()
23                     Unite(S, r, R.top().state)
24                     AddAcc(S, r, acc_r)
25                 AddAcc(S, s, acc_t) // Add acc_t to S(s)
26                 if I_M = S(s).acc then // Acc. cycle
27                     report Acc and exit
28         if s = R.top() then // Completed SCC
29             Dead := Dead ∪ S(s).states
30             R.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 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 14-27. For each successor t we consider three cases:

- $t \in$ 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' on the R stack such that $t \in \mathcal{S}(r').states$ and hence a cycle can be formed. The algorithm then continuously takes the top two states from the R stack and unites them (and adds the acceptance mark) until $\mathcal{S}(s)$ and $\mathcal{S}(t)$ are the same. Finally, the acceptance marks from acc_t are added. At line 26, $\mathcal{S}(s)$ contains all states in the cycle from s to t and forms a partial SCC. $\mathcal{S}(s)$ 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^i 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 $\mathsf{acc}_t \cap F \neq \emptyset$ it is not possible to start a recursive call with acc_t (thus acc_t never appears on the R stack) nor is it possible to call AddAcc with acc_t 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 $\mathcal{S}$ 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, $\mathcal{S}$, 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). $\mathcal{S}$ 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 `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 `Fstates` as a cyclic list becomes clear in the next example. Suppose the `Fstates` 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 `Fstates` as a cyclic list, without much effort we can check which searches have not been completed yet. The `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 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 $(F_i, \{I_i^1, \ldots, I_i^{l_i}\})$ 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_i^j, 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 TGRAs. 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 ω-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 $\mathsf{Fin}(I) \vee \mathsf{Inf}(F)$ instances (or equivalent, $\mathsf{Inf}(I) \Rightarrow \mathsf{Inf}(F)$), 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 some tools from Spot: `ltl2tgba` for generating TGBAs, and `autfilt` for converting automata with other accepting conditions into TGRAs.
- `cpphoafparser v0.99.2`[4]. We used this library to parse a HOA automaton [1] and create an internal representation for LTSmin.
- `Rabinizer v3.1` [13, 19]. 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 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` [24][5]. We used this tool to generate nondeterministic automata with an arbitrarily complex transition-based acceptance. We then used Spot's `autfilt --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 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 [25] 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 TGRP 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 (118 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 cases where timeouts and memory errors occurred in the TGBA algorithm were filtered out as a result of our selection criteria.

For each pair of model M and formula φ we solved the model checking task $\mathscr{L}(M \otimes A_{\neg\varphi}) = \emptyset$ 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.

[4]Available on http://automata.tools/hoa/cpphoafparser.
[5]Available on https://github.com/jurajmajor/ltl3hoa.

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 TGRP 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 TGRP sequentially, i.e., all workers are assigned to the first TGRP and continue to the second pair (if there is one) when the first TGRP 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) TGRAs that likely do not differ much. We observe that on average, in Table 1, the TGBA checking algorithm performs 19% faster when compared to `Rabinizer 3` and 16% faster when compared to `LTL3DRA`. We highlight a couple of instances.

Arguably the worst performing model is the one at (x,y) position (116,517) in the top-right scatter plot, meaning that the TGRA checking algorithm took 517 seconds to complete, while the TGBA checking algorithm performed 4.6 times faster. The corresponding TGBA consists of 1 acceptance set and the TGRA 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 TGRA even contains fewer transitions, namely $1.05 \cdot 10^9$ compared to $1.23 \cdot 10^9$ of the TGBA. However, over 15% of the transitions in the TGRA 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 (9.1,6.3) in the top-right scatter plot. In this case, we also have a TGBA with 1 acceptance set and a TGRA 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 $1.5 \cdot 10^6$ transitions and the TGRA has $0.6 \cdot 10^6$ transitions. This significant difference with the previous instance is explained by a costly successor function. We argue that the TGRA checking algorithm takes advantage of the reduced state-space in this instance to outperform the TGBA checking algorithm.

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).

	LTL3HOA	LTL3DRA	Rabinizer 3	TGBA-TGRA	TGBA
Counterexample	4.19 (0.48)	10.06 (1.14)	11.38 (1.29)	8.53 (0.97)	8.80
No counterexample	10.68 (1.50)	8.30 (1.17)	8.38 (1.18)	7.14 (1.00)	7.12
Total	9.74 (1.34)	8.46 (1.16)	8.64 (1.19)	7.27 (1.00)	7.27

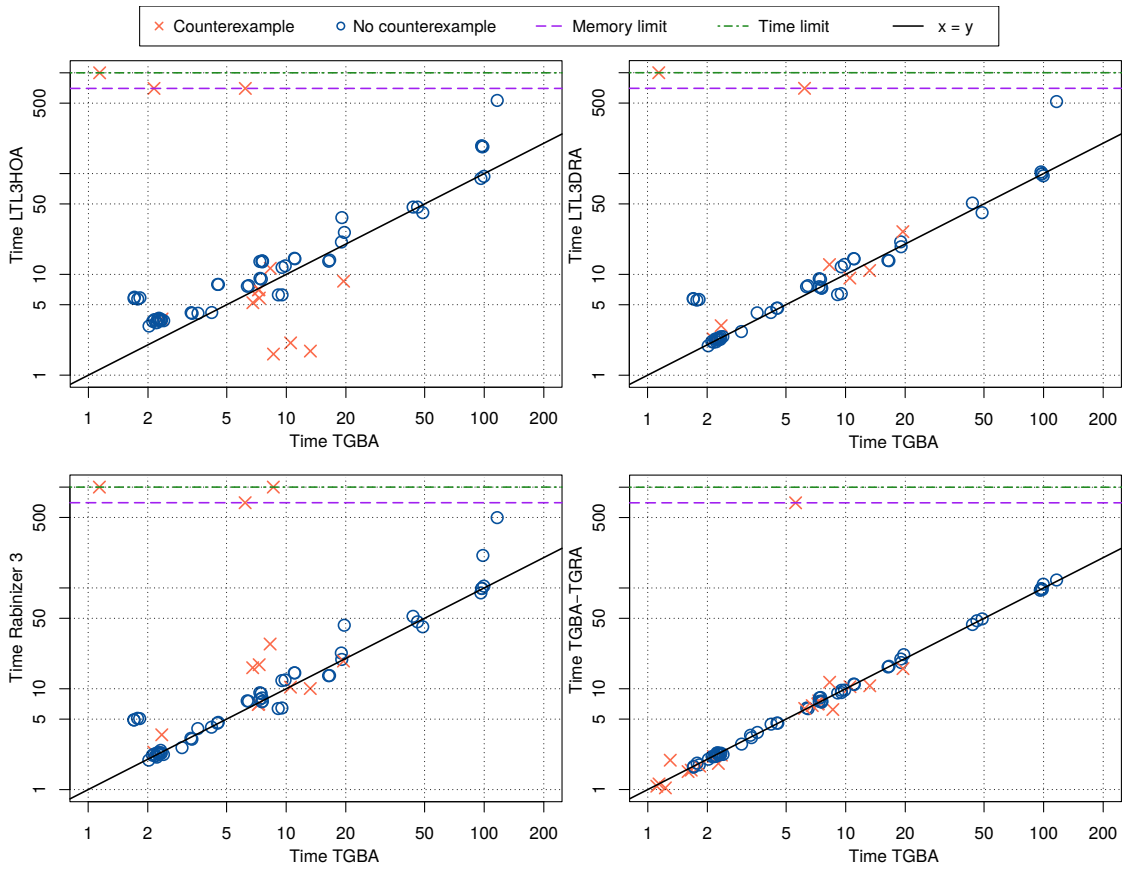

Figure 4: Time (in seconds) comparisons of the TGBA (x-axis) and the TGRA emptiness checks (y-axis), for various LTL to TGRA translations. Each point represents the time to perform an emptiness check using 16 cores, averaged over 10 runs. The TGRA algorithm performed faster for instances below the x=y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

Cross-validation with TGBA Seen as TGRA. Since any TGBA can be trivially rewritten into a TGRA with an empty 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. |Aut| denotes the number of states in the LTL automaton, |Pairs| the number of TGRPs in the TGRA, and |States| and |Trans| provide the sizes of the product automaton. We only used data from experiments that without a counterexample and were checked in *all* configurations (46 in total).

	|Aut|	|Pairs|	|States|	|Trans|
LTL3HOA	1.03	1.41	$1.11 \cdot 10^6$	$5.34 \cdot 10^6$
LTL3DRA	1.02	1.02	$0.78 \cdot 10^6$	$3.78 \cdot 10^6$
Rabinizer 3	1.53	1.03	$0.80 \cdot 10^6$	$3.84 \cdot 10^6$
TGBA-TGRA	1.44	1.00	$0.88 \cdot 10^6$	$4.40 \cdot 10^6$
TGBA	1.44	1.00	$0.88 \cdot 10^6$	$4.40 \cdot 10^6$

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 automaton. 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

[1] T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Křetínský, D. Müller, D. Parker, and J. Strejček. The Hanoi Omega-Automata Format. In *Proc. of CAV'15*, vol. 9206 of *LNCS*, pp. 479–486. Springer, 2015.
[2] T. Babiak, F. Blahoudek, M. Křetínský, and J. Strejček. Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F,G)-Fragment. In *Proc. of ATVA'13*, pp. 24–39. Springer, 2013.
[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[4] F. Blahoudek, M. Křetínský, and J. Strejček. Comparison of LTL to Deterministic Rabin Automata Translators. In *Proc. of LPAR-19*, pp. 164–172. Springer, 2013.

[5] V. Bloemen, A. Laarman, and J. van de Pol. Multi-core On-the-fly SCC Decomposition. In *Proc. of PPoPP'16*, pp. 8:1–8:12. ACM, 2016.

[6] V. Bloemen and J. van de Pol. Multi-core SCC-Based LTL Model Checking. In *Proc. of HVC'16*, pp. 18–33. Springer, 2016.

[7] K. Chatterjee, A. Gaiser, and J. Křetínský. Automata with Generalized Rabin Pairs for Probabilistic Model Checking and LTL Synthesis. In *Proc. of CAV'13*, pp. 559–575. Springer, 2013.

[8] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *Proc. of SPIN'05*, vol. 3639 of *LNCS*, pp. 143–158. Springer, 2005.

[9] E. W. Dijkstra. Finding the Maximum Strong Components in a Directed Graph. In *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science, pp. 22–30. Springer, 1982.

[10] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proc. of ATVA'16*, vol. 9938 of *LNCS*, pp. 122–129. Springer, 2016.

[11] A. Duret-Lutz, D. Poitrenaud, and J.-M. Couvreur. On-the-fly Emptiness Check of Transition-Based Streett Automata. In *Proc. of ATVA'09*, vol. 5799 of *LNCS*, pp. 213–227. Springer, 2009.

[12] E. A. Emerson and C.-L. Lei. Modalities for Model Checking (Extended Abstract): Branching Time Strikes Back. In *Proc. of POPL'85*, pp. 84–96. ACM, 1985.

[13] J. Esparza, J. Křetínský, and S. Sickert. From LTL to deterministic automata. *Formal Methods in System Design*, 49(3):1–53, 2016.

[14] S. Evangelista, A. Laarman, L. Petrucci, and J. van de pol. Improved Multi-Core Nested Depth-First Search. In *Proc. of ATVA'12*, vol. 7561 of *LNCS*, pp. 269–283. Springer, 2012.

[15] I. Filippidis and G. J. Holzmann. An Improvement of the Piggyback Algorithm for Parallel Model Checking. In *Proc. of SPIN'14*, pp. 48–57. ACM, 2014.

[16] G. Holzmann, R. Joshi, and A. Groce. Swarm Verification Techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011.

[17] G. J. Holzmann. Parallelizing the Spin Model Checker. In *Proc. of SPIN'12*, vol. 7385 of *LNCS*, pp. 155–171. Springer, 2012.

[18] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In *Proc. of TACAS'15*, vol. 9035 of *LNCS*, pp. 692–707. Springer, 2015.

[19] Z. Komárková and J. Křetínský. Rabinizer 3: Safraless Translation of LTL to Small Deterministic Automata. In *Proc. of ATVA'14*, pp. 235–241. Springer, 2014.

[20] F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, A. Linard, M. Beccuti, A. Hamez, E. Lopez-Bobeda, L. Jezequel, J. Meijer, E. Paviot-Adet, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, and K. Wolf. Complete Results for the 2015 Edition of the Model Checking Contest. http://mcc.lip6.fr/2015/results.php, 2015.

[21] A. Laarman, J. van de Pol, and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In *Proc. of NFM'11*, Lecture Notes in Computer Science, pp. 506–511. Springer, 2011.

[22] Y. Liu, J. Sun, and J. Dong. Scalable Multi-core Model Checking Fairness Enhanced Systems. In *Proc. of ICFEM'09*, vol. 5885 of *LNCS*, pp. 426–445. Springer, 2009.

[23] G. Lowe. Concurrent depth-first search algorithms based on Tarjan's Algorithm. *International Journal on Software Tools for Technology Transfer*, pp. 1–19, 2015.

[24] J. Major. Translation of LTL into nondeterministic automata with generic acceptance condition. Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2017. http://is.muni.cz/th/396325/fi_m.

[25] Z. Manna and A. Pnueli. A Hierarchy of Temporal Properties. In *Proc. of PODC'87*, pp. 205–205. ACM, 1987.

[26] E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Variations on parallel explicit emptiness checks for generalized Büchi automata. *International Journal on Software Tools for Technology Transfer*, pp. 1–21, 2016.

[27] S. Schwoon and J. Esparza. A Note on On-the-Fly Verification Algorithms. In *Proc. of TACAS'05*, vol. 3440 of *LNCS*, pp. 174–190. Springer, 2005.

[28] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS'86*, pp. 322–331. IEEE Computer Society, 1986.

[29] A. Wijs. BFS-Based Model Checking of Linear-Time Properties with an Application on GPUs. In *Proc. of CAV'16*, pp. 472–493. Springer, 2016.