International Journal on Software Tools for Technology Transfer (2019) 21:307–324
https://doi.org/10.1007/s10009-019-00508-4

SPIN 2017

Model checking with generalized Rabin and Fin-less automata

Vincent Bloemen[1] · **Alexandre Duret-Lutz**[2] · **Jaco van de Pol**[3]

Published online: 25 February 2019

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 a transition-based generalized Rabin automaton (TGRA), 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 and generated LTL formulas for models from the BEEM database. While we found little to no improvement by checking TGRAs directly, we show how various aspects of a TGRA's structure influences the model checking performance. In this paper, we also introduce a Fin-less acceptance condition, which is a disjunction of TGBAs. We show how to convert TGRAs into automata with Fin-less acceptance and show how a TGBA emptiness procedure can be extended to check Fin-less automata.

Keywords Model checking · Explicit state · LTL · ω-Automata · On-the-fly · Generalized · Büchi · Rabin · Multi-core · Parallel

1 Introduction

1.1 Model checking

Model checking is a way to ensure that a system modelled by a Kripke Structure K satisfies some behavioural properties expressed as an LTL formula φ. In the automata theoretic approach to LTL model checking [34], the formula φ is first transformed into a Büchi automaton $A_{\neg\varphi}$ capturing forbidden behaviours. This automaton is then synchronized with the system K, and the procedure then amounts to testing whether the language of this synchronized product is empty: $\mathscr{L}(K \otimes A_{\neg\varphi}) = \emptyset$. If the language of the product is non-empty, it means there exists a counterexample: an execution of K that does not satisfy φ.

When performed explicitly (i.e. not using any kind of symbolic representation of those automata), the procedure is limited by the well known *state-space explosion problem*, where the product automaton $K \otimes A_{\neg\varphi}$ becomes too large to handle.

In what follows, we will focus on the emptiness check procedure, i.e. the algorithm that takes an automaton $\mathcal{A}$ as input, and decides if its language $\mathscr{L}(\mathcal{A})$ is empty. We abstract away that fact that in a model checker $\mathcal{A}$ is a product, but we have to account for the fact that $\mathcal{A}$ can be quite large.

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. Reachable states are computed on demand 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 [33].

✉ Vincent Bloemen
v.bloemen@utwente.nl

Alexandre Duret-Lutz
adl@lrde.epita.fr

Jaco van de Pol
jaco@cs.au.dk

[1] University of Twente, Enschede, The Netherlands

[2] LRDE, EPITA, Kremlin-Bicêtre, France

[3] University of Aarhus, Århus, Denmark

With current hardware systems, one can further improve the model checking performance by using 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 [8,16,20,32].

Finally a 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.

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 an algorithm that enumerates strongly connected components (SCCs) at no extra cost compared to SCC-based algorithms on BAs [10].

1.2 Our goal: emptiness checks using generalized Rabin automata

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. More recently, order-of-magnitude speedups were reported when performing probabilistic model checking using a generalized acceptance condition called *transition-based generalized Rabin automata* (TGRAs) [9]. Also, there has been a lot of interest into building tools such as `LTL3DRA` [2] and `Rabinizer 3&4` [15,22,24] 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) [8,16,32,33]; however, for Rabin acceptance there is only a recent work on a GPU algorithm for checking (non-generalized) RAs [35] and a TGRA checking algorithm for probabilistic model checking [9].

None of these works address our question: in a setting where determinism is not necessary, 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 [21], and benchmark several model checking tasks realized using TGRAs or TGBAs.

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 [12] that work with automata using arbitrary acceptance conditions [1]. In Spot 2.5.2, ω-automata with complex acceptance conditions are first converted into TGBAs before being emptiness-checked.

Two recent tools called `LTL3TELA` (no publication yet) and `Delag` [30] convert LTL formulas into automata with unconstrained acceptance conditions: the acceptance is simply chosen to help the translation into smaller (in the case of `LTL3TELA`) or deterministic (`Delag`) automata. So far it is not clear whether these translations can be useful in the context of explicit model checking. By looking at TGRAs, we contribute a partial answer to this question.

This paper is an updated version of an article published in the Spin'17 conference [6]. We extend that previous work in two ways.

First, in this paper we also introduce a *Fin-less* acceptance condition, which is a disjunction of TGBAs. We show how a TGBA emptiness procedure can be trivially extended to support Fin-less acceptance and we empirically compare how automata with a Fin-less acceptance relate to TGBA and TGRA in terms of model checking performance.

Second, we extend the set of experiments to include benchmark models from the BEEM database [31], accompanied by randomly generated LTL formulas (obtained from [4]). In total, more than 3000 model and formula combinations were added. We observed that the TGRAs generated from these formulas have a more complex structure compared to the ones from prior experiments. Experiments show how this affects the relative performance of checking TGRAs versus TGBAs. We also provide a more detailed analysis of the results.

1.3 Overview

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

Table 1 Acceptance condition formulas corresponding to classical names

(B)	Büchi	$\mathsf{Inf}(I_1)$
(GB)	Generalized-Büchi	$\bigwedge_i \mathsf{Inf}(I_i)$
(C)	Co-Büchi	$\mathsf{Fin}(F_1)$
(R)	Rabin	$\bigvee_i \mathsf{Fin}(F_i) \wedge \mathsf{Inf}(I_i)$
(GR)	Generalized-Rabin	$\bigvee_i \mathsf{Fin}(F_i) \wedge \mathsf{Inf}(I_i^1) \wedge \mathsf{Inf}(I_i^2) \wedge \ldots \wedge \mathsf{Inf}(I_i^{p_i})$
(FL)	Fin-less	$\bigvee_i \mathsf{Inf}(I_i^1) \wedge \mathsf{Inf}(I_i^2) \wedge \ldots \wedge \mathsf{Inf}(I_i^{p_i})$
(EL)	Emerson–Lei	any positive formula of $\mathsf{Fin}(F_i)$ and $\mathsf{Inf}(I_i)$

F_i and I_i denote sets of transitions

is similar to the formalism used by Emerson and Lei 30 years ago [14] using state-based acceptance.

Definition 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 occur infinitely often in ρ.

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 $\mathcal{A}$, denoted $\mathscr{L}(\mathcal{A})$, is the set of all words ρ^ℓ recognized by some accepting run ρ.

Some shape of acceptance conditions Acc are given names, as shown in Table 1. We use the abbreviation TXA, where X is any value of the first column of Table 1, to denote a TELA whose acceptance condition has the shape given in the last column.

For instance, 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.

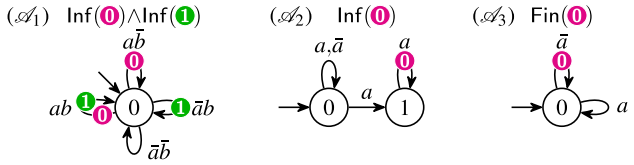


Fig. 1 ($\mathcal{A}_1$) a deterministic transition-based generalized Büchi automaton recognizing $\mathsf{GF}a \wedge \mathsf{GF}b$. ($\mathcal{A}_2$) a non-deterministic transition-based Büchi automaton recognizing $\mathsf{FG}a$. ($\mathcal{A}_3$) a deterministic transition-based co-Büchi automaton recognizing $\mathsf{FG}a$ (colour figure online)

As an example, automaton $\mathcal{A}_1$ from Fig. 1 represents a TGBA for the formula $\mathsf{GF}a \wedge \mathsf{GF}b$. Here, transitions are labelled by all possible assignments of a and b, i.e. elements of $\Sigma = \{\bar{a}\bar{b}, \bar{a}b, a\bar{b}, ab\}$ ($\bar{a}$ 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 both *acceptance marks* 0 and 1 infinitely often.

A transition-based generalized Rabin automaton (TGRA) is a TELA where Acc has the form $\bigvee_{i=1}^{n}\big(\mathsf{Fin}(F_i) \wedge \mathsf{Inf}(I_i^1) \wedge \mathsf{Inf}(I_i^2) \wedge \ldots \wedge \mathsf{Inf}(I_i^{p_i})\big)$ for some values of n, and $p_1, p_2, \ldots, p_n$. This is a generalization of Rabin acceptance in the sense that in Rabin acceptance $p_i = 1$ for all i. Each conjunctive clause of the form $\mathsf{Fin}(F_i) \wedge \mathsf{Inf}(I_i^1) \wedge \mathsf{Inf}(I_i^2) \wedge \ldots \wedge \mathsf{Inf}(I_i^{p_i})$ is called a transition-based generalized Rabin pair (TGRP). A transition-based co-Büchi automaton is a TGRA with $n = 1$ and $p_1 = 0$; co-Büchi acceptance consists of a single clause of the form $\mathsf{Fin}(F_1)$.

The two automata $\mathcal{A}_2$ and $\mathcal{A}_3$ from Fig. 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 an automaton. Moreover, $\mathcal{A}_3$ is a deterministic automaton, whereas no equivalent deterministic BA exists.

Figure 2 depicts a deterministic TGRA ($\mathcal{A}_4$) and a non-deterministic 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 reducing the size of the automaton.

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 $\mathsf{Fin}(F^1) \wedge \mathsf{Fin}(F^2) \wedge \ldots \wedge \mathsf{Fin}(F^m) \wedge \mathsf{Inf}(I^1) \wedge \mathsf{Inf}(I^2) \wedge \ldots \wedge \mathsf{Inf}(I^p)$ by the TGRP $\mathsf{Fin}(\bigcup_{i=1}^{n} F^i) \wedge \mathsf{Inf}(I^1) \wedge \mathsf{Inf}(I^2) \wedge \ldots \wedge \mathsf{Inf}(I^p)$. This con-

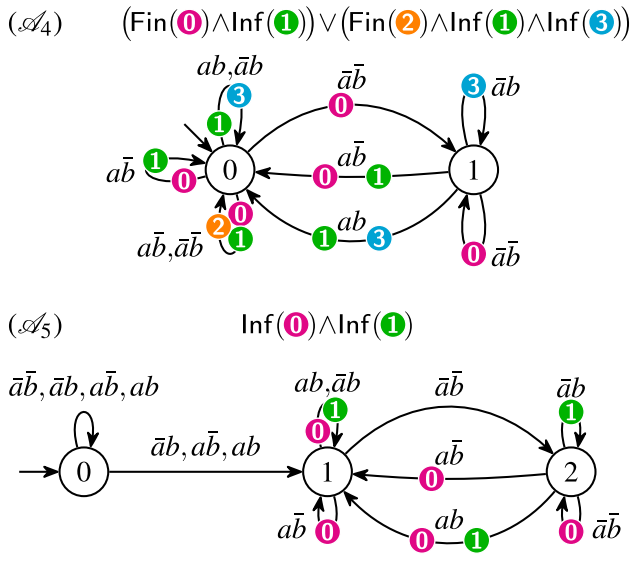


Fig. 2 Two automata recognizing $\mathsf{FG}((\mathsf{F}a) \mathsf{U}\, b)$. ($\mathcal{A}_4$) a non-deterministic transition-based generalized-Rabin automaton with two pairs. ($\mathcal{A}_5$) a non-deterministic transition-based generalized Büchi automaton (colour figure online)

version can be done without changing the transition structure of the automaton; but the 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 (*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 without losing strong connectivity. An SCC is called *trivial* if $C_\delta = \emptyset$, and hence C_Q consists of a single state.

In a TGBA whose acceptance condition has 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. A TGRP $\mathsf{Fin}(F) \wedge \mathsf{Inf}(I^1) \wedge \mathsf{Inf}(I^2) \wedge \ldots \wedge \mathsf{Inf}(I^p)$ has an accepting run if there is a trace from the initial state to a reachable partial SCC C with $F \cap C_\delta = \emptyset$ and $I_i \cap C_\delta \neq \emptyset$ for all $1 \leq i \leq p$. In other words, 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 for a *partial* SCC intersecting all acceptance sets by the search for 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 Fig. 2 checked against the TGRP $\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)$: the automaton has a unique maximal SCC, consisting of both states and every transition, which does not satisfy $0 \cap C_\delta = \emptyset \wedge 1 \cap C_\delta \neq \emptyset$. 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 labelled by Fin sets.

At the cost of introducing non-determinism, any generalized Rabin automaton can be converted into what we have called Fin-less automaton[1] (TFLA) in Table 1.

Proposition 1 (Fin-removal) *Given a TGRA* $\mathcal{A} = (\Sigma, Q, q_0, \delta, \bigvee_{i=1}^n \mathsf{Fin}(F_i) \wedge \mathsf{Inf}(I_i^1) \wedge \mathsf{Inf}(I_i^2) \wedge \ldots \wedge \mathsf{Inf}(I_i^{p_i}))$, *the TFLA* $\mathcal{B} = (\Sigma, Q', q_0, \delta', \bigvee_{i=1}^n \mathsf{Inf}(J_i^1) \wedge \mathsf{Inf}(J_i^2) \wedge \ldots \wedge \mathsf{Inf}(J_i^{p_i}))$ *where:*

- $Q' = Q \cup Q \times \{1, 2, \ldots, n\}$
- $\delta' = \delta$
  $\cup\, \{(t^s, t^\ell, (t^d, i)) \mid i \in \{1, 2, \ldots, n\} \wedge t \in (\delta \backslash F_i)\}$
  $\cup\, \{((t^s, i), t^\ell, (t^d, i)) \mid i \in \{1, 2, \ldots, n\} \wedge t \in (\delta \backslash F_i)\}$
- $J_i^j = \{(t^s, i), t^\ell, (t^d, i) \mid t \in (I_i^j \setminus F_i)\}$ *for* $i \in \{1, 2, \ldots, n\}$ *and* $j \in \{1, 2, \ldots, p_i\}$

is such that $\mathscr{L}(\mathcal{A}) = \mathscr{L}(\mathcal{B})$.

Any accepting run of $\mathcal{A}$ will eventually reach a point where all the transitions it visited satisfy one of the generalized Rabin pairs. The above construction, illustrated by Fig. 3, works by introducing non-determinism to guess this point and the pair satisfied. The non-deterministic transitions (pictured with dashed lines) connect to clones of the original automaton,[2] in which taking the transitions of F_i (for pair i) is forbidden from now on, and the acceptance condition is set so that the other Inf sets still have to be visited infinitely often.

Given a TFLA $\mathcal{B}$ with acceptance $\bigvee_{i=1}^n \mathsf{Inf}(J_i^1) \wedge \mathsf{Inf}(J_i^2) \wedge \ldots \wedge \mathsf{Inf}(J_i^{p_i})$, an accepting run has to satisfy one Inf conjunction. This means that there is an accepting run if, for some

[1] Strictly speaking, a Fin-less automaton could use any formula using only Inf terms. We assume that the formula is under disjunctive normal form for simplicity and because this is what the construction of Proposition 1 produces.

[2] When implementing this construction, the number of non-deterministic jumps can be reduced: only one such jump is needed per cycle of the sub-automaton created for each generalized Rabin pair.

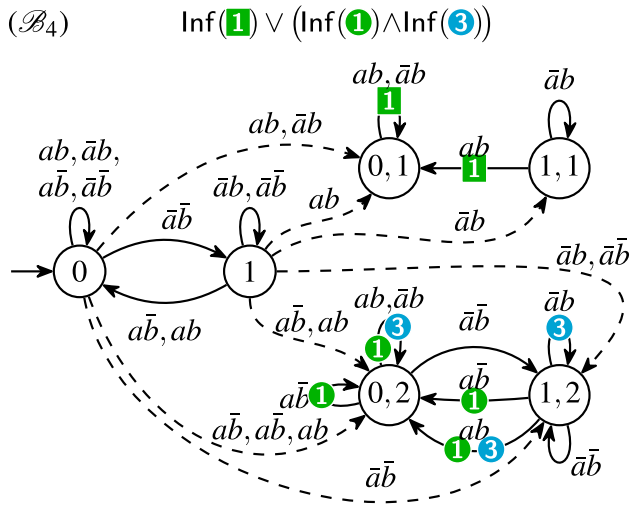


Fig. 3 Application of Proposition 1 to transform the TGRA $\mathcal{A}_4$ of Fig. 2 into a TFLA $\mathcal{B}_4$. The dashed transitions correspond to non-deterministic jumps added to some copies of the original automaton. Each copy handles one generalized Rabin pair of the original acceptance condition. In the original acceptance 1 was used in both pairs, but since Proposition 1 creates a different set of each use, we distinguish these two sets with 1 and 1 here (colour figure online)

$1 \leq i \leq n$, there is a trace from the initial state to a reachable partial SCC C such that $J_i^k \cap C_\delta \neq \emptyset$, for all $1 \leq k \leq p_i$.

A TGBA checking algorithm can be easily extended to also check for TFLA emptiness. This is achieved by tracking all acceptance sets in each found partial SCC. Thus, given the automaton $\mathcal{B}_4$, we would track whether 1, 1, and 3 have a non-empty intersection in each SCC C. Then, when checking whether a partial SCC C contains an accepting cycle, the algorithm iterates over $1 \leq i \leq n$ to search for an i such that $J_i^k \cap C_\delta \neq \emptyset$ holds for all $1 \leq k \leq p_i$. Note that, as with the case for TGBAs, it is also always valid to only check for *maximal* SCCs instead of *partial* SCCs.

The Fin-removal construction procedure, in combination with the extended TGBA emptiness checking algorithm, could be regarded as a method for checking TGRAs. In fact, the non-determinism introduced by the Fin-removal construction is very similar to non-deterministically choosing to check for one of the TGRPs. An accepting run on a TFLA starts by first forming a trace from the initial state of a TGRA to an arbitrary state in the TGRA. Then, a non-deterministic choice is made to decide which TGRP will be checked. Finally, the trace is continued to a reachable partial SCC that does not contain any Fin transitions from the chosen TGRP. Note that this accepting run is also accepting in the corresponding TGRA. The downsides of using the Fin-removal construction are that it introduces non-determinism in the automaton, and it linearly increases the size of the automaton (in the number of TGRPs).

Algorithm 1: Checking TGRA by checking TGRPs.

```
1 function TGRACheck (Q, q0, TGRA = {TGRP1, ..., TGRPn})
2     forall the i ∈ {1, ..., n} do
3         TGRPAcc(Q, q0, TGRPi)
4     return No_Acc // No TGRPAcc call reported Acc
```

3 Algorithm for TGRA emptiness

In this section, we present a direct algorithm for checking emptiness on TGRAs, without removing Fin sets first. We start by splitting up the TGRA acceptance into individual TGRPs and show how these can be checked.

3.1 Checking Rabin pairs

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. The TGRPAcc procedure itself will be explained later.

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, if $\text{TGRP}_1 = \text{TGRP}_2$. Even an overlap in the Fin and/or Inf fragments of the TGRPs might offer an opportunity to combine gained information.

3.2 TGRP checking algorithm

Throughout this section, we consider checking a TGRP with acceptance of the form $Acc = (F, \{I^1, \ldots, I^p\})$. We note that a TGRP can be seen as an extension of a TGBA, in which a Fin constraint is added. The algorithm that we propose is an extension of the best parallel algorithm for checking TGBAs that we know [7,8]. We present the algorithm's sequential execution and show how it can be parallelized.

3.2.1 Abstract idea of the algorithm

The general idea of the algorithm, which we present in Algorithm 3 (a simplified version of the algorithm is given in Algorithm 2), is to perform an SCC decomposition of the automaton without allowing any F transitions from being part of the SCCs. As a result, we obtain SCCs that contain all edges except those in F. Formally, we have that each SCC C is a partial SCC of $\mathcal{A}$ that is maximal on the TGRP $\mathcal{A}_{\delta \setminus F} := (\Sigma, Q, q_0, \delta \setminus F, Acc)$. C is an accepting SCC if $C_\delta \cap I^i \neq \emptyset$ for each $1 \leq i \leq p$, 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.

3.2.2 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.[3] 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 automaton $\mathcal{A}'$, 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 consider the following 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 Fig. 4. Here, two TGRPs are checked separately. Note that due to the way how ❶ and ❸ 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 ❶) and the search for pair 2 detects $\mathcal{A}'_2 = C_1 \cup C_2$ (avoiding ❸). 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.[4] If we now find the edge from v to r, thus from C_2 to C_1, we should not report an accepting cycle as it would contain the ❶ mark. This is guaranteed, since the search from u is initiated *after* the search from q_0 is complete; so 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.

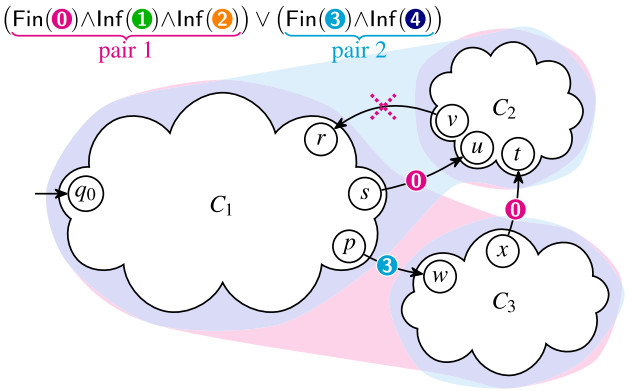


Fig. 4 Example of running an emptiness check on an TGRA with two pairs. C_1, C_2, and C_3 represent components (not necessarily strongly connected) that do not contain any transition in the sets ❶ or ❸. Workers doing the emptiness check for the first pair will first explore $C_1 \cup C_3$, attempting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

This is because the edge from s to u is not included as an edge in the SCC decomposition and hence no cycle can be formed with a ❶ mark.

3.2.3 Simplified algorithm

In Algorithm 2, we present a simplified version of the TGRP checking algorithm. The function should be initially called with `s` $:= q_0$ and it recursively explores the graph in a depth-first order. A state is marked visited, then it's successors are processed, then the state is explored and it gets marked `Dead`. All `Dead` states are ignored when processing successors. If the successor t has not been visited yet, depending on whether it is reached via an F transition, we either store it in the `Fstates` set, or we recursively explore the state. If t has been visited before and it is not reached via an F transition, then we detected a cycle. We then want to combine the states and transition marks on this cycle to check whether this is an accepting cycle. Once the initial state is fully explored, we check whether there are stored states in `Fstates`, and if so, we start searching from these states until we processed the entire state-space.

3.2.4 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 $1 \leq i \leq p$, and refer to these marks with F_M and I^i_M. We denote the set of all Inf marks by $\mathscr{I}_M$, i.e. $\mathscr{I}_M := \bigcup_{1 \leq i \leq p} I^i_M$. The complete set of markings M is

[3] 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.

[4] 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.

Algorithm 2: Simplified algorithm for one TGRP.

```
1  Visited := Dead := Fstates := ∅ // Initializing sets
2  function TGRPAccSimple (Q, s, TGRP = (F_M, 𝓘_M))
3      Visited := Visited ∪ s // Mark s as visited
4      forall the (acc_t, t) ∈ suc(s) do // Explore successors
5          // Ignore fully explored (Dead) states
6          if t ∈ Dead then continue
7          else if t ∉ Visited then // Unseen state t
8              // Store F transitions for later
9              if acc_t ∩ F_M ≠ ∅ then Fstates.addState(t)
10             else // recursively explore otherwise
11                 TGRPAccSimple(Q, t, TGRP)
12         // If visited before, we found a cycle
13         else if acc_t ∩ F_M = ∅ then // No F transition
14             Unite states and combine acc sets on cycle
15             if 𝓘_M = 'combined acc set' then
16                 report Acc and exit // Accepting cycle
17     Dead := Dead ∪ s // Explored s, mark s as Dead
18     // If backtracking from the 'initial' state
19     if Dead = Visited then
20         if ¬Fstates.isEmpty() then // Stored F states
21             f := Fstates.pickState() // Try next one
22             TGRPAccSimple(Q, f, TGRP) // New search
23         else exit // No accepting cycle found
```

thus defined as $M := \{F_M, I_M^0, \ldots, I_M^p\}$. 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 $1 \leq i \leq p$.

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

$$\forall u, v \in Q\colon u \in \mathscr{S}(v).\mathit{states} \Leftrightarrow \mathscr{S}(v) = \mathscr{S}(u)$$

This further implies that every state is part of exactly one set of states. In the algorithm, we use $\mathscr{S}$ to associate each state u to its partial SCC that contains the states $\mathscr{S}(u).\mathit{states}$ and visits all the marks in $\mathscr{S}(u).\mathtt{acc}$. $\mathscr{S}$ pairs can be combined using a `Unite` function. We use an example to illustrate $\mathscr{S}$ and the `Unite` function. Let $\mathscr{S}(u) := (\{u, w\}, \{F_M\})$ and $\mathscr{S}(v) := (\{v\}, \{F_M, I_M^1\})$, we can use the `Unite` function to combine the two structures. After calling $\mathtt{Unite}(\mathscr{S}, u, v)$ we have $\mathscr{S}(u) = \mathscr{S}(v) = (\{u, v, w\}, \{F_M, I_M^1\})$, while keeping all other mappings the same. For more details on this structure, we refer to Bloemen et al. [7]. We use an additional function `AddAcc` to 'add' (the union of) acceptance marks to the set, thus $\mathtt{AddAcc}(\mathscr{S}, v, \{I_M^1, I_M^2\})$ will ensure that $\mathscr{S}(v).\mathtt{acc}$ 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 non-empty. 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 3: 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 the (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()
```

3.2.5 Detailed algorithm

The detailed sequential algorithm for checking a TGRP is presented in Algorithm 3. We repeatedly perform SCC decompositions, using Dijkstra's algorithm [11]. To this end, we will maintain a stack R, which can be regarded as an extension to the *roots* stack from Dijkstra's SCC algorithm [11].

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. 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, then the complete state-space must have 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 $\texttt{acc}_s$ (note that since there is no transition to the initial state, the empty set is given in line 8). 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 $t \in \mathscr{S}(r').states$ 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 $\mathscr{S}(s)$ and $\mathscr{S}(t)$ are the same. Finally, the acceptance marks from $\texttt{acc}_t$ are added. At line 26, $\mathscr{S}(s)$ contains all states in the cycle from s to t and forms a partial SCC. $\mathscr{S}(s)$ 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`.

Figure 5 shows an example run of the algorithm on a small automaton.

3.2.6 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 $1 \leq i \leq p$, and no transition from F. Due to the conditions of lines 17 and 20, for a transition with $\texttt{acc}_t \cap F \neq \emptyset$ it is not possible to start a recursive call with $\texttt{acc}_t$ (thus $\texttt{acc}_t$ never appears on the R stack) nor is it possible to call `AddAcc` with $\texttt{acc}_t$ 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.

3.2.7 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 $\mathscr{S}$ structure. From previous work [7], we know that the union-find structure (without tracking acceptance marks) causes the complete algorithm to operate in quasi-linear time. By assuming that the number of acceptance marks $|M|$ $(= 1 + p)$ is a small constant (which holds 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 $\log(|\delta|)$ factor acts as an over approximation for the quasi-linear time).

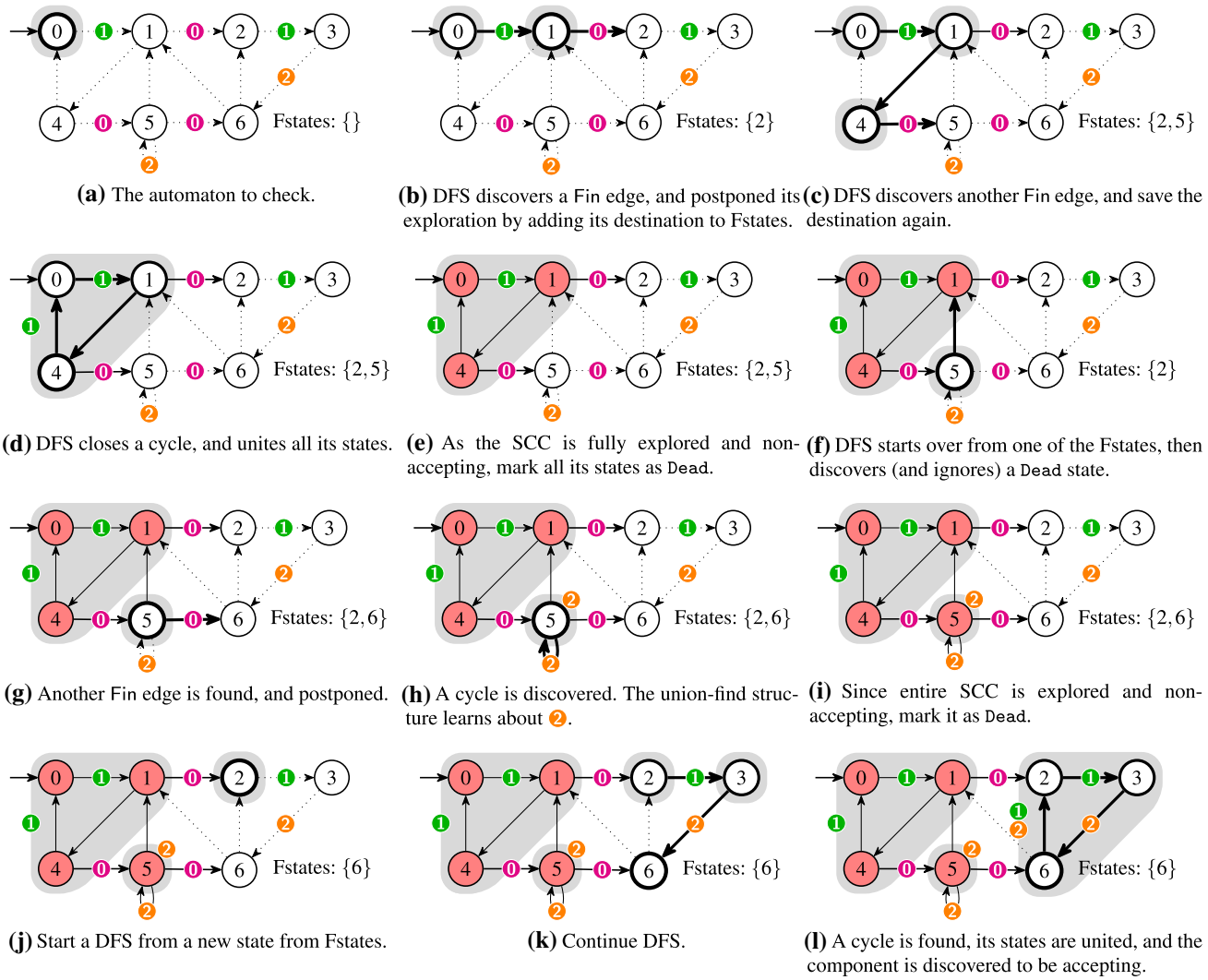
The space complexity is limited by the sizes of the R, $\mathscr{S}$, and `Fstates` structures. R may contain up to $|Q|$ states and acceptance marks in the worst case (by visiting every state in a single path). $\mathscr{S}$ can be implemented as an array of length $|Q|$ of structs that are of constant 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.

3.3 Parallel implementation

We now present two ways in which parallelism can be used to speed up the checking process for TGRAs.

3.3.1 Parallel TGRA checking

We first consider Algorithm 1 for checking each TGRP separately. After a TGRPAcc call has finished, the next Rabin



(a) The automaton to check.

(b) DFS discovers a Fin edge, and postponed its exploration by adding its destination to Fstates.

(c) DFS discovers another Fin edge, and save the destination again.

(d) DFS closes a cycle, and unites all its states.

(e) As the SCC is fully explored and non-accepting, mark all its states as `Dead`.

(f) DFS starts over from one of the Fstates, then discovers (and ignores) a `Dead` state.

(g) Another Fin edge is found, and postponed.

(h) A cycle is discovered. The union-find structure learns about 2.

(i) Since entire SCC is explored and non-accepting, mark it as `Dead`.

(j) Start a DFS from a new state from Fstates.

(k) Continue DFS.

(l) A cycle is found, its states are united, and the component is discovered to be accepting.

Fig. 5 Example of running the algorithm on a small automaton and a single TGRP: Fin(0) ∧ Inf(1) ∧ Inf(2). In this case, transitions labelled by 0 should be avoided. Unexplored transitions are dotted. States and transitions on the DFS stack are in bold. Dead states are coloured. The grey background represents the current view of the SCCs as stored in the union-find data structure; for instance,  means that $\mathscr{S}(u) = \mathscr{S}(v) = (\{u, v\}, 1)$ (colour figure online)

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 the latter 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.[5]

However, one advantage of checking all pairs in parallel is that these jobs are completely independent, so this could be the basis for a distributed algorithm. We expect better scalability, since parallel workers on a single pair have to synchronize on a shared datastructure (our implementation uses a concurrent hashtable and a shared union-find datastructure). Also, when P gets large compared to the size of the (remaining) graph, the probability that workers have to wait for each other, or perform duplicate work, increases.

[5] 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.

Another advantage could be that counterexamples may be detected faster in the latter setting. Suppose for example that only the nth pair contains a counterexample that is detected by visiting only part of the state-space. Then the parallel pairs approach prevents traversing the complete state-space $n - 1$ times.

We have experimented with these two extreme parallelization approaches, but it is conceivable that a more flexible job scheduler with load balancing leads to even higher speedups. Note, however, that in practice the number of Rabin pairs is quite limited.

3.3.2 Parallel TGRP checking

Algorithm 3 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. [7,8]. 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 [8].

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 [7].

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 a significant performance 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, if all TGRPs are checked simultaneously, a copy of the global data structures has to be made for each group of workers that process a different TGRP. As a result, n times more memory is required for these structures in case there are n TGRPs. Though, as mentioned before, checking TGRPs simultaneously could reduce the computation time (compared to checking every TGRP one-by-one).

4 Related work

4.1 Related work on checking Büchi automata

Explicit state on-the-fly algorithms for checking can be distinguished into two classes, namely *Nested Depth-First Search (NDFS)* and *SCC-based* algorithms. Schwoon and Esparza provide a great overview on these techniques [33]. 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 [16] to be the state-of-the-art NDFS-like algorithm. It is based on *swarm verification* [19] 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. [8] outperforms other techniques and performs comparable 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 those from Renault et al. [32] and Lowe [28].

4.2 Related work on checking Rabin automata

As mentioned in Sect. 1, when checking LTL properties for probabilistic systems, the automaton needs to be deterministic [3]. Chatterjee et al. [9] 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 non-generalized variant of deterministic Rabin automata. They also present improvements for computing a winning strategy in LTL(F,G) games by using a fixpoint algorithm for general-

ized Rabin pairs. Our algorithm is different in that it operates on-the-fly and in a multi-core setting.

Wijs [35] recently presented an 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 [18,20]. 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.

4.3 Related work on co-Büchi emptiness checks

Livelock detection algorithms such as the DFS_{fifo} algorithm [17,25] search the state-space for cycles that avoid "progress transitions". If we label all progress transitions with ❶, such a search amount to the emptiness check of an automaton with acceptance $\mathsf{Fin}(❶)$, i.e. a transition-based co-Büchi automata. DFS_{fifo} detects non-progress cycles by performing a DFS that is restricted to non-progress transitions, but that remembers the set F of all states that are the destination of an ignored progress transition. Once the "restricted" DFS terminates, it is started again from one of the F sets. The similarity with Algorithm 3 should not be a surprise: co-Büchi acceptance is just a generalized Rabin pair without any Inf set. In our case, the DFS has to track (partial) strongly connected components simply to ensure we visit each Fin set infinitely often.

This similarity suggests another application of our algorithm: detection of non-progress cycles under fairness. For instance assume a communicating-process model where two clients C_1, C_2 are in contact with three servers S_3, S_4, S_5. We want to know whether any of the client can livelock under the fairness assumption that the servers progress infinitely often. If we denote by $T_1, T_2, \ldots, T_5$ the set of progress transitions of each of these five processes, the problem amounts to the emptiness check of the state-space with the generalized Rabin condition $(\mathsf{Fin}(T_1) \wedge \mathsf{Inf}(T_3) \wedge \mathsf{Inf}(T_4) \wedge \mathsf{Inf}(T_5)) \vee (\mathsf{Fin}(T_2) \wedge \mathsf{Inf}(T_3) \wedge \mathsf{Inf}(T_4) \wedge \mathsf{Inf}(T_5))$.

4.4 Related work on checking different automata

Emerson and Lei [14] 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. [13] 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. [27] 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 512 GB memory available. We performed all experiments using 16 cores.

5.1.1 Implementation

References to all the tools involved in the experiments are given in Table 2. The TGRA checking algorithm is implemented in a development version of the LTSmin toolset. Additionally, we used several external tools and libraries for generating and parsing the automata:

- We used `Rabinizer 3` to generate deterministic transition-based generalized Rabin automata.
- The tool `LTL3DRA` was also used to generate deterministic automata with transition-based generalized Rabin acceptance. However, `LTL3DRA` only supports a subset of LTL, called LTL\GUX in [5], 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.
- We used some tools from Spot: `ltl2tgba` for generating TGBAs, and `autfilt` for converting automata with other accepting conditions into TGRAs or TFLAs.
- We used `LTL3TELA` to generate non-deterministic 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 `cpphoafparser` library to parse a HOA automaton [1] and create an internal representation for LTSmin.

The commands used to generate automata from LTL are summarized in Table 3.

We used the algorithm from Bloemen et al. [8] to model check TGBAs and TFLAs, and used the algorithm presented

Table 2 Tools used in the experimental evaluation

Tool	Version	Ref.	Web page
Spot (`ltl2tgba`, `autfilt`)	2.5.2	[12]	https://spot.lrde.epita.fr/
`Rabinizer`	3.1	[15,22]	https://www7.in.tum.de/~kretinsk/rabinizer3.html
`LTL3DRA`	0.2.6	[2]	https://sourceforge.net/projects/ltl3dra/
`LTL3TELA`	1.1.1		https://github.com/jurajmajor/ltl3tela/
`cpphoafparser`	0.99.2	[1]	http://automata.tools/hoa/cpphoafparser/
LTSMIN	3.0	[21]	http://ltsmin.utwente.nl/

Table 3 Tool configurations used for generating an automaton from an LTL formula φ

Key	Automaton type	Command
`Rabinizer 3`	TGRA	`java -jar Rabinizer3.jar -format=hoa -in=formula -out=std -silent` φ
`LTL3DRA`	TGRA	`ltl3dra -f` φ
`LTL3TELA`	TGRA	`ltl3tela -f` φ `| autfilt --generalized-rabin`
`Fin-less`	TFLA	`ltl3tela -f` φ `| autfilt --remove-fin`
`TGBA`	TGBA	`ltl2tgba -f` φ

in this paper for checking TGRAs, both are implemented in LTSMIN. The algorithms make use of LTSMIN's internal shared hash tables [26], and the same randomized successor distribution method is used throughout. The shared hash table is initialized to store up to 2^{28} states.

5.1.2 Experiments

The models and properties for our experiments were obtained from the 2015 Model Checking Contest [23], and from models from the BEEM benchmark [31] for which a set of non-trivial, randomly generated LTL formulas is available [4].

The MCC model set was restricted to those that do not describe obligation properties because using non-Büchi acceptance cannot help producing smaller automata on this class [29]. This selection is further reduced by selecting only the instances where the 'TGRA generators' (`LTL3DRA`, `Rabinizer 3` and `LTL3TELA`) 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 min. We remark that this selection is in favour 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. We take the mean of the results to mitigate randomness introduced by the multi-core search procedure. The configurations were: `ltl2tgba` using the TGBA checking algorithm, `LTL3DRA`, `Rabinizer 3`, and `LTL3TELA` translated to TGRA, where the latter three cases used the TGRA checking algorithm introduced in this paper. Every task was run with a timeout of 10 min. In total, the MCC experiments took approximately 5 days to complete.

The second set of experiments consists of 3200 formulas over 16 models extracted from the BEEM database [4,31]. For each model, 100 verified (empty product) and 100 violated (non-empty product) formulas were generated [4]. We refer to this set of experiments by the BEEM experiments.

For this set of experiments, we generated the following automata: TGBAs from the `ltl2tgba` tool, TGRAs from `LTL3DRA`, `Rabinizer 3`, and `LTL3TELA`, and we also generated Fin-less automata by first generating TGRAs with `LTL3TELA` and using Spot's `autfilt` to convert these into TFLAs. We only considered TFLAs that contain a disjunction of multiple TGBA acceptance sets, hence we filtered out all TFLAs generated from a TGRA with a single Rabin pair. We used the (extended) TGBA checking implementation to check the TGBAs and TFLAs, and we used the TGRA checking implementation to check the TGRAs and also the TFLAs. Every task was ran once with a timeout of 10 min. In total, the BEEM experiments also took approximately 5 days to complete.

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

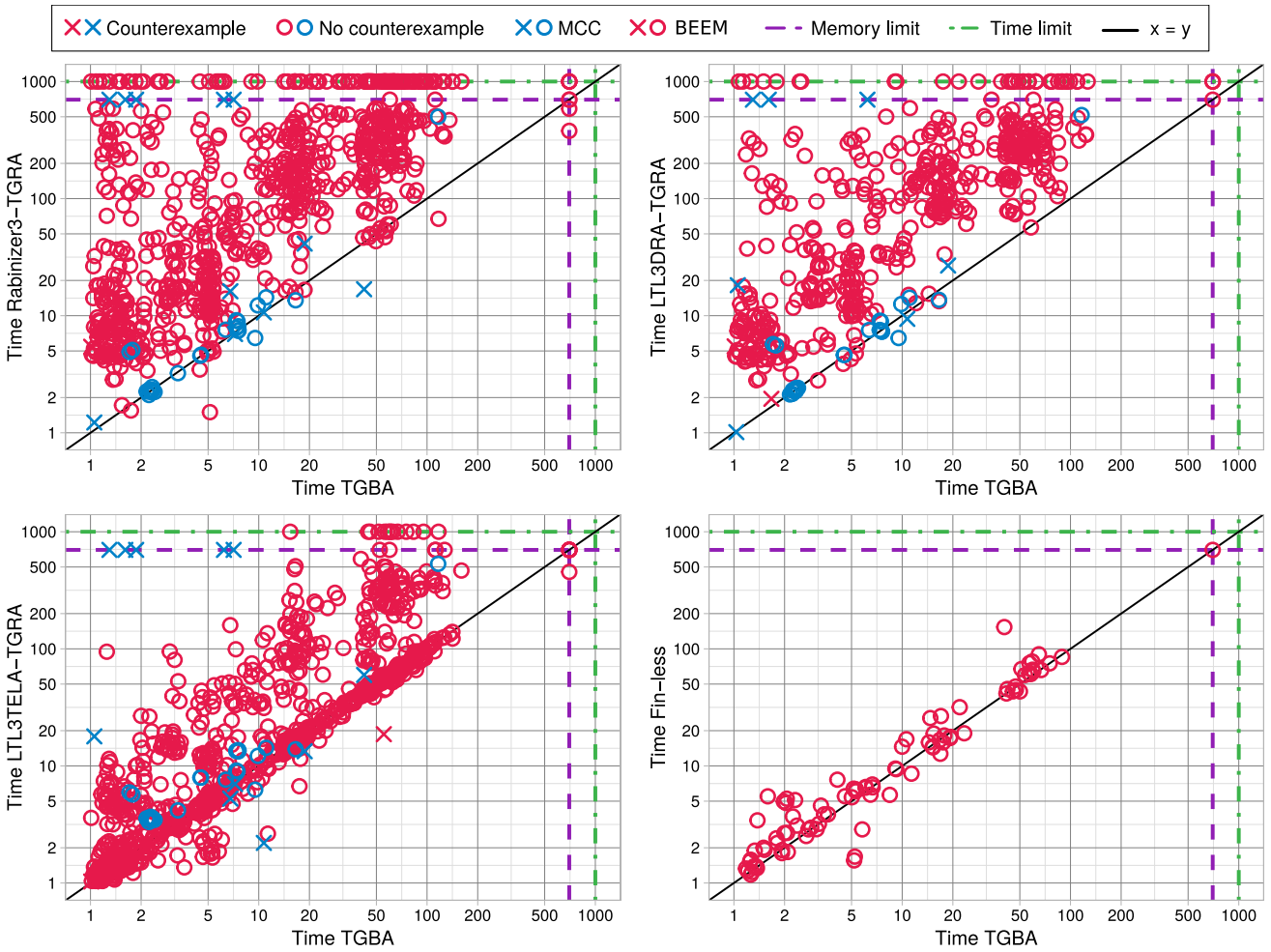


Fig. 6 Time (in seconds) comparisons of the TGBA (x-axis) and the TGRA and TFLA emptiness checks (y-axis), for various LTL to TGRA and TFLA translations. Each point represents the time to perform an emptiness check using 16 cores. The TGRA and TFLA algorithms performed faster for instances below the `x=y` line (colour figure online)

5.2 Main results

The main results of the experiments are presented in Fig. 6 and are summarized in Table 4. We also depict specific characteristics of the experiments in Table 5, Figs. 7, and 8. 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 TGRP and TFLA checking experiments are shown on the y-axis. 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 some errors in the experiments. There were a number of instances that resulted in a memory error, meaning that the data structures became too large to fit in the memory during the model checking procedure. These errors only occurred for the TGRA checks and were presumably caused by the additional allocation of the `Fstates` data structure. There are also several instances that resulted in timeouts for some of the configurations.

We first analyse the performance of the TGRA checking algorithm and how the different TGRA generators influence the results, then we consider the translation to Fin-less automata, and finally we discuss some additional results.

5.2.1 Comparison between the three TGRAs and TGBA

We first take a look at Fig. 6 and Table 4. It becomes clear that for most experiments it is more beneficial to model check using TGBAs compared to TGRAs (using our algorithm). In the scatter-plot we observe that in some cases the TGBA checking algorithm is more than two orders of magnitude faster. That being said, there are instances where the TGRA checking algorithm is faster, especially for the TGRAs generated by `LTL3TELA`.

Table 4 Comparison of the geometric mean execution times (in seconds)

	MCC				BEEM			
	`LTL3TELA`	`LTL3DRA`	`Rabinizer 3`	`TGBA`	`LTL3TELA`	`LTL3DRA`	`Rabinizer 3`	`TGBA`
Counterexample	1.51 (1.89)	1.69 (2.12)	1.03 (1.29)	0.80	0.12 (1.14)	0.26 (2.41)	0.43 (4.04)	0.11
No counterexample	7.68 (1.60)	5.67 (1.18)	5.64 (1.17)	4.81	8.11 (1.44)	34.25 (6.06)	33.98 (6.02)	5.65
Total	4.47 (1.69)	3.79 (1.43)	3.20 (1.21)	2.64	0.66 (1.25)	1.83 (3.49)	2.48 (4.74)	0.52

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* TGRA and TGBA configurations without running out of memory or time (39 in total for the MCC experiments, with 13 counterexamples, and 1167 in total for the BEEM experiments, with 699 counterexamples)

Table 5 Geometric mean sizes of the automata and products

	$\|Aut\|$	$\|Pairs\|$	$\|States\|$	$\|Trans\|$
MCC				
`LTL3TELA`	1.00	1.53	0.71×10^6	4.26×10^6
`LTL3DRA`	1.00	1.00	0.47×10^6	2.78×10^6
`Rabinizer 3`	1.38	1.00	0.47×10^6	2.78×10^6
`TGBA`	1.31	1.00	0.51×10^6	3.11×10^6
BEEM				
`LTL3TELA`	2.85	1.07	4.02×10^6	13.86×10^6
`LTL3DRA`	4.00	1.27	10.61×10^6	42.17×10^6
`Rabinizer 3`	3.56	1.24	10.29×10^6	40.86×10^6
`TGBA`	3.23	1.00	3.80×10^6	13.74×10^6

$|Aut|$ 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 without a counterexample and that were checked in *all* TGRA and TGBA configurations without running out of memory or time (26 in total for MCC and 468 in total for BEEM)

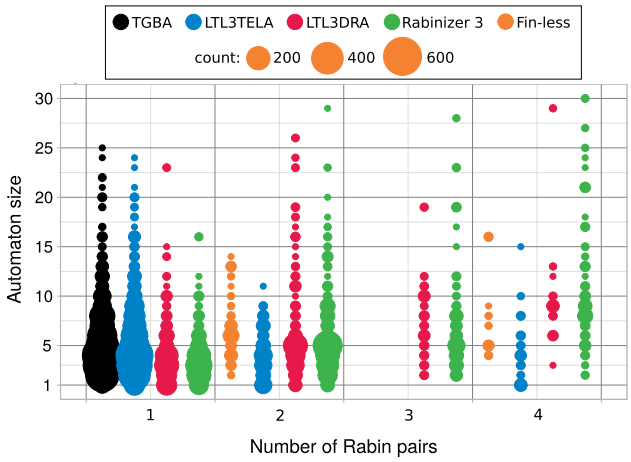


Fig. 7 Distribution of automaton sizes for TGBAs, TGRAs and TFLAs. The x-axis shows the number of pairs (or Inf sets for TFLAs) in the automaton and the y-axis shows the size of the automaton. The size of the circle denotes how many automata belong to the same class, i.e. an automaton having x Rabin pairs and y states (colour figure online)

We notice that there is quite some difference between the results from the MCC experiments and the BEEM experiments. We argue that the LTL formulas for the MCC experiments are more realistic, but also less complex. The effect is that the produced TGRAs are simpler in structure and are therefore similar to the TGBAs. The results show this as well; in many MCC experiments the difference between the TGRA and TGBA emptiness checks is relatively small (compared to the BEEM experiments).

The BEEM experiments are generated randomly and therefore do not give an accurate representation of reality. However, the generated formulas are generally more complex and cause the TGRAs and TGBAs to differ a lot more than they do in the MCC experiments. We can see that these more complex TGRAs (and TGBAs) cause the TGRA checking algorithm to perform significantly worse, when compared with the MCC experiments. This implies that our algorithm for checking TGRAs performs relatively poorly for complex LTL formulas.

We can also observe that there are significant differences between the TGRAs generated by the three TGRA generators. The (non-deterministic) TGRAs from the `LTL3TELA` generator leads to the most favourable results. Both deterministic TGRA generators (especially `Rabinizer 3`) generate automata for which the model checking algorithm performs significantly worse. Hence we argue that in our setting we do not benefit much from deterministic TGRAs compared to non-deterministic ones.

5.2.2 Analysing the TGRAs

We analyse how the different TGRA generators influence the automaton sizes in Table 5. For the MCC experiments, we observe that the automaton sizes for `LTL3TELA` and `LTL3DRA` are on average smaller than a TGBA. Moreover, the product automaton for both deterministic automata are smaller than that of a TGBA. The product size of the `LTL3TELA` TGRAs are larger than for the other automata, and in Table 4 we see that `LTL3TELA` performs the worst on the MCC experiments.

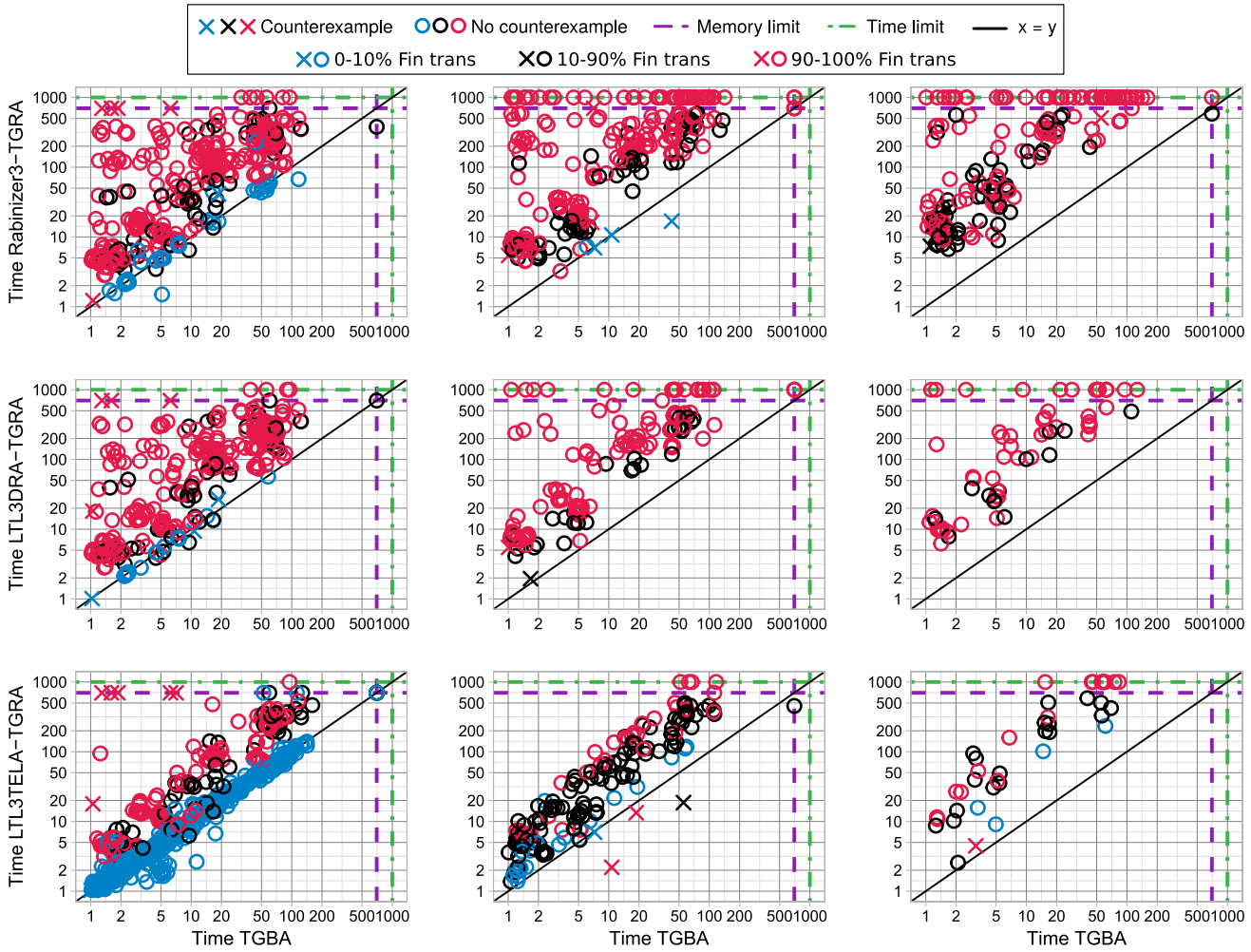


Fig. 8 Time (in seconds) comparisons of the emptiness checks in the same format as Fig. 6, but here the plots in the left column only contain TGRAs that have a single Rabin pair. The middle column only shows results for TGRAs with exactly two Rabin pairs, and the right column shows results for the remaining automata, i.e. automata with more than two Rabin pairs (colour figure online)

However, the automata for the BEEM experiments are quite different from the MCC ones. Here, both deterministic TGRA generators generate significantly larger automata and product automata (a factor of three larger) compared to both `LTL3TELA` and TGBA. Here it seems that `LTL3TELA` leads to more suitable automata for emptiness checking TGRAs. As can be observed in Table 4, the product automaton size seems to correlate well with the TGRA emptiness check performance, as `LTL3TELA` is clearly the favourite of the three.

In Fig. 7 we show how the number of Rabin pairs correlates with the size of the automaton. Note that this figure only shows the majority of the generated automata; there are automata with more than 30 states and more than 4 Rabin pairs. It is clear that most of the automata contain a single Rabin pair and have less than five states. Most of `LTL3TELA` TGRAs seem to have the same number of states as the TGBAs, and `LTL3TELA` also produces small automata for multiple Rabin pairs. The TGRAs from both deterministic TGRA generators have similar characteristics. There are fewer automata with a single Rabin pair, and the automaton sizes are larger when there are multiple Rabin pairs. This makes sense as multiple Rabin pairs indicate that the LTL formula is more complex, which causes the deterministic TGRA generators to require more states to remain deterministic.

5.2.3 Influence of the number of Rabin pairs and Fin transitions

In Fig. 8 we split up the results from Fig. 6 and show how the performance is affected when the TGRAs contain more Rabin pairs. We can see that in general, the TGRA checking algorithm performs relatively worse for automata with multiple Rabin pairs compared to TGRAs with a single pair. We can also see that most of the timeouts occur for two or

Table 6 Comparison of the geometric mean execution times (in seconds) in the same format as Table 4

	LTL3TELA	Fin-less	TGBA
Counterexample	0.20 (1.55)	0.13 (0.96)	0.13
No counterexample	15.02 (3.70)	4.52 (1.11)	4.06
Total	2.11 (2.49)	0.88 (1.04)	0.85

We only used data from experiments that were checked in the TFLA, LTL3TELA-TGRA, and TGBA configurations without running out of memory or time (195 BEEM experiments in total, with 89 counterexamples)

Table 7 Geometric mean sizes of the automata and products in the same format as Table 5

	$\|Aut\|$	$\|Pairs\|$	$\|States\|$	$\|Trans\|$
LTL3TELA	3.31	2.21	5.22×10^6	11.30×10^6
Fin-less	5.79	2.17	2.74×10^6	9.06×10^6
TGBA	3.76	1.00	2.37×10^6	7.43×10^6

Here, $|Pairs|$ denotes the number of Inf sets in the TFLA. We only used data from experiments without a counterexample and that were checked in the TFLA, LTL3TELA-TGRA, and TGBA configurations without running out of memory or time (106 in total for BEEM)

more Rabin pairs. This result is not surprising, as checking a TGRA with two pairs involves traversing the entire state-space of the product automaton twice, assuming there is no counterexample.

We found that for most product automata (with TGRAs), the number of Fin transitions is either a very small or very large proportion of the total number of transitions. In Fig. 8 we coloured three classes: automata for which the number of Fin transitions is 0–10% of the total number of transitions (blue), automata for which this ratio is between 10 and 90% (black), and automata in which 90% or more transitions are Fin transitions (red).

It becomes clear that this aspect greatly influences the TGRA model checking performance. Its performance is actually comparable to TGBA for a small fraction of Fin transitions. Furthermore, there are almost no instances where the TGRA checking algorithm performs well for cases with many Fin transitions.

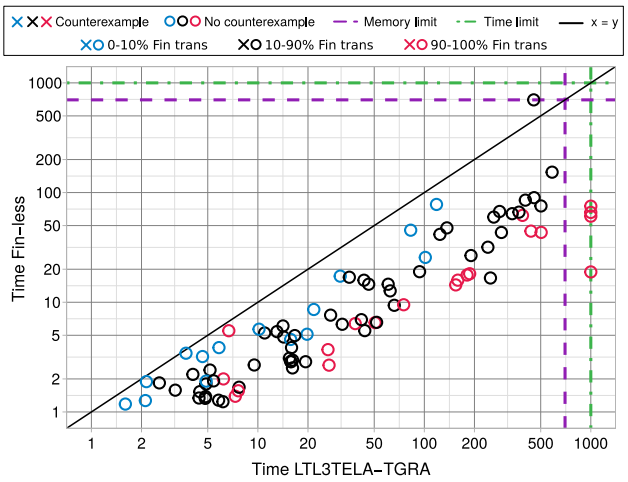


Fig. 9 Time (in seconds) comparisons of the emptiness checks in the same format as Fig. 8, but here the x-axis represents the time required for LTL3TELA-TGRA and the y-axis shows the time needed for TFLA (colour figure online)

5.3 Fin-less results

We now discuss how the emptiness check for Fin-less automata compares with the check for TGBAs. As noted before, we only considered TFLAs with at least two Inf sets (and hence are obtained from TFLAs with at least two Rabin pairs). In Fig. 6 we can see that the performance of the TFLA checking algorithm is comparable to that of the TGBA checking one. While there is only a small difference in the emptiness procedure, the TFLAs are quite different from TGBAs. In Table 6 we summarize the performance results and indeed see that there is only a small difference between the two emptiness checks.

It could be expected that a TFLA emptiness check performs worse than the corresponding TGBA one as there is a slightly greater overhead on verifying the acceptance condition. Also, as a result of their construction process, TFLAs are non-deterministic and contain (linearly, in the number of TGRPs) more states than the original TGRAs. A reason for why a TFLA might be checked faster than a TGBA is that a counterexample may be detected earlier as there are multiple acceptance sets that can be checked simultaneously. Another reason is because the acceptance condition is more lenient than a TGBA. This allows a TFLA generator to construct more complex automata without significantly impacting the emptiness checking algorithm.

In Table 7 we see that a TFLA has significantly more states than the corresponding TGBA, which holds as well for its product automaton. The effect of the TFLA construction procedure is visible in Fig. 7. TFLAs are constructed from the TGRAs generated by LTL3TELA. One can observe that as a result of this construction, the number of states in the automata shifts up when comparing the TFLA with the corresponding TGRA. Table 7 also shows that TFLAs are significantly larger than TGRAs. However, even though the TFLAs are larger and contain more non-determinism than TGRAs, the product automaton is on average smaller.

Figure 9 shows how the performance of the TFLA checking algorithm compares with that for the TGRA version of the automata. The TFLA version is significantly faster in practically all instances. The difference is especially large in cases where the TGRA state-space consists of many Fin transitions. We can argue that it is more advantageous to first

construct a TFLA from a TGRA and check the TFLA, than to check the TGRA directly (using our algorithm).

5.4 Additional results

5.4.1 Checking TGRPs in parallel

In preliminary experiments we experimented with checking TGRPs in parallel, as suggested in Sect. 3.1. 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 to check the TGRPs sequentially.

5.4.2 Scalability

Our existing TGBA checking algorithm [7,8] 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 comparable to a TGBA emptiness check in some scenarios. We analysed various aspects of the TGRAs and how these affect the model checking performance. The TGRA checking algorithm seems most beneficial in instances where only a small fraction of the state-space is part of a Fin set.

We also introduced Fin-less automata, which can be constructed from TGRAs. We showed that the emptiness check for such a TFLA can be implemented in a TGBA emptiness implementation without a large performance impact. Experiments showed that emptiness checks for TFLAs and TGBAs perform comparably. From our findings we conclude that it is more beneficial to construct a Fin-less automaton and check that for emptiness than to check the TGRA directly.

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. Future work is also needed to check whether our algorithm is useful in settings other than model checking (e.g. equivalence checking of automata), where synchronization that results in a large number of Fin transitions are perhaps less likely. Another direction for future work is to investigate a variation of the proposed algorithm to check fairness or Streett automata.

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

References

1. Babiak, T., Blahoudek, F., Duret-Lutz, A., Klein, A. Křetínský, J. Müller, D., Parker, D., Strejček, J.: The Hanoi omega-automata format. In: Proceedings of CAV'15, vol. 9206 of LNCS, pp. 479–486. Springer (2015)
2. Babiak, T., Blahoudek, F., Křetínský, M., Strejček, J.: Effective translation of LTL to deterministic Rabin automata: beyond the (F,G)-fragment. In: Proceedings of ATVA'13, pp. 24–39. Springer (2013)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
4. Ben Salem, A., Duret-Lutz, A., Kordon, F., Thierry-Mieg, Y.: Symbolic model checking of stutter-invariant properties using generalized testing automata. In: Tools and Algorithms for the Construction and Analysis of Systems—20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, vol. 8413 of LNCS, pp. 440–454. Springer (2014)
5. Blahoudek, F., Křetínský, M., Strejček, J.: Comparison of LTL to deterministic Rabin automata translators. In: Proceedings of LPAR-19, pp. 164–172. Springer (2013)
6. Bloemen, V., Duret-Lutz, A., van de Pol, J.: Explicit state model checking with generalized Büchi and Rabin automata. In: Proceedings of SPIN'17, pp. 50–59. ACM (2017)
7. Bloemen, V., Laarman, A., van de Pol, J.: Multi-core on-the-fly SCC decomposition. In: Proceedings of PPoPP'16, pp. 8:1–8:12. ACM (2016)
8. Bloemen, V., van de Pol, J.: Multi-core SCC-based LTL model checking. In: Proceedings of HVC'16, pp. 18–33. Springer (2016)
9. Chatterjee, K., Gaiser, A., Křetínský, J.: Automata with generalized Rabin pairs for probabilistic model checking and LTL synthesis. In: Proceedings of CAV'13, pp. 559–575. Springer (2013)

10. Couvreur, J.-M., Duret-Lutz, A., Poitrenaud, D.: On-the-fly emptiness checks for generalized Büchi automata. In: Proceedings of SPIN'05, vol. 3639 of LNCS, pp. 143–158. Springer (2005)
11. Dijkstra, E.W.: 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)
12. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0—a framework for LTL and ω-automata manipulation. In: Proceedings of ATVA'16, vol. 9938 of LNCS, pp. 122–129. Springer (2016)
13. Duret-Lutz, A., Poitrenaud, D., Couvreur, J.-M.: On-the-fly emptiness check of transition-based Streett automata. In: Proceedings of ATVA'09, vol. 5799 of LNCS, pp. 213–227. Springer (2009)
14. Emerson, E.A., Lei, C.-L.: Modalities for model checking (extended abstract): branching time strikes back. In: Proceedings of POPL'85, pp. 84–96. ACM (1985)
15. Esparza, J., Křetínský, J., Sickert, S.: From LTL to deterministic automata. Form. Methods Syst. Des. **49**(3), 1–53 (2016)
16. Evangelista, S., Laarman, A., Petrucci, L., van de pol, J.: Improved multi-core nested depth-first Search. In: Proceedings of ATVA'12, vol. 7561 of LNCS, pp. 269–283. Springer (2012)
17. Faragó, D., Schmitt, P.H.: Improving non-progress cycle checks. In: Proceedings of the 16th International SPIN Workshop, pp. 50–67. Springer (2009)
18. Filippidis, I., Holzmann, G.J.: An improvement of the Piggyback algorithm for parallel model checking. In: Proceedings of SPIN'14, pp. 48–57. ACM (2014)
19. Holzmann, G., Joshi, R., Groce, A.: Swarm verification techniques. IEEE Trans. Softw. Eng. **37**(6), 845–857 (2011)
20. Holzmann, G.J.: Parallelizing the spin model checker. In: Proceedings of SPIN'12, vol. 7385 of LNCS, pp. 155–171. Springer (2012)
21. Kant, G., Laarman, A., Meijer, J., van de Pol, J., Blom, S., van Dijk, T.: LTSmin: high-performance language-independent model checking. In: Proceedings of TACAS'15, vol. 9035 of LNCS, pp. 692–707. Springer (2015)
22. Komárková, Z., Křetínský, J.: Rabinizer 3: Safraless translation of LTL to small deterministic automata. In: Proceedings of ATVA'14, pp. 235–241. Springer (2014)
23. Kordon, F., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Linard, A., Beccuti, M., Hamez, A., Lopez-Bobeda, E., Jezequel, L., Meijer, J., Paviot-Adet, E., Rodriguez, C., Rohr, C., Srba, J., Thierry-Mieg, Y., Wolf, K.: Complete Results for the 2015 Edition of the Model Checking Contest (2015). http://mcc.lip6.fr/2015/results.php
24. Křetínský, J., Meggendorfer, T., Sickert, S.: Rabinizer 4: from LTL to your favourite deterministic automaton. In: Proceedings of CAV'18, July 2018. (**to appear**)
25. Laarman, A., Faragó, D.: Improved on-the-fly livelock detection. In: Proceedings of the 5th NASA Formal Methods symposium, pp. 32–47. Springer (2013)
26. Laarman, A., van de Pol, J., Weber, M.: Multi-core LTSmin: marrying modularity and scalability. In: Proceedings of NFM'11, Lecture Notes in Computer Science, pp. 506–511. Springer (2011)
27. Liu, Y., Sun, J., Dong, J.: Scalable multi-core model checking fairness enhanced systems. In: Proceedings of ICFEM'09, vol. 5885 of LNCS, pp. 426–445. Springer (2009)
28. Lowe, G.: Concurrent depth-first search algorithms based on Tarjan's algorithm. Int. J. Softw. Tools Technol. Transf. **18**, 1–19 (2015)
29. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: Proceedings of PODC'87, pp. 205–205. ACM (1987)
30. Müller, D., Sickert, S.: LTL to deterministic Emerson–Lei automata. In: Proceedings of GandALF'17, vol. 256 of EPTCS, pp. 180–194, Sept. 2017
31. Pelánek, R.: BEEM: Benchmarks for Explicit Model Checkers, pp. 263–267. Springer, Berlin (2007)
32. Renault, E., Duret-Lutz, A., Kordon, F., Poitrenaud, D.: Variations on parallel explicit emptiness checks for generalized Büchi automata. Int. J. Softw. Tools Technol. Transf. **19**, 1–21 (2016)
33. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Proceedings of TLTL3HOoACAS'05, vol. 3440 of LNCS, pp. 174–190. Springer (2005)
34. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: Proceedings of LICS'86, pp. 322–331. IEEE Computer Society (1986)
35. Wijs, A.: BFS-based model checking of linear-time properties with an application on GPUs. In: Proceedings of CAV'16, pp. 472–493. Springer (2016)