Generic Emptiness Check for Fun and Profit

Christel Baier[1], František Blahoudek[2], Alexandre Duret-Lutz[3], Joachim Klein[1], David Müller[1], and Jan Strejček[4]

[1] Technische Universität Dresden, Germany
{`christel.baier`, `joachim.klein`, `david.mueller2`}`@tu-dresden.de`
[2] University of Mons, Belgium, `xblahoud@fi.muni.cz`
[3] LRDE, EPITA, Le Kremlin-Bicêtre, France, `adl@lrde.epita.fr`
[4] Masaryk University, Brno, Czech Republic, `strejcek@fi.muni.cz`

Note: This version of the paper includes appendices that are not published in the ATVA'19 proceedings.

Abstract. We present a new algorithm for checking the emptiness of ω-automata with an Emerson-Lei acceptance condition (i.e., a positive Boolean formula over sets of states or transitions that must be visited infinitely or finitely often). The algorithm can also solve the model checking problem of probabilistic positiveness of MDP under a property given as a deterministic Emerson-Lei automaton. Although both these problems are known to be NP-complete and our algorithm is exponential in general, it runs in polynomial time for simpler acceptance conditions like generalized Rabin, Streett, or parity. In fact, the algorithm provides a unifying view on emptiness checks for these simpler automata classes. We have implemented the algorithm in Spot and PRISM and our experiments show improved performance over previous solutions.

1 Let's Play

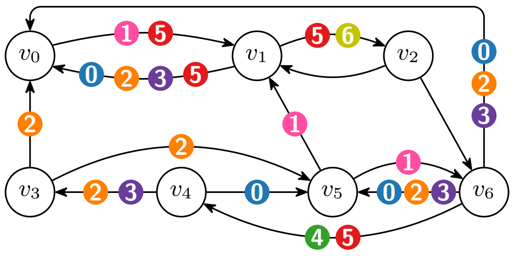


Fig. 1. Find a cycle satisfying the condition given in the text.

Consider the graph in Figure 1. Can you find a cycle satisfying the condition $\Big(\big(\mathsf{Fin}(⓪)\wedge\mathsf{Inf}(①)\big)\vee\big(\mathsf{Fin}(②)\wedge\mathsf{Inf}(③)\big)\Big)\wedge\big(\mathsf{Fin}(④)\vee\mathsf{Inf}(⑤)\big)\wedge\big(\mathsf{Fin}(⑥)\vee\mathsf{Inf}(⑦)\big)$, where $\mathsf{Fin}(ⓘ)$ is satisfied iff the mark ⓘ is not on the cycle and $\mathsf{Inf}(ⓙ)$ is satisfied iff the mark ⓙ is on the cycle? Such a cycle exists.[5]

[5] As this problem can be understood by the average Sudoku player, more instances can be found at `https://adl.github.io/genem-exp/examples/` either to practice the algorithm by hand, or as an entertaining prophylaxis of Alzheimer's disease.

In this paper we introduce an algorithm deciding whether a given graph contains a cycle satisfying a given condition, we show that this algorithm generalizes some known algorithms dedicated to simpler subclasses of conditions, and we discuss other related work (Section 3). Further, we present two implemented applications that motivated the algorithm. First, we show that the algorithm decides whether a given ω-automaton with an Emerson-Lei acceptance condition represents an empty language (Section 4). Second, we show that the algorithm can be used in probabilistic model checking (Section 5). In both cases, experimental results indicate significant improvements over previous solutions.

2 Preliminaries

Marked graph. Intuitively, a *marked graph* G is a graph with edges marked by non-negative integers. We denote the set of all non-negative integers as $\mathbb{N}_0$ and we call its elements *marks*. Formally, G is a tuple $G = (V, E)$ of a finite set of *vertices* and a finite set of *edges* $E \subsetneq V \times 2^{\mathbb{N}_0} \times V$ where the set of *marks* M is finite for each edge (v_1, M, v_2). The set of all marked graphs is denoted by $\mathbf{G}$.

A *cycle* is a sequence of consecutive edges that starts and ends in the same vertex, i.e., $(v, M_0, v_1)(v_1, M_1, v_2)\ldots(v_n, M_n, v)$. The union $M_0 \cup M_1 \cup \ldots \cup M_n$ is the set of marks that *are on the cycle*. The marked graph $S = (V', E')$ is a *strongly connected component (SCC)* of $G = (V, E)$ if $V' \subseteq V$, $E' \subseteq E \cap (V' \times 2^{\mathbb{N}_0} \times V')$, and for each pair of distinct vertices $v, v' \in V'$ there is a sequence of consecutive edges from E' that connects v with v'. An SCC S is *maximal*, if there is no other SCC (V'', E'') of G such that $V' \subseteq V''$ and $E' \subsetneq E''$. Further, S is *non-trivial* if $E' \neq \emptyset$. Each non-trivial SCC has at least one cycle.

Acceptance condition. An *acceptance condition* over $\mathbb{N}_0$ is every formula φ built by the following grammar where m ranges over $\mathbb{N}_0$, and t and f stand for *true* and *false*, respectively. The set of all acceptance formulas is denoted by $\mathcal{C}$.

$$\varphi \ ::= \ \mathsf{t} \mid \mathsf{f} \mid \mathsf{Inf}(m) \mid \mathsf{Fin}(m) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi)$$

The concept of acceptance conditions comes from the theory of ω-automata (automata that read infinite words) and the grammar above is inspired by the definition introduced by the Hanoi Omega-Automata Format [2]. Table 1 presents the shape of formulas for some traditional ω-automata acceptance conditions. Note that marks can appear more than once in those formulas; for instance $(\mathsf{Fin}(0) \wedge \mathsf{Inf}(1)) \vee (\mathsf{Fin}(1) \wedge \mathsf{Inf}(0))$ is a Rabin acceptance formula.

Acceptance formulas are interpreted over sets of marks. We write $M \models \varphi$ if M *satisfies* φ with the relation $\models$ defined as follows.

$$\begin{array}{lll} M \models \mathsf{t} & M \models \mathsf{Inf}(m) \text{ iff } m \in M & M \models \varphi_1 \wedge \varphi_2 \text{ iff } M \models \varphi_1 \text{ and } M \models \varphi_2 \\ M \not\models \mathsf{f} & M \models \mathsf{Fin}(m) \text{ iff } m \notin M & M \models \varphi_1 \vee \varphi_2 \text{ iff } M \models \varphi_1 \text{ or } M \models \varphi_2 \end{array}$$

The following trivial simplifications, propagating f and t, are assumed to occur every time a formula is built or modified.

$$\begin{array}{llll} \mathsf{t} \vee \varphi = \mathsf{t} & \mathsf{f} \vee \varphi \ = \varphi & \mathsf{t} \wedge \varphi = \varphi & \mathsf{f} \wedge \varphi \ = \mathsf{f} \\ \varphi \vee \mathsf{t} = \mathsf{t} & \varphi \vee \mathsf{f} \ = \varphi & \varphi \wedge \mathsf{t} = \varphi & \varphi \wedge \mathsf{f} \ = \mathsf{f} \end{array} \tag{1}$$

Table 1. Shape of acceptance formulas corresponding to classical names. Here $m, m_0, m_1, ...$ correspond some acceptance marks, and J_i are sets of natural numbers.

Büchi	$\mathsf{Inf}(m)$
generalized Büchi	$\bigwedge_i \mathsf{Inf}(m_i)$
Fin-less [4]	any positive formula of $\mathsf{Inf}(...)$
co-Büchi	$\mathsf{Fin}(m)$
generalized co-Büchi	$\bigvee_i \mathsf{Fin}(m_i)$
Rabin	$\bigvee_i (\mathsf{Fin}(m_{2i}) \wedge \mathsf{Inf}(m_{2i+1}))$
generalized Rabin [27]	$\bigvee_i \Big(\mathsf{Fin}(m_i) \wedge \bigwedge_{j\in J_i} \mathsf{Inf}(m_j)\Big)$
Streett	$\bigwedge_i (\mathsf{Inf}(m_{2i}) \vee \mathsf{Fin}(m_{2i+1}))$
parity (min even)	$\mathsf{Inf}(m_0) \vee (\mathsf{Fin}(m_1) \wedge (\mathsf{Inf}(m_2) \vee (\mathsf{Fin}(m_3) \wedge \ldots)))$
hyper-Rabin [5]	$\bigvee_i \bigwedge_{j\in J_i} (\mathsf{Fin}(m_{2j}) \vee \mathsf{Inf}(m_{2j+1}))$
Emerson-Lei [19]	any positive formula of $\mathsf{Fin}(...)$ and $\mathsf{Inf}(...)$

The notation $\varphi[a \leftarrow b]$ means that all occurrences of the subformula a are replaced by b in φ. For instance, if $\varphi_1 = \big(\mathsf{Fin}(0)\wedge\mathsf{Inf}(1)\big)\vee\big(\mathsf{Fin}(2)\wedge\mathsf{Inf}(3)\big)$, then we have $\varphi_1[\mathsf{Inf}(1) \leftarrow \mathsf{f}] = \mathsf{Fin}(2) \wedge \mathsf{Inf}(3)$. We can also quantify the substitution over sets of marks. For example, $\varphi_1[\forall m \in \{0,1\} : \mathsf{Inf}(m) \leftarrow \mathsf{f}]$ yields $\mathsf{Fin}(2) \wedge \mathsf{Inf}(3)$ again while $\varphi_1[\forall m \in \{1,3\} : \mathsf{Inf}(m) \leftarrow \mathsf{f}] = \mathsf{f}$.

To distinguish from f and t in formula notation, we use $\mathbb{B} = \{\bot, \top\}$ to denote the set of Boolean constants in descriptions of algorithms.

The reason we do not express acceptance conditions over sets of transitions (like $\mathsf{Inf}(\mathcal{T})$), but use marks as indirection, is so that a condition may be specified even for a graph that is not fully known (e.g., constructed on-the-fly).

3 Algorithm

Algorithm 1 decides whether a given graph $G \in \mathbf{G}$ contains no cycle satisfying a given condition $\varphi \in \mathcal{C}$. Its main function is called IS_EMPTY, as a graph containing no such cycle can be seen as *empty* with respect to φ.

Each cycle in a graph G lies completely in some non-trivial SCC. Hence, the function IS_EMPTY(G,φ) decomposes the graph G into maximal SCCs using SCCS_OF(G) and runs IS_SCC_EMPTY(S,φ) for each non-trivial maximal SCC S. The graph is empty if and only if all its maximal SCCs are empty.

The function IS_SCC_EMPTY(S,φ) gets a non-trivial SCC. It first calls the function MARKS_OF(S) that returns the set M_{occur} of the marks that occurs on some edges in S and the set M_{every} of the marks that occurs on all edges in S. Formally, if E is the set of edges in S, then $M_{\text{occur}} = \bigcup_{(v,M,v')\in E} M$ and $M_{\text{every}} = \bigcap_{(v,M,v')\in E} M$.

This information is used to simplify φ on lines 8 and 9. For each mark m not occurring in S, we replace $\mathsf{Fin}(m)$ in φ by t and $\mathsf{Inf}(m)$ by f as all cycles in S satisfy $\mathsf{Fin}(m)$ and do not satisfy $\mathsf{Inf}(m)$. Similarly, for each mark m appearing on all edges of S, we replace $\mathsf{Fin}(m)$ in φ by f and $\mathsf{Inf}(m)$ by t as all cycles in S

Algorithm 1 **Input:** a graph $G \in \mathbf{G}$ and a condition $\varphi \in \mathcal{C}$
Output: IS_EMPTY(G, φ) returns $\bot$ if G contains a cycle satisfying φ, otherwise it returns $\top$

```
1  IS_EMPTY($G \in \mathbf{G}, \varphi \in \mathcal{C}$) $\to \mathbb{B}$:
2      foreach non-trivial $S \in$ SCCs_OF($G$) do
3          if $\neg$IS_SCC_EMPTY($S, \varphi$) then return $\bot$
4      return $\top$
5
6  IS_SCC_EMPTY($S \in \mathbf{G}, \varphi \in \mathcal{C}$) $\to \mathbb{B}$:
7      $(M_{\text{occur}}, M_{\text{every}}) \longleftarrow$ MARKS_OF($S$)
8      $\varphi \longleftarrow \varphi[\forall m \notin M_{\text{occur}} : \mathsf{Inf}(m) \leftarrow \mathsf{f}, \mathsf{Fin}(m) \leftarrow \mathsf{t}]$
9      $\varphi \longleftarrow \varphi[\forall m \in M_{\text{every}} : \mathsf{Inf}(m) \leftarrow \mathsf{t}, \mathsf{Fin}(m) \leftarrow \mathsf{f}]$
10     if $\varphi = \mathsf{t}$ then return $\bot$
11     if $\varphi = \mathsf{f}$ then return $\top$
12     if $\varphi[\forall m \in M_{\text{occur}} : \mathsf{Inf}(m) \leftarrow \mathsf{t}] = \mathsf{t}$ then return $\bot$
13     // Every minimal model of $\varphi$ contains some $\mathsf{Fin}(m)$ such that $m \in M_{\text{occur}}$
14     // We assume that $\varphi$ has the form $\varphi = \bigvee_{j \in J} \varphi_j$ where $\varphi_j$ are not disjunctions
15     foreach disjunct $\varphi_j$ of $\varphi$ do
16         if $\varphi_j = \bigwedge_{m \in M'} \mathsf{Fin}(m) \ \wedge \ \varphi'$ then
17             if $\neg$IS_EMPTY(REMOVE($S, M'$), $\varphi'$) then return $\bot$
18         else
19             pick some $m$ such that $\mathsf{Fin}(m)$ occurs in $\varphi_j$
20             if $\neg$IS_EMPTY(REMOVE($S, \{m\}$), $\varphi_j[\mathsf{Fin}(m) \leftarrow \mathsf{t}]$) then return $\bot$
21             if $\neg$IS_SCC_EMPTY($S, \varphi_j[\mathsf{Fin}(m) \leftarrow \mathsf{f}]$) then return $\bot$
22     return $\top$
```

satisfy $\mathsf{Inf}(m)$ and do not satisfy $\mathsf{Fin}(m)$. If the simplified formula φ is equivalent to t, then it is satisfied by all cycles of S and thus we return $\bot$ as the considered SCC is nonempty. Analogously, if the current φ is equivalent to f, then no cycle can satisfy it and thus we return $\top$ as the SCC is empty.

Since S is an SCC, there exists a cycle going through all its edges and visiting all marks in M_{occur}. Line 12 checks whether φ can be satisfied by such a cycle and returns $\bot$ if it is the case. If this check fails, we know that every cycle potentially satisfying the formula has to satisfy some $\mathsf{Fin}(m)$ subformula.

In the rest of the algorithm, we see φ as a disjunction $\varphi = \bigvee_{j \in J} \varphi_j$, where φ_j are not disjunctions. If φ is not a disjunction, we see the whole formula as a single disjunct. The SCC is empty with respect to φ if and only if it is empty with respect to each disjunct φ_j. Hence, the algorithm processes these disjuncts one by one and if the SCC is not empty with respect to some disjunct, we immediately return $\bot$.

The previous part of the algorithm implies that each disjunct φ_j can be satisfied only by a cycle satisfying some subformulas $\mathsf{Fin}(m)$. We can easily detect some of these subformulas if φ_j has the from $\varphi_j = \bigwedge_{m \in M'} \mathsf{Fin}(m) \ \wedge \ \varphi'$. Note that we see the formulas $\varphi_j = \bigwedge_{m \in M'} \mathsf{Fin}(m)$ and $\varphi_j = \mathsf{Fin}(m)$ as special cases of the conjunction and φ' stands for t in these cases. In practice, we consider the set M' on line 16 to be maximal in the sense that φ' is not a conjunction with another conjunct of the form $\mathsf{Fin}(m)$. When M' is identified, we remove all edges containing some marks of M' from S as these edges cannot be part of any cycle satisfying φ_j. The removal is done by the function REMOVE that takes a graph and a set of marks and returns the graph without all edges that contain some mark of the set. After this removal, we call IS_EMPTY to check whether the graph after the removal is empty with respect to φ'.

Finally, if φ_j is not of the form $\bigwedge_{m\in M'} \mathsf{Fin}(m) \;\wedge\; \varphi'$, line 19 nondeterministically chooses a subformula of the form $\mathsf{Fin}(m)$. Line 20 looks whether there is a cycle satisfying φ_j and containing no m mark (i.e., automatically satisfying $\mathsf{Fin}(m)$). If there is such a cycle, we return $\perp$ as S is not empty. Otherwise, line 21 checks whether there exists a cycle satisfying φ_j independently on $\mathsf{Fin}(m)$. More precisely, we replace $\mathsf{Fin}(m)$ in φ_j by f and check emptiness of S against the resulting condition. As this step does not remove any edges of S, we can use the function IS_SCC_EMPTY to check the emptiness.

3.1 Solving the Puzzle From the First Page

We illustrate how the Algorithm 1 works by running it step-by-step on the puzzle from the first page. In figures, we use gray boxes to enclose maximal SCCs. The red S or S_i next to each box is the name of this SCC used in the description. Trivial SCCs are indicated by a dashed border

To solve the puzzle, we call IS_EMPTY(G, φ) where G is the marked graph of Figure 1 and φ is the corresponding condition. The function SCCS_OF(G) on line 2 identifies one maximal SCC S which is the whole G, see Figure 2(a). As S is non-trivial, we call IS_SCC_EMPTY(S, φ). Lines 7–9 detect that ❼ does not occur in S ($\text{❼} \notin M_{\mathrm{occur}}$) and thus φ becomes $\varphi[\mathsf{Inf}(\text{❼}) \leftarrow \mathsf{f}]$ denoted by φ_1.

$$\varphi_1 = \Big(\big(\mathsf{Fin}(\text{⓿}) \wedge \mathsf{Inf}(\text{❶})\big) \vee \big(\mathsf{Fin}(\text{❷}) \wedge \mathsf{Inf}(\text{❸})\big)\Big) \wedge \big(\mathsf{Fin}(\text{❹}) \vee \mathsf{Inf}(\text{❺})\big) \wedge \mathsf{Fin}(\text{❻})$$

As φ_1 is neither t nor f, and it cannot be satisfied only by satisfaction of $\mathsf{Inf}(m)$ subformulas, the tests on lines 10–12 fail.

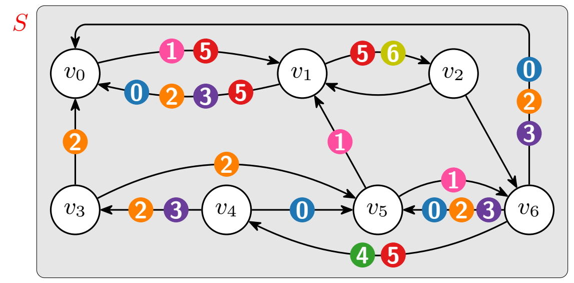
Further, φ_1 is a conjunction and thus the **foreach** loop on line 15 treats φ_1 as a single disjunct. As φ_1 matches the condition on line 16 for $M' = \{\text{❻}\}$, we remove all edges marked with ❻ from the graph and consider $\mathsf{Fin}(\text{❻})$ as satisfied. Now we call IS_EMPTY(G_1, φ_2) on graph G_1 depicted in Figure 2(b) with condition φ_2.

$$\varphi_2 = \Big(\big(\mathsf{Fin}(\text{⓿}) \wedge \mathsf{Inf}(\text{❶})\big) \vee \big(\mathsf{Fin}(\text{❷}) \wedge \mathsf{Inf}(\text{❸})\big)\Big) \wedge \big(\mathsf{Fin}(\text{❹}) \vee \mathsf{Inf}(\text{❺})\big)$$

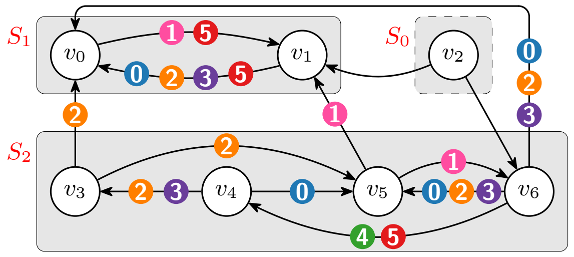
The function SCCS_OF(G_1) returns three maximal SCCs indicated in Figure 2(b), where only S_1 and S_2 are non-trivial. Let us assume that the **foreach** loop on line 2 processes S_1 first and calls IS_SCC_EMPTY(S_1, φ_2).

We can see that ❹ does not occur in S_1 and the condition is thus transformed into $\varphi_3 = \varphi_2[\mathsf{Fin}(\text{❹}) \leftarrow \mathsf{t}]$ on line 8.

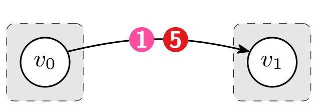
$$\varphi_3 = \big(\mathsf{Fin}(\text{⓿}) \wedge \mathsf{Inf}(\text{❶})\big) \vee \big(\mathsf{Fin}(\text{❷}) \wedge \mathsf{Inf}(\text{❸})\big)$$



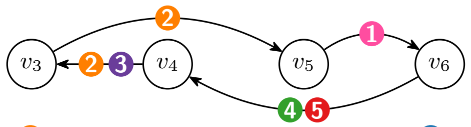
(a) Graph G from Figure 1, with one maximal SCC S.



(b) Decomposition of $G_1 = \textsc{remove}(S, \{❻\})$ into maximal SCCs.



(c) $G_2 = \textsc{remove}(S_1, \{⓿\}) = \textsc{remove}(S_1, \{❷\})$



(d) $G_3 = \textsc{remove}(S_2, \{⓿\})$

Fig. 2. Intermediate automata encountered while checking the emptiness of G.

The two disjuncts of φ_3 are checked independently in the **foreach** loop on line 15. Assume that the loop first considers the left disjunct and then the right one.

Note: In the proceedings of ATVA'19, the first item contains several typos. This version is fixed.

1. The first disjunct $\mathsf{Fin}(⓿) \wedge \mathsf{Inf}(❶)$ is matched by line 16 and thus we call $\textsc{is\_empty}(G_2, \mathsf{Inf}(❶))$, where G_2 is S_1 without edges marked with ⓿. You can find G_2 in Figure 2(c) which also reveals that G_2 has no non-trivial SCC. Therefore, $\textsc{is\_SCC\_empty}(G_2, \mathsf{Inf}(❶))$ immediately returns $\top$.
2. For the second disjunct $\mathsf{Fin}(❷) \wedge \mathsf{Inf}(❸)$, the computation follows the same scenario as S_1 without edges marked with ❷ is precisely the same graph G_2.

Overall, $\textsc{is\_SCC\_empty}(S_1, \varphi_2)$ returns $\top$ which corresponds to the fact that S_1 is empty with respect to φ_2.

The **foreach** loop on line 2 now processes the second non-trivial SCC S_2 and calls $\textsc{is\_SCC\_empty}(S_2, \varphi_2)$. In this case, lines 7–9 have no effect on φ_2 as all marks of the condition occur in S_2, but none of them occurs on every edge. The tests on lines 10–12 have no effect as well. Condition φ_2 is a conjunction and thus the **foreach** loop on line 15 treats it as a single disjunct. As φ_2 does not match the pattern on line 16, we reach line 19 and select some $\mathsf{Fin}(m)$ subformula of

Table 2. Complexity of Algorithm 1 for various types of acceptance formulas. $|V|$ and $|E|$ are the number of vertices and edges in the graph, n is the number of acceptance marks, $|\varphi|$ is the size of the formula, and f is the number of marks m_i that appears as $\mathsf{Fin}(m_i)$ in φ. We assume $|V| \leq |E|$, and that all marks between 0 and $n-1$ occur in φ (maybe multiple times, hence $f \leq n \leq |\varphi|$). We also show the subset of lines necessary to handle the given acceptance (when line 15 is missing, it is assumed that $\varphi_j = \varphi$).

acceptance type	complexity	range of lines needed	similar to
Büchi	$\mathcal{O}(\lvert E\rvert)$	1–8, 11–12	[22]
generalized Büchi	$\mathcal{O}(n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert)$	1–8, 11–12	[11, 37]
Fin-Less	$\mathcal{O}(n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert)$	1–8, 11–12	
Rabin	$\mathcal{O}(n \cdot \lvert\varphi\rvert \cdot \lvert E\rvert)$	1–8, 11–17, 22	
generalized Rabin	$\mathcal{O}(n \cdot \lvert\varphi\rvert \cdot \lvert E\rvert)$	1–8, 11–17, 22	[8, 4]
Streett	$\mathcal{O}(f \cdot (n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert))$	1–8, 11–12, 16–17, 22	[19, 17, 32]
parity	$\mathcal{O}(f \cdot (n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert))$	1–8, 11–12, 16–17, 22	
hyper-Rabin	$\mathcal{O}(\lvert\varphi\rvert \cdot (n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert))$	1–8, 11–17, 22	[19]
Emerson-Lei	$\mathcal{O}\left(2^f \cdot n \cdot \lvert\varphi\rvert \cdot \lvert E\rvert\right)$	1–8, 11–12, 19–22	

φ_2. Assume that we pick mark ⓿ to be removed from the condition. Hence, we remove all edges marked with ⓿ from S_2 and call IS_EMPTY(G_3, φ_4), where G_3 is S_2 after the removal and φ_4 stands for $\varphi_4 = \varphi_2[\mathsf{Fin}(⓿) \leftarrow \mathsf{t}]$.

$$\varphi_4 = \Big(\mathsf{Inf}(❶) \vee \big(\mathsf{Fin}(❷) \wedge \mathsf{Inf}(❸)\big)\Big) \wedge \big(\mathsf{Fin}(❹) \vee \mathsf{Inf}(❺)\big)$$

The graph G_3 can be seen in Figure 2(d). As G_3 is a single maximal non-trivial SCC, we can proceed to IS_SCC_EMPTY(G_3, φ_4). Lines 7–9 do not modify the condition, but the test on line 12 finally succeeds because $\varphi_4[\mathsf{Inf}(❶) \leftarrow \mathsf{t}, \mathsf{Inf}(❸) \leftarrow \mathsf{t}, \mathsf{Inf}(❺) \leftarrow \mathsf{t}] = \mathsf{t}$. Hence, G_3 is not empty with respect to φ_4. The cycle satisfying φ_4 can be easily seen in Figure 2(d). The value $\perp$ is returned and propagated all the way back to IS_EMPTY(G, φ). This corresponds to the fact that G is nonempty with respect to φ.

3.2 Correctness, Complexity, and Related Work

The following theorem summarizes the statements proved in Appendix A (termination and correctness) and Appendix B (complexity).

Theorem 1. *Algorithm 1 always terminates, and it returns $\perp$ if and only if G contains a cycle satisfying φ. Furthermore, its time complexity satisfies the upper bounds given in Table 2.*

It should be noted that lines 9–10 are not necessary for correctness, and that only the lines shown in Table 2 are needed to handle the listed acceptance types. Restricting Algorithm 1 to the specified ranges of lines also give the algorithm a behavior and complexity *comparable* to known algorithms for the subclass (as indicated in the last column), even if the actual implementation differ.

While the emptiness check for graphs with Emerson-Lei acceptance is known to be NP-complete [19, Thm. 4.7],[6] our main goal was to write a *simple* algorithm that covers the full spectrum of Emerson-Lei acceptance formulas while retaining polynomial complexity for commonly used subclasses. Emerson and Lei [19, Sec. 4.2–4.3] give an algorithm for the emptiness-check for Streett and hyper-Rabin conditions that behaves exactly like ours on these classes. They argue that at the cost of an extra $\mathcal{O}\left(2^{|\varphi|}\right)$ factor, any acceptance conditions can be converted into hyper-Rabin by first converting it to disjunctive normal form (DNF), and then inserting some unsatisfiable Inf and Fin terms to match the hyper-Rabin form. Our algorithm tries to avoid this blind DNF transformation by first reducing the acceptance condition to the marks used in the SCC, and by only "distributing" Fin marks (via the recursive calls of lines 20–21).

Better algorithms exist for some of these subclasses [5, Table 2]. However, the proposed algorithm can be implemented to work on a graph G that is not known explicitly, but on which forward successors can be computed on-the-fly, as usually done in the model checking community. In this context, the SCC decomposition algorithm may need to store all vertices of G, but the entire algorithm can be run without storing the edges of G, saving a lot of memory. For instance one faster emptiness check for Streett [9] requires the full graph to be available, with knowledge of the predecessors of each state.

4 Application 1: Emptiness Check for ω-Automata

This section discuses the proposed algorithm in the context of ω-automata, namely how it solves the emptiness check for *transition-based Emerson-Lei automata (TELA)* and several use cases for it. A TELA $\mathcal{A}$ over an *alphabet* Σ can be seen as a marked graph where vertices are called states, one state ι is the *initial* state, and each edge (called *transition*) carries in addition to marks also a *label* $\ell \in \Sigma$; transitions are thus quadruples of the form (s_1, ℓ, M, s_2). The *language* of $\mathcal{A}$ is the set $L(\mathcal{A}) \subseteq \Sigma^\omega$ of infinite words w such that there exists an infinite sequence of consecutive transitions starting in ι whose composition of labels is equal to w and the set of marks that appear infinitely often in the sequence, satisfies the acceptance condition of the automaton.

Many algorithms for ω-automata need to decide whether $L(\mathcal{A})$ is empty or not. This can be reduced to locating a reachable cycle whose set of marks satisfies the acceptance condition. To decide emptiness of $L(\mathcal{A})$, we can safely ignore transition labels, thus Algorithm 1 solves our problem.

The ω-automata with Emerson-Lei acceptance[7] were introduced more than 30 years ago. However, tools supporting acceptance conditions more complex

[6] In Appendix B.1 you can find a proof for a stronger statement relating satisfiability of Boolean formulas and Emerson-Lei acceptance more closely.

[7] The original definition was *state-based*, which means that marks were on states and not transitions. As ω-automata with state-based acceptance can be easily converted to transition-based acceptance without changing the transition structure, we focus on transition-based ω-automata only.

than generalized Büchi were rare in the past. In recent years, we could see a blossoming of tools producing ω-automata with various acceptance conditions (Streett, Rabin, parity, generalized Rabin, and even Emerson-Lei) [25, 27, 1, 29, 8, 26, 18, 20, 35, 21, 30]. The need for interaction between tools producing and consuming ω-automata with various acceptance conditions has led to the introduction of the Hanoi Omega-Automata Format [2], where acceptance conditions are expressed using formulas as in Section 2. The existence of this format and tools supporting this format had a boosting impact on the research community.

Spot [18] is one such tool: it supports TELA and aims to offer useful ω-automata algorithms that work on any acceptance formula when possible.

4.1 Use Cases in Spot

TELA emptiness check is useful in Spot on several places. Here are four:

- Spot's `ltlcross` tool is used to cross-compare the automata produced by various tools translating LTL to TELA automata and to test these tools. Initially, it only supported generalized Büchi automata [15, Sec. 3.2], but was extended to support arbitrary acceptance formulas. To detect buggy automata, `ltlcross` checks emptiness of automata products.[8]
- Deciding whether a TELA $\mathcal{A}$ is *stutter-invariant* also boils down to the emptiness check of the intersection of some modified version of $\mathcal{A}$ with its complement. Michaud and Duret-Lutz [33] describes this algorithm for generalized Büchi acceptance, but they apply (and are implemented) for TELA with arbitrary acceptance formula.
- Deciding whether a TELA $\mathcal{A}$ describes an *obligation property* also requires intersection of some modified version of $\mathcal{A}$ with its complement. Dax et al. [13] describes this algorithm for state-based Büchi acceptance, but Spot's implementation works for TELA with arbitrary acceptance formula.
- Deciding whether an SCC is *inherently weak* (does not mix accepting and rejecting cycles) can be done with two emptiness checks: one with the original acceptance formula, and one with the complementary acceptance formula. If one of these emptiness checks is successful, the SCC is inherently weak.

4.2 Previous and New Emptiness Check Implementations

Algorithm 1 has been implemented in Spot and released in version 2.8. The implementation has the following additional optimizations:

- The enumeration of SCCs of G, on line 2 is done by a Dijsktra-based algorithm [14] that also records the sets M_{occur} and M_{every} for each SCC. As a consequence, the tests of lines 10–12 can be moved into this SCC enumeration algorithm, and that algorithm can be configured to stop as soon at it find an SCC for which line 12 (or 10) would return $\perp$. If one such SCC is found, the loop of line 2-3 is not even executed, avoiding some useless recursions into SCC using more complex conditions.

[8] In fact, Figure 1 comes from a product of a Rabin and a Streett automaton.

– The REMOVE function is not implemented as a function that returns a new automaton. Instead, the SCC enumeration algorithm accepts a set of marks that define which transitions should be disallowed in SCCs (this is implemented by keeping the destination of marked transitions in a set of secondary SCC roots, as in the *Fstate* of Bloemen et al. [4]). Doing so ensure that the algorithm can run in place, without making any copy of the automaton.
– Finally, the top-most call to IS_EMPTY first checks if the acceptance condition has the form $\bigwedge_{m\in M'} \mathsf{Fin}(m) \ \wedge \ \varphi'$. If yes, M' is passed to the initial SCC enumeration algorithm and the relevant transitions are disallowed from start.

Before Algorithm 1 was implemented, Spot only had emptiness-check algorithms for generalized Büchi acceptance [12, 37], and those were easily extended to deal with Fin-less acceptance. Basically, such an algorithm just has to enumerate the SCCs of the automaton, and check whether the set of marks occurring in some SCC satisfies the acceptance formula.

To perform emptiness checks of a TELA $\mathcal{A}$ using some Fin in its acceptance formula, old versions of Spot would first convert $\mathcal{A}$ into some Fin-less TELA $\mathcal{B}$, and then check the emptiness of $\mathcal{B}$. Spot employs four techniques for Fin-removal:

– The generic case is discussed by Bloemen et al. [4, Prop. 1] and Duret-Lutz [16, Sec. 4.5]. Essentially, a disjunctive normal form (DNF) of the acceptance condition φ is computed as an irredundant sum-of-product [34] for a BDD representing φ (v_1 encodes $\mathsf{Inf}(1)$ and $\bar{v}_1$ encodes $\mathsf{Fin}(1)$). $\mathcal{B}$ is obtained by cloning $\mathcal{A}$ once for each clause of the DNF. A clone for a clause ignores transitions with marks appearing in the Fin terms of the clause.
– A special construction similar to the DBA-realizability algorithm of Krishnan et al. [28] is used when the input is a Rabin automaton. It helps to avoid cloning some SCCs of $\mathcal{A}$ for some clauses in the previous approach.
– A Streett automaton with k clauses and $|V|$ states is transformed into a generalized Büchi automaton with k marks and $|V| \cdot (2^k + 1)$ states.
– Weak automata are converted into Büchi automata with identical graph.

One obvious disadvantage of going through this Fin-removal procedure is that it completely generates the automaton $\mathcal{B}$ before it can be checked for emptiness. This is usually wasting time in case the automaton is nonempty. It is also wasting space compared to Algorithm 1 which can be implemented in place without making any automaton copy (in Spot, the REMOVE function is implemented by recursively propagating a list of marks to filter in the SCC enumeration code). Finally, the first step of the Fin-removal procedure in the general case is to rewrite the acceptance formula into DNF, without any consideration for the actual automaton. Although this exponential cost is still present in Algorithm 1 (via the recursive calls of lines 20–21), the algorithm tries its best to avoid this costs by using information from the automaton to reduce the acceptance condition, by detecting acceptance conditions that can be solved linearly (lines 15–16), and in the worst case by using an exponential factor of $\mathcal{O}\left(2^f\right)$ instead of $\mathcal{O}\left(2^{|\varphi|}\right)$. (Recall that f is the number of distinct Fin marks in φ.)

Table 3. Average runtime (in milliseconds) of the old and current implementations of the generic emptiness checks of Spot, over 5 different datasets with n automata that can be empty or non-empty. Since arithmetic means (amean) are biased towards larger automata, geometric means (gmean) are also provided.

	non-empty					empty				
		`old_is_empty`		`is_empty`			`old_is_empty`		`is_empty`	
	n	amean	gmean	amean	gmean	n	amean	gmean	amean	gmean
random	44	1105.6	390.1	3.311	0.326	6	678.8	364.7	20.032	15.938
random-rep	43	254.2	186.3	2.980	0.306	7	91.1	57.1	18.732	16.595
Rabin	9	26.7	26.7	1.208	0.356	41	11.6	11.6	10.195	10.194
Streett	50	1.0	0.2	0.002	0.001	50	14.3	3.2	11.616	0.926
parity-like	50	1124.4	6.4	0.003	0.002	50	809.7	28.9	1.272	0.485
(all)	196	592.3	14.6	1.454	0.021	154	301.2	12.6	8.530	1.811

We report on a benchmark that shows how Algorithm 1 improved the runtime of TELA emptiness checks in Spot compared to the previous approach with the Fin-removal procedure.

We prepared 5 data sets containing a mix of empty and non-empty automata:

random Contains 100 000-state automata with Random acceptance conditions. Each acceptance condition uses 20 acceptance marks, and those marks occur exactly once (either as Fin or as Inf). These automata are generated using Spot's `randaut` tool configured to have a low density of transitions (to get some empty automata) and with at least one successor per state.

random-rep Has similar automata, except that each of the 20 acceptance marks may occur multiple times (possibly in both Fin and Inf terms).

Rabin Contains random 10 000-states Rabin automata with 32 marks used without repetition.

Streett Contains Streett automata generated from random LTL formulas as follows. First we consider only LTL formulas that are neither persistence nor recurrence properties. For some LTL formula ψ and its negation, we use `ltl2dstar` 0.5.4 [24] to generate two Streett automata A_ψ, and $A_{\neg\psi}$. Then, if the product $P = A_\psi \otimes A_{\neg\psi}$ has more than 1000 states, we include P in the dataset (this will be an empty automaton), and we also include the larger of A_ψ or $A_{\neg\psi}$ (both are non-empty due to our selection of ψ).

parity-like Also contains generated automata of the form $P = A_\psi \otimes A_{\neg\psi}$, but now A_ψ and $A_{\neg\psi}$ are deterministic parity min-odd automata generated by Spot's `ltl2tgba -P D` command. In this case, the product has not exactly parity acceptance formulae; it is rather a conjunction of parity acceptance. However the subset of Algorithm 1 needed to handle parity or Streett can also handle conjunction of parity. Again, for each product P, we also add the larger of A_ψ and $A_{\neg\psi}$ to the dataset, to include some non-empty automata.

Table 3 shows the average runtime of the new version of the emptiness check (called `is_empty`) compared to the previous implementation based on Fin-removal (`old_is_empty`). Figure 3 shows a scatter plot of the same experi-

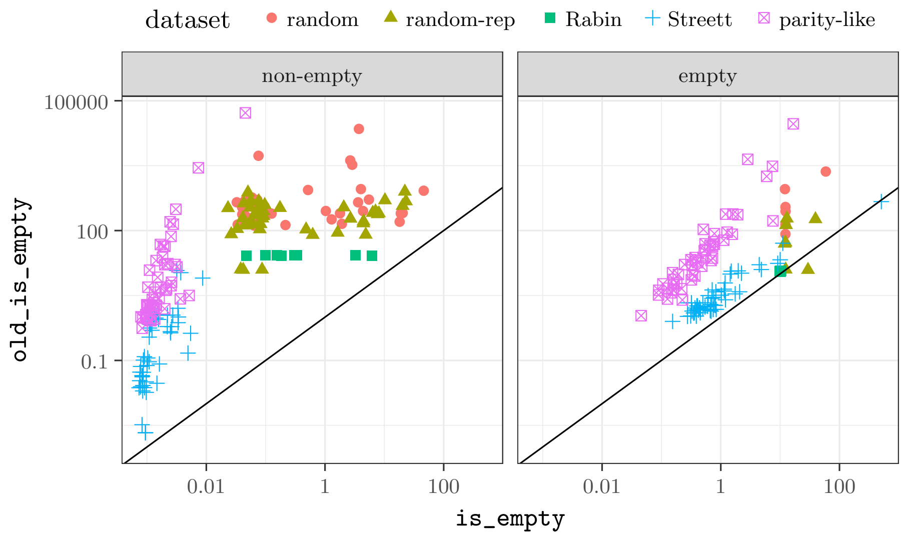


Fig. 3. Runtime comparison (in milliseconds) of the old and current implementations of the generic emptiness checks of Spot, over 5 different datasets.

ment. All measurements were done on an Intel Core i7-6820HQ CPU (2.70GHz) running Linux, using a development version of Spot.[9]

We should start by commenting the results on empty Rabin automata. On those automata, the amount of work needed in `old_is_empty` to create the equivalent Fin-less automaton is similar to the amount of work performed by the new `is_empty`. The minor improvement is due to the fact that `is_empty` creates no new automaton. For empty Streett automata with k clauses, the Fin-removal procedure has to generate a generalized Büchi automaton that is 2^k time larger, and `old_is_empty` pays this price. For empty automata with random and parity acceptance, `old_is_empty` is also slower due to the fact that it has to convert the acceptance formula to DNF upfront, and then process many disjuncts.

There is actually one case from the random-rep dataset where the conversion to DNF saves the day for `old_is_empty`: that point is visible below the diagonal on the right of Figure 3. This case correspond to an automaton whose random acceptance condition is actually equivalent to false. Since our DNF conversion works by encoding the acceptance formula into a BDD, equivalence to false or true are often quickly detected. However outside of such random acceptance condition, unsatisfiable conditions are very rarely seen.

For the non-empty cases, the impressive improvement are easily explained by the fact that `old_is_empty` constructs an equivalent Fin-less automaton before starting to actually check it for emptiness, while `is_empty` will abort as soon as it finds an accepting SCC.

[9] See `https://adl.github.io/genem-exp/bench-app1/` to reproduce.

5 Application 2: Probabilistic Model Checking

Now we turn to the second application of our algorithm, *probabilistic model checking (PMC)* against ω-regular path properties. Here, we are given a model incorporating stochastic behavior, such as a *discrete time Markov chain (DTMC)*, or stochastic and non-deterministic behavior, such as a *Markov decision process (MDP)*. We are then interested in computing the maximal or minimal probabilities of an ω-regular path property, typically given as an LTL formula. For an overview of these topics, we refer to Baier and Katoen [3].

The standard approach for PMC against LTL [38] relies on the construction of a deterministic automaton for the formula, a product construction with the model, a subsequent analysis of so-called *bottom strongly connected components (BSCC)* in DTMCs and *maximal end-components (MEC)* in MDPs, yielding parts of the product model where the acceptance condition of the automaton holds with probability 1 (for DTMCs) or where the non-determinism in the model can be resolved such that it holds with probability 1 (for MDPs). Subsequently, a standard PMC reachability computation yields the desired probabilities. In the case of DTMCs, determining whether a BSCC, i.e., an SCC with no outgoing edges, satisfies an Emerson-Lei acceptance condition with probability 1 is easily decidable in polynomial time(Appendix C).

For MDP, the step of finding end components that almost surely accept an Emerson-Lei acceptance condition is more involved. Here, we are in a situation similar to finding an accepting cycle, with end-components (strongly connected subgraphs that are closed under probabilistic branching) taking the role of SCCs. The computation of maximal end-components can be done in $\mathcal{O}\left(|E| \cdot \min\left(\sqrt{|E|}, |V|\right)\right)$ if represented explicitly [6, 7] and $\mathcal{O}\left(|V| \cdot \sqrt{|E|}\right)$ if represented symbolically [10]. We adapt Algorithm 1 by dealing with the graph of the product-MDP, replacing the SCC decomposition in line 2 with the computation of MECs.

We have implemented this adapted algorithm as an extension to PRISM 4.4 and report here on the benchmark. This extension will be integrated in an official future PRISM version.

As PRISM itself generates deterministic Rabin automata from LTL formulas, we rely on Spot 2.7.4 for producing deterministic Emerson-Lei automata (columns "Emerson-Lei" in Table 4). We use state-based acceptance as PRISM does not support transition-based acceptance. For comparison, we consider the standard implementations in PRISM using the internal generation of deterministic Rabin automata via the algorithms of `ltl2dstar` [24] (columns "Rabin") and using deterministic generalized Rabin automata obtained from Rabinizer 4 [30] (columns "generalized Rabin"). The computation was carried out with the explicit engine of PRISM on a computer with two Intel E5-2680 8-core CPUs at 2.70 GHz with 384GB of RAM running Linux and a time-out of 30 min and 10GB memory limit.[10]

[10] See `https://adl.github.io/genem-exp/bench-app2/` to reproduce.

Table 4. Model checking times for the mutual exclusion protocol with four participants. The model contains 27600 states. $t^{\mathrm{EL}}_{\mathrm{MEC}}$ denotes the time for the MEC analysis using Algorithm 1, whereas $t^{\mathrm{Rabin}}_{\mathrm{MEC}}$ denotes the time for the MEC analysis using the standard (generalized) Rabin algorithm as implemented in PRISM. n stands for the number of acceptance marks occurring in the deterministic automaton. All times are measured in seconds and '−' means time-out.

	Emerson-Lei		generalized Rabin			Rabin		
Property	$t^{\mathrm{EL}}_{\mathrm{MEC}}$	n	$t^{\mathrm{Rabin}}_{\mathrm{MEC}}$	$t^{\mathrm{EL}}_{\mathrm{MEC}}$	n	$t^{\mathrm{Rabin}}_{\mathrm{MEC}}$	$t^{\mathrm{EL}}_{\mathrm{MEC}}$	n
$\mathrm{Pr}^{\min}(\psi_3)$	109.8	4	130.7	121.1	4	−	−	14
$\mathrm{Pr}^{\max}(\psi_4)$	0.4	3	234.3	0.7	6	−	585.9	8
$\mathrm{Pr}^{\max}(\psi_6)$	0.4	3	100.1	0.6	5	−	855.1	6
$\mathrm{Pr}^{\min}(\psi_8)$	0.6	4	251.9	119.0	6	1.6	0.6	6
$\mathrm{Pr}^{\max}(\psi_9)$	−	4	−	−	12	−	−	−
$\mathrm{Pr}^{\min}(\psi_{10})$	107.0	6	355.3	127.3	10	54.9	9.6	6

As the benchmark model and LTL formulas, we focus here on the mutual exclusion protocol [36], which is provided as an MDP by the PRISM benchmark suite [31]. The LTL formulas we check are described by Hahn et al. [23, Table 2] and are listed in Appendix C.1. We removed 2 formulas equivalent to t.

Table 4 shows the running times of the MEC analysis. Algorithm 1 shows a good behavior in general. If we focus on the two first columns Emerson-Lei and Generalized Rabin, the time used in our Emerson-Lei check is always smaller than time in the original generalized Rabin check. In particular line 12 of Algorithm 1 turns out to be beneficial, as it allows determine early if the MEC is accepting. Also, the columns n show that the number of acceptance marks is typically small (in this benchmark always 14 at most). In the last line of Table 4, $t^{\mathrm{EL}}_{\mathrm{MEC}}$ is much smaller for the Rabin automaton than for the Emerson-Lei automaton due to the more complicated acceptance condition of the latter.

6 Conclusion

We presented a simple and efficient algorithm deciding emptiness of automata over infinite words with transition-based Emerson-Lei acceptance condition. The algorithm subsumes several known algorithms for automata classes with simpler acceptance conditions and keeps polynomial time complexity for these automata classes. We have also suggested an application of the algorithm in probabilistic model checking of MDPs. The algorithm has been implemented in Spot and PRISM and experimental evaluation shows improved performance of these tools.

The algorithm can be further improved in several directions. In particular, the running time of the current algorithm is influenced by some nondeterministic choices (lines 2, 15, and 19). We plan to replace these choices by heuristics. Another research goal is a parallel version of the algorithm.

Acknowledgement. This research was partially supported by the DFG through the DFG-project BA-1679/11-1, the DFG-project BA-1679/12-1, the Collaborative Research Centers CRC 912 (HAEC) and CRC 248 (DFG grant 389792660 as part of TRR 248), the Cluster of Excellence EXC 2050/1 (CeTI, project ID 390696704, as part of Germany's Excellence Strategy), the Research Training Groups QuantLA (GRK 1763), by F.R.S.-FNRS through the grant F.4520.18 (ManySynth), and by the Czech Science Foundation through the grant GA19-24397S.

References

1. T. Babiak, F. Blahoudek, M. Křetínský, and J. Strejček. Effective translation of LTL to deterministic Rabin automata: Beyond the (F, G)-fragment. In *ATVA'13*, *LNCS* 8172, pp. 24–39. Springer, 2013.
2. T. Babiak, F. Blahoudek, A. Duret-Lutz, J. Klein, J. Křetínský, D. Müller, D. Parker, and J. Strejček. The Hanoi Omega-Automata Format. In *CAV'15*, *LNCS* 8172, pp. 442–445. Springer, 2015. See also `http://adl.github.io/hoaf/`.
3. C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.
4. V. Bloemen, A. Duret-Lutz, and J. van de Pol. Model checking with generalized Rabin and Fin-less automata. *International Journal on Software Tools for Technology Transfer*, 2019.
5. U. Boker. Why these automata types? In *LPAR'18*, vol. 57 of *EPiC Series in Computing*, pp. 143–163. EasyChair, 2018.
6. K. Chatterjee and M. Henzinger. Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In *SODA'11*, pp. 1318–1336. SIAM, 2011.
7. K. Chatterjee and M. Henzinger. Efficient and dynamic algorithms for alternating Büchi games and maximal end-component decomposition. *Journal of the ACM*, 61(3), 2014.
8. K. Chatterjee, A. Gaiser, and J. Křetínský. Automata with generalized Rabin pairs for probabilistic model checking and LTL synthesis. In *CAV'13*, *LNCS* 8044, pp. 559–575. Springer, 2013.
9. K. Chatterjee, M. Henzinger, and V. Loitzenbauer. Improved algorithms for parity and Streett objectives. *Logical Methods in Computer Science*, 13(3), 2017.
10. K. Chatterjee, M. Henzinger, V. Loitzenbauer, S. Oraee, and V. Toman. Symbolic algorithms for graphs and Markov decision processes with fairness objectives. In *CAV'18*, *LNCS* 10982, pp. 178–197. Springer, 2018.
11. J.-M. Couvreur. On-the-fly verification of temporal logic. In *FM'99*, *LNCS* 1708, pp. 253–271. Springer, 1999.
12. J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In *SPIN'05*, *LNCS* 3639, pp. 143–158. Springer, 2005.
13. C. Dax, J. Eisinger, and F. Klaedtke. Mechanizing the powerset construction for restricted classes of ω-automata. In *ATVA'07*, *LNCS* 4762. Springer, 2007.
14. E. W. Dijkstra. Finding the maximal strong components in a directed graph. In *A Discipline of Programming*, chapter 25, pp. 192–200. Prentice-Hall, 1976.
15. A. Duret-Lutz. Manipulating LTL formulas using Spot 1.0. In *ATVA'13*, *LNCS* 8172, pp. 442–445. Springer, 2013.
16. A. Duret-Lutz. *Contributions to LTL and ω-Automata for Model Checking.* Habilitation thesis, Université Pierre et Marie Curie (Paris 6), Feb. 2017.

17. A. Duret-Lutz, D. Poitrenaud, and J.-M. Couvreur. On-the-fly emptiness check of transition-based Streett automata. In *ATVA'09*, *LNCS* 5799, pp. 213–227. Springer, 2009.
18. A. Duret-Lutz, F. Kordon, D. Poitrenaud, and E. Renault. Heuristics for checking liveness properties with partial order reductions. In *ATVA'16*, *LNCS* 9938, pp. 340–356. Springer, 2016.
19. E. A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.
20. J. Esparza, J. Křetínský, J. Raskin, and S. Sickert. From LTL and limit-deterministic Büchi automata to deterministic parity automata. In *TACAS'17*, *LNCS* 10205, pp. 426–442, 2017.
21. J. Esparza, J. Křetínský, and S. Sickert. One theorem to rule them all: A unified translation of LTL into ω-automata. In *LICS'18*, pp. 384–393. ACM, 2018.
22. J. Geldenhuys and A. Valmari. Tarjan's algorithm makes on-the-fly LTL verification more efficient. In *TACAS'04*, *LNCS* 2988, pp. 205–219. Springer, 2004.
23. E. M. Hahn, G. Li, S. Schewe, A. Turrini, and L. Zhang. Lazy probabilistic model checking without determinisation. In *CONCUR'15*, vol. 42 of *LIPIcs*, pp. 354–367. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
24. J. Klein and C. Baier. Experiments with deterministic ω-automata for formulas of linear temporal logic. *Theoretical Computer Science*, 363(2):182–195, 2006.
25. J. Klein and C. Baier. On-the-fly stuttering in the construction of deterministic ω-automata. In *CIAA'07*, *LNCS* 4783, pp. 51–61. Springer, 2007.
26. Z. Komárková and J. Křetínský. Rabinizer 3: Safraless translation of LTL to small deterministic automata. In *ATVA'14*, *LNCS* 8837, pp. 235–241. Springer, 2014.
27. J. Křetínský and J. Esparza. Deterministic automata for the (F,G)-fragment of LTL. In *CAV'12*, *LNCS* 7358, pp. 7–22. Springer, 2012.
28. S. C. Krishnan, A. Puri, and R. K. Brayton. Deterministic ω-automata vis-a-vis deterministic Büchi automata. In *ISAAC'94*, *LNCS* 834, pp. 378–386. Springer, 1994.
29. J. Křetínský and R. Ledesma-Garza. Rabinizer 2: Small deterministic automata for LTL $\setminus$ GU. In *ATVA'13*, *LNCS* 8172, pp. 446–450. Springer, 2013.
30. J. Křetínský, T. Meggendorfer, S. Sickert, and C. Ziegler. Rabinizer 4: From LTL to your favourite deterministic automaton. In *CAV'18*, *LNCS* 10981, pp. 567–577. Springer, 2018.
31. M. Z. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *QEST'12'*, pp. 203–204. IEEE Computer Society, 2012.
32. Y. Liu, J. Sun, and J. Dong. Scalable multi-core model checking fairness enhanced systems. In *ICFEM'09*, *LNCS* 5885, pp. 426–445. Springer, 2009.
33. T. Michaud and A. Duret-Lutz. Practical stutter-invariance checks for ω-regular languages. In *SPIN'15*, *LNCS* 9232, pp. 84–101. Springer, 2015.
34. S. Minato. Fast generation of irredundant sum-of-products forms from binary decision diagrams. In *SASIMI'92*, pp. 64–73, 1992.
35. D. Müller and S. Sickert. LTL to deterministic Emerson-Lei automata. In *GandALF'17*, vol. 256 of *EPTCS*, pp. 180–194, 2017.
36. A. Pnueli and L. D. Zuck. Verification of multiprocess probabilistic protocols. *Distributed Computing*, 1(1):53–72, 1986.
37. E. Renault, A. Duret-Lutz, F. Kordon, and D. Poitrenaud. Three SCC-based emptiness checks for generalized Büchi automata. In *LPAR'13*, *LNCS* 8312, pp. 668–682. Springer, 2013.
38. M. Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *FOCS'85*, pp. 327–338. IEEE Computer Society, 1985.

The following appendices are not part of the published version.

A Termination and Correctness of Algorithm 1

This section proves that Algorithm 1 always terminates and that it provides correct results.

A.1 Termination

The function IS_SCC_EMPTY(S, φ) is recursive. However, any recursive call (either direct on line 21 or indirect via call to IS_EMPTY on lines 17 and 20) uses a condition with strictly less subformulas of the form $\mathsf{Fin}(m)$ compared to the original argument φ. Hence, the recursion depth is bounded by the number of $\mathsf{Fin}(m)$ subformulas in φ. As both **foreach** loops of the algorithm have always only a finite number of iterations and external functions SCCS_OF, MARKS_OF, and REMOVE can be clearly implemented as terminating, the algorithm terminates.

A.2 Correctness

We first prove correctness of the function IS_SCC_EMPTY(S, φ).

Theorem 2. *For any non-trivial SCC S and condition φ it holds that S contains a cycle satisfying φ if and only if* IS_SCC_EMPTY(S, φ) *returns $\bot$.*

Proof. We prove the theorem by induction on the number of $\mathsf{Fin}(m)$ subformulas in φ. Before that, we make a general comment regarding the lines 7–9. These lines simplify the condition by replacing subformulas that trivially cannot by satisfied by any cycle in S by $\bot$ and all subformulas that are trivially satisfied by each cycle in S by $\top$. The SCC S is empty with respect to the simplified formula if and only is it is empty with respect to the formula before simplification.

Base case. First we consider the case where IS_SCC_EMPTY(S, φ) is called with a Fin-less condition φ. The algorithm simplifies φ. If the simplified condition is t, the algorithm correctly returns $\bot$ as S is a non-trivial SCC and thus it contains some cycle and each cycle satisfies t. If the condition is f, the algorithm correctly returns $\top$ as this condition cannot be satisfied any cycle. If the simplified condition is neither t nor f, it has to be a Fin-less condition build over subformulas of the form $\mathsf{Inf}(m)$ where m ranges over M_{occur} (other $\mathsf{Inf}(m)$ subformulas have been replaced by simplification). Line 12 replaces all these subformulas by t and because φ contains no negation, the condition reduces to t and the algorithm immediately returns $\bot$. This is correct as S contains a cycle that visits all its edges and thus satisfies all $\mathsf{Inf}(m)$ subformulas and therefore also the whole condition.

Inductive step. Now we assume that IS_SCC_EMPTY(S, φ) is called with a condition that contains at least one subformula of the form $\mathsf{Fin}(m)$ and that the statement holds for all conditions with less subformulas of this form. The algorithm simplifies the condition and check whether it is t or f. If this is the case, the algorithm returns the correct results due to the same arguments as in the base case. After the simplification, the

Table 5. Complexity of Algorithm 1 for various acceptance formulas. $|V|$ and $|E|$ are the number of vertices and edges in the graph, n is the number of acceptance marks, $|\varphi|$ is the size of the formula, and f is the number of marks m_i that appears as $\mathsf{Fin}(m_i)$ in φ. We also show the subset of lines necessary to handle the given acceptance (when line 15 is missing, it is assumed that $\varphi_j = \varphi$).

X	acceptance type	complexity	range of lines needed	
	Büchi	$\mathcal{O}(\lvert E\rvert)$	1–8, 11–12	(prop. 1)
	generalized Büchi	$\mathcal{O}(n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert)$	1–8, 11–12	(prop. 1)
FL	Fin-Less	$\mathcal{O}(n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert)$	1–8, 11–12	(prop. 1)
	Rabin	$\mathcal{O}(n \cdot \lvert\varphi\rvert \cdot \lvert E\rvert)$	1–8, 11–17, 22	(prop. 2)
GR	generalized Rabin	$\mathcal{O}(n \cdot \lvert\varphi\rvert \cdot \lvert E\rvert)$	1–8, 11–17, 22	(prop. 2)
S	Streett	$\mathcal{O}(f \cdot (n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert))$	1–8, 11–12, 16–17, 22	(prop. 3)
P	parity	$\mathcal{O}(f \cdot (n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert))$	1–8, 11–12, 16–17, 22	(prop. 4)
HR	hyper-Rabin	$\mathcal{O}(\lvert\varphi\rvert \cdot (n \cdot \lvert E\rvert + \lvert\varphi\rvert \cdot \lvert V\rvert))$	1–8, 11–17, 22	(prop. 5)
EL	Emerson-Lei	$\mathcal{O}\left(2^f \cdot n \cdot \lvert\varphi\rvert \cdot \lvert E\rvert\right)$	1–8, 11–12, 19–22	(prop. 6)

condition contains only subformulas with marks that occur in the SCC S. Obviously, S contains a cycle containing all edges and thus satisfying all subformulas of the form $\mathsf{Inf}(m)$. If satisfaction of these subformulas implies satisfaction of the whole condition, Line 12 correctly returns $\bot$ as the cycle satisfies φ. In the opposite case, each disjunct of the condition (now seen as a disjunction) has to contain a subformula of the form $\mathsf{Fin}(m)$. A cycle satisfies the condition if and only if it satisfies at least one of its disjuncts. Hence, the **foreach** cycle starting at line 15 checks all the disjuncts one by one and reduces the number of $\mathsf{Fin}(m)$ subformulas in the disjunct.

A disjuncts of the form $\varphi_j = \bigwedge_{m \in M'} \mathsf{Fin}(m) \ \land \ \varphi'$ can be satisfied only by a cycle containing no edge marked with any $m \in M'$. Hence, we remove these edges and look for satisfaction of φ'. We return $\bot$ if such a cycle is found. Correctness of the answer follows from the inductive hypothesis and the fact that each cycle of a graph is included in some non-trivial SCC of the graph.

If a disjunct φ_j does not have this form, we simply select a $\mathsf{Fin}(m)$ subformula of the disjunct. Each cycle satisfying the disjunct either satisfies $\mathsf{Fin}(m)$ or not. A cycle satisfying $\mathsf{Fin}(m)$ contains no edges marked with m. Hence, we remove these edges and look for a cycle satisfying $\varphi_j[\mathsf{Fin}(m) \leftarrow \mathsf{t}])$. If such a cycle is found, we return $\bot$ and the answer is correct due to the induction hypothesis and the same fact as in the previous paragraph. Otherwise, we look for a cycle satisfying $\varphi_j[\mathsf{Fin}(m) \leftarrow \mathsf{f}])$. Again, we return $\bot$ if such a cycle is found and correctness of this answer follows directly from the induction hypothesis.

If no cycle satisfying any disjunct is found, we correctly return $\top$. □

Correctness of Algorithm 1 is then a simple corollary of Theorem 2.

Corollary 1. *For any graph G and condition φ it holds that G contains a cycle satisfying φ if and only if* IS_EMPTY(G, φ) *returns $\bot$.*

B Complexity of Algorithm 1

Let $G = (V, E) \in \mathbf{G}$ be a marked graph. In this section, we consider the time complexity of running Algorithm 1 on G with different kinds of acceptance formulas φ over n

different acceptance marks. The results from Table 2 are summarized again in Table 5, this time with some extra columns related to the upcoming profs.

We denote by $|\varphi|$ the size of the acceptance formulas (number of terms and operators), and assume, without loss of generality, that $n \leq |\varphi|$ and that each vertex has at least one successor (implying $|V| \leq |E|$). Finally we denote by f the number of marks m_i that appears as $\mathsf{Fin}(m_i)$ terms in φ: this f obviously decreases with each recursive call. We assume that all marks between 0 and $n-1$ occur in φ, so this implies $f \leq n \leq |\varphi|$.

Let $T_{\mathsf{X}}(V, E, \varphi, n, f)$ (resp. $T'_{\mathsf{X}}(V, E, \varphi, n, f)$) denote the time complexity of IS_EMPTY (resp. IS_SCC_EMPTY) when formula φ belongs to the acceptance type abbreviated by X in the first column of Table 5.

Enumerating the non-trivial SCCs of G on line 2 can be done in $\mathcal{O}(|V| + |E|)$ time (or simply $\mathcal{O}(|E|)$ with our assumptions), using well known graph algorithms [14, e.g.]. The maximum number of non-trivial SCCs is $|V|$, if each SCC is a single state with self-loops. In the following, we use V_S and E_S for the sets of vertices and edges in some SCC S. We always have:

$$T_{\mathsf{X}}(V, E, \varphi, n, f) = \mathcal{O}(|E|) + \sum_S T'_{\mathsf{X}}(V_S, E_S, \varphi, n, f) \tag{2}$$

It is implicit that this sum over S covers the graph (V, E) or less (since trivial SCCs are ignored), and therefore $\sum_S |E_S| \leq |E|$.

We assume that the marks on edges of the graph and the sets M_{occur} and M_{every} on line 7 are represented as bit sets, so that union and intersections of these sets can be done in $\mathcal{O}(n)$. Computing the sets M_{occur} and M_{every} can be done with a DFS to explore all reachable edges during which we update M_{occur} and M_{every}, for a total cost of $\mathcal{O}(n \cdot |E_S|)$. Rewriting the acceptance formula φ on lines 8–9, 12, and 20–21 by applying the trivial identities of Equation (1) can be done in $\mathcal{O}(|\varphi|)$. As a consequence, lines 7–12 can be executed in $\mathcal{O}(n \cdot |E_S| + |\varphi|)$.[11]

If φ is a Fin-less formula, Algorithm 1 will never go past line 12, because the acceptance formula is turned either into f by line 8, or to t by lines 9 or 12.

$$\begin{aligned}
T'_{\mathsf{FL}}(V_S, E_S, \varphi, n, f) &= \mathcal{O}(n \cdot |E_S| + |\varphi|) \\
T_{\mathsf{FL}}(V, E, \varphi, n, f) &= \mathcal{O}(|E|) + \sum_S \mathcal{O}(n \cdot |E_S| + |\varphi|) \text{ from (2)} \\
&= \mathcal{O}(|E|) + \mathcal{O}(n \cdot |E| + |\varphi| \cdot |V|) = \mathcal{O}(n \cdot |E| + |\varphi| \cdot |V|)
\end{aligned}$$

Above we use the fact that $\sum_S |E_S| \leq |E|$ and the number of different SCC S is bounded by $|V|$. This gives us the following result.

Proposition 1. *When φ is a Fin-less acceptance formula, Algorithm 1 runs in $T_{\mathsf{FL}}(V, E, \varphi, n, 0) = \mathcal{O}(n \cdot |E| + |\varphi| \cdot |V|) = \mathcal{O}(|\varphi| \cdot |E|)$ time.*

Proposition 1 shows complexity terms that we will encounter in other acceptance classes as well: $\mathcal{O}(n.|E|)$ is the cost of running MARKS_OF over the edges of multiple SCCs that may cover the entire automaton, and $\mathcal{O}(|\varphi|.|V|)$ comes from the evaluation of the acceptance formula in each SCC (the number of SCCs is bounded by $|V|$).

[11] Lines 9–10 are not necessary for the correctness of the algorithm: line 10 is covered by line 12, and line 10 is just an optimization that does not change the complexity.

Consider now φ to be a generalized Rabin acceptance formula with n marks and $\mathcal{O}(|\varphi|)$ disjuncts.[12] We may reach line 15 and iterate over each disjunct φ_j of the form $\mathsf{Fin}(m_i) \wedge \bigwedge_{j\in J_i} \mathsf{Inf}(m_j)$. Such φ_j necessarily satisfies the condition of line 16 and we execute the recursive call on line 17. The formula φ' from line 16 is a Fin-less acceptance formula of size smaller than φ_j. Finally, the REMOVE operation can be executed in $\mathcal{O}(n \cdot |E_S|)$. We therefore have:

$$\begin{aligned}T'_{\mathsf{GR}}(V_S, E_S, \varphi, n, f) &\leq \mathcal{O}(n{\cdot}|E_S| + |\varphi|) + \sum_{\varphi_j}\big(\mathcal{O}(n{\cdot}|E_S|) + T_{\mathsf{FL}}(V_S, E_S, \varphi_j, n, 0)\big)\\ &\leq \mathcal{O}(n{\cdot}|E_S| + |\varphi|) + \sum_{\varphi_j}\mathcal{O}(n{\cdot}|E_S| + |\varphi_j|{\cdot}|V_S|)\end{aligned}$$

We remove the sum using the fact that the number of disjuncts φ_j is bounded by $|\varphi|$, and so is the sum of their sizes.

$$T'_{\mathsf{GR}}(V_S, E_S, \varphi, n, f) \leq \mathcal{O}(n{\cdot}|\varphi|{\cdot}|E_S| + |\varphi|{\cdot}|V_S|) = \mathcal{O}(n{\cdot}|\varphi|{\cdot}|E_S|)$$

Plugging this in (2) we get the following result.

Proposition 2. *Algorithm 1 runs in* $T_{\mathsf{GR}}(V, E, \varphi, n, f) = \mathcal{O}(n \cdot |\varphi| \cdot |E|)$ *time when* φ *is a generalized Rabin acceptance formula.*

The case of Streett acceptance is more subtle. Let φ be a conjunctions of clauses of the form $\mathsf{Inf}(m_{2i}) \vee \mathsf{Fin}(m_{2i+1})$. For an SCC S, if all clauses are satisfied, either because $\mathsf{Fin}(2i+1)$ is replaced by t on line 8 or because $\mathsf{Inf}(2i)$ is replaced by t on line 12, then the algorithm will not go past line 12.

The worst case occurs when at least one clause is not satisfied. For such a clause, we have necessarily m_{2i} missing and $\mathsf{Inf}(m_{2i})$ to be replaced by f on line 8, leaving only $\mathsf{Fin}(m_{2i+1})$. This means that after the check of line 12 fails, it is guaranteed that one of the conjuncts of φ has been reduced to $\mathsf{Fin}(\ldots)$. The condition on line 16 is therefore successful and we execute line 17. Note that φ is a conjunction, so the loop on line 15 does only one iteration with $\varphi_j = \varphi$. Furthermore, line 12 does not modify φ. Therefore, the clauses that were satisfied by S because of their $\mathsf{Inf}(m_{2i})$ term will be still present in φ' and some of them may become unsatisfied after the removal of some edges. Therefore, even if there is at most one recursive call to IS_EMPTY per call to IS_SCC_EMPTY, the depth of the recursion is bounded by the the f different $\mathsf{Fin}(m)$ sets that can occur in the Streett acceptance. We have:

$$T'_{\mathsf{S}}(V_S, E_S, \varphi, n, f) \leq \mathcal{O}(n \cdot |E_S| + |\varphi|) + T_{\mathsf{S}}(V_S, E_S, \varphi, n, f-1)\big)$$

Using (2) we get:

$$T_{\mathsf{S}}(V, E, \varphi, n, f) \leq \mathcal{O}(n \cdot |E| + |\varphi| \cdot |V|) + \sum_{S} T_{\mathsf{S}}(V_S, E_S, \varphi, n, f-1)\big)$$

and iterating this definition[13] we have:

$$T_{\mathsf{S}}(V, E, \varphi, n, f) \leq f \cdot \mathcal{O}(n \cdot |E| + |\varphi| \cdot |V|)$$

[12] Even if φ uses f Fin marks, our definition of Rabin and generalized Rabin (in Table 1) allows acceptance marks to be reused multiple times in φ, allowing formulas with more than f clauses.

[13] Nested sums that appear in the process still cover the graph at most once.

Proposition 3. *When φ is a Streett acceptance formula, Algorithm 1 runs in* $T_{\mathsf{S}}(V, E, \varphi, n, f) = \mathcal{O}(f \cdot (n \cdot |E| + |\varphi| \cdot |V|)) = \mathcal{O}(f \cdot |\varphi| \cdot |E|)$ *time.*

Notice how the upper-bound for Streett acceptance is f time that of Fin-less acceptance. This is because in the worst case a Streett emptiness check will revisit the entire graph f times.

The algorithm handles parity conditions in a way that is similar to the way it deals with Streett acceptance. Assume $\varphi = \mathsf{Inf}(m_0) \lor (\mathsf{Fin}(m_1) \land (\mathsf{Inf}(m_2) \lor (\mathsf{Fin}(m_3) \land \ldots)))$. If φ is not found satisfied by M_{occur} (as checked on lines 8 and 12), it must be that he lowest mark present in the SCC has an odd index. So if the algorithm goes past line 12, the rewritten φ has the shape of $\mathsf{Fin}(m_{2i+1}) \land (\ldots)$. As a conjunction, it causes only one iteration of loop of line 15, and then one recursive call on line 17, where is it sure that at least one Fin term has been removed.

Proposition 4. *When φ is a parity acceptance formula, Algorithm 1 runs in* $T_{\mathsf{P}}(V, E, \varphi, n, f) = \mathcal{O}(f \cdot (n \cdot |E| + |\varphi| \cdot |V|)) = \mathcal{O}(f \cdot |\varphi| \cdot |E|)$ *time.*

Although it can be confusing, hyper-Rabin formulas [5] are actually disjunctions of Streett formulas. They are easily broken down into Streett formulas by the **foreach** loop, and if we reach the loop, it will be the case that each φ_j originally comes from a Streett condition that was not satisfied by M_{occur}, so φ_j has a Fin term as a top-level conjunct. We therefore execute line 17 with a cost of $T_{\mathsf{S}}(V_S, E_S, \varphi_j, n, f' - 1)$ where f' is the number of Fin marks in φ_j. Clearly $f' \leq |\varphi_j|$, so we can round up the previous complexity to $T_{\mathsf{S}}(V_S, E_S, \varphi_j, n, |\varphi|)$ for simplicity.

$$\begin{aligned}
T'_{\mathsf{HR}}(V_S, E_S, \varphi, n, f) &\leq \mathcal{O}(n \cdot |E_S| + |\varphi|) + \sum_{\varphi_j} \big(\mathcal{O}(n \cdot |E_S|) + T_{\mathsf{S}}(V_S, E_S, \varphi_j, n, |\varphi_j|)\big) \\
&\leq \mathcal{O}(n \cdot |\varphi| \cdot |E_S|) + \sum_{\varphi_j} \mathcal{O}(|\varphi_j| \cdot (n \cdot |E_S| + |\varphi_j| \cdot |V_S|)) \\
&\leq \mathcal{O}(n \cdot |\varphi| \cdot |E_S|) + \mathcal{O}(|\varphi| \cdot (n \cdot |E_S| + |\varphi| \cdot |V_S|) \\
&\leq \mathcal{O}(|\varphi| \cdot (n \cdot |E_S| + |\varphi| \cdot |V_S|))
\end{aligned}$$

Combining the above with (2) we obtain the following result.

Proposition 5. *When φ is an hyper-Rabin acceptance formula, Algorithm 1 runs in* $T_{\mathsf{HR}}(V, E, \varphi, n, f) = \mathcal{O}(|\varphi| \cdot (n \cdot |E| + |\varphi| \cdot |V|))$ *time.*

So far in this section, we have not discussed cases where lines 19–21 are used. However, we have already encountered such a case in Section 3.1 when processing φ_2. Notice that if we remove lines 15–18 (assuming $\varphi_j = \varphi$ in lines 19–21), the algorithm is still correct. The binary tree of recursive calls on lines 20 and 21 will eventually tests all possible valuations of all the $\mathsf{Fin}(m_i)$. At some point, each branch of the recursive tree will reach an acceptance formula that is Fin-less and the recursion will necessarily stop. The maximum depth of this recursion is the number of marks f that are used in Fin terms, so the largest tree we can build has 2^f leaves. If we keep the **foreach** loop as in the proposed algorithms, but still assume that the condition on line 16 always fails, we have the following:

$$\begin{aligned}
T'_{\mathsf{EL}}(V_S, E_S, \varphi, n, f) &\leq \mathcal{O}\left(n \cdot |E_S| + |\varphi|\right) + \sum_{\varphi_j} \big(\mathcal{O}\left(n \cdot |E_S|\right) \\
&\qquad\qquad + T_{\mathsf{EL}}(V_S, E_S, \varphi_j, n, f-1) \\
&\qquad\qquad + T'_{\mathsf{EL}}(V_S, E_S, \varphi_j, n, f-1)\big) \\
&\leq \mathcal{O}\left(\varphi{\cdot}n{\cdot}|E_S|\right) + \sum_{\varphi_j} \left(T'_{\mathsf{EL}}(V_S, E_S, \varphi_j, n, f{-}1) + \sum_{S'} T'_{\mathsf{EL}}(V_{S'}, E_{S'}, \varphi_j, n, f{-}1) \right)
\end{aligned}$$

It can be shown by induction that $T'_{\mathsf{EL}}(V_S, E_S, \varphi, n, f) = \mathcal{O}\left(2^f \cdot n \cdot |\varphi| \cdot |E_S|\right)$ using the above inequality. Combined with (2) we get the following result.

Proposition 6. *Algorithm 1 runs in* $T_{\mathsf{EL}}(V, E, \varphi, n, f) = \mathcal{O}\left(2^f \cdot n \cdot |\varphi| \cdot |E|\right)$ *time when* φ *is an Emerson-Lei acceptance formula.*

It should be noted that if the optional **foreach** loop is removed, the complexity in the EL case is reduced to $\mathcal{O}\left(2^f \cdot (n \cdot |E| + |\varphi| \cdot |V|)\right) = \mathcal{O}\left(2^f \cdot |\varphi| \cdot |E|\right)$, which is 2^f times the complexity of the Fin-less case. With the **foreach** loop, the unsimplified complexity for the EL case is $\mathcal{O}\left(2^f \cdot (n \cdot |\varphi| \cdot |E| + |\varphi| \cdot |V|\right)$.The extra $|\varphi|$ factor can be easily seen if the formula φ contains $\Theta(|\varphi|)$ repetitions of the same φ_j. There is certainly some room for improvement here. On the other hand, without the **foreach** loop, the generalized Rabin acceptance formulas would be treated as the EL case.

B.1 NP-Hardness and Membership

The previous propositions explain the runtime complexity of Algorithm 1. Now we turn to prove that deciding whether a marked graph has an accepting cycle for an acceptance formula φ is NP-complete, by proving an stronger statement, namely that if deciding satisfiability for a class of Boolean formulas is NP-complete, then deciding whether a marked graph contains an accepting cycle for an acceptance condition in the corresponding class is NP-complete as well.

Lemma 1. *Let* $\mathcal{C} - \mathsf{SAT}$ *be a class of Boolean formulas for which the satisfiability problem is NP-complete. Then, deciding whether a marked graph* $G = (V, E)$ *contains a cycle satisfying a given condition* φ *in* $\mathcal{C} - \mathsf{SAT}$ *is NP-complete.*

Proof. For proving the hardness we provide a polynomial reduction from $\mathcal{C} - \mathsf{SAT}$. Let φ' be a $\mathcal{C}$-formula with atomic propositions $X = \{x_1, \ldots, x_n\}$. We generate a marked graph $G = (\{v\}, V)$ with a marked edge $(v, \{i\}, v) \in V$ for every atomic proposition x_i in φ.

For further considerations, we identify an edge by its marks, i.e., an edge subset $Y \subseteq \{n\} \times X \times \{n\}$ can be identified by the corresponding marks $\{k \in \mathbb{N}_0 : (v, \{k\}, v) \in Y\}$. The acceptance condition φ is obtained from φ' by replacing every occurrence of x_i with $\mathsf{Inf}(i)$ and every occurrence of $\neg x_i$ with $\mathsf{Fin}(i)$.

Obviously, an edge subset $Y = \{i_0, \ldots, i_k\}$ (seen as a set of marks) satisfies φ if and only if $Y' = \{x_{i_0}, \ldots, x_{i_k}\} \models \varphi'$. Every model $Y' = \{x_{i_0}, \ldots, x_{i_k}\}$ of φ can be transformed to a corresponding cycle in G by visiting $Y = \{i_0, \ldots, i_k\}$ infinitely often, since there is only one node with a self-loop marked i for every atomic proposition $x_i \in X$. Analogously, every subset of edges $Y = \{i_0, \ldots, i_k\}$ satisfying the acceptance condition φ can be easily translated to a model $Y' = \{x_{i_0}, \ldots, x_{i_k}\}$ for φ'. Therefore, G contains an accepting cycle for φ if and only if φ' is satisfiable.

To prove that finding an accepting cycle belongs to NP, we assume that we are given a marked graph G and an acceptance condition φ. One can choose non-deterministically a cycle, and check whether the set of markings in the cycle satisfies φ. $\square$

Unless $\mathsf{P} \neq \mathsf{NP}$, the converse direction does not necessarily hold: Deciding the satisfiability for $2\mathsf{CNF}$, i.e., conjunctive normal form where every clause contains at most two literals, can be done in polynomial time. For Emerson-Lei acceptance, $2\mathsf{CNF}$ is NP-complete, as formulas of the form $\bigwedge_{i=0}^{n}\big(\mathsf{Fin}(m_{2i}) \vee \mathsf{Fin}(m_{2i+1})\big)$ yield already NP-completeness [19, Proof of Thm. 4.7].

C Probabilistic Model Checking

For a clear and easy presentation we start with Markov chains and progress afterwards to Markov decision processes.

Markov chains are an operational model for systems that exhibit solely probabilistic choices.

Definition 1. *A* Markov chain *is a tuple* $\mathcal{M} = (S, P, \iota, \mathrm{AP}, \ell)$*, where*

- S *is a finite set of states,*
- $P : S \times S \to [0,1]$ *is the transition probability function satisfying:*

$$\sum_{s' \in S} P(s,s') \in \{0,1\} \qquad \textit{for all } s \in S,$$

- $\iota : S \to [0,1]$ *is the initial distribution satisfying* $\sum_{s \in S} \iota(s) = 1$,
- $\ell : S \to 2^{\mathrm{AP}}$ *is a labeling function.*

The last two components, AP and ℓ, serve to formalize properties of paths in $\mathcal{M}$. Paths in $\mathcal{M}$ are finite or infinite sequences $\pi = s_0\, s_1\, s_2 \ldots$ starting in the initial state s_0 that are built by consecutive steps, i.e., $P(s_i, s_{i+1}) > 0$ for all i. The trace of π is the word over the alphabet $\Sigma = 2^{\mathcal{AP}}$ that arises by taking the projections to the state labels, i.e., $\mathrm{trace}(\pi) = \ell(s_0)\,\ell(s_1)\,\ell(s_2)\ldots$. For an ω-regular language L, we denote the probability, that the Markov chain $\mathcal{M}$ generates a trace contained in L by $\Pr^{\mathcal{M}}(L)$. For further detail on the probability measure generated we refer to Baier and Katoen [3]. For Markov chains the analogous problem to (non)-emptiness is the positiveness problem, i.e., deciding $\Pr^{\mathcal{M}}(\varphi) > 0$ for a Markov chain $\mathcal{M}$ and an acceptance condition φ.

In contrast to automata, a Markov chain cannot visit a connected component that is a strict subset of a SCC from some point on exclusively with probability 1. Instead, a Markov chain will enter a reachable bottom SCC (BSCC), i.e., a SCC without outgoing transitions, and visit there every state in the BSCC infinitely often with probability 1. Thus, the existence of a reachable BSCC accepting φ is equivalent to positiveness. Deciding whether a bottom SCC with state (or transition) set $\mathcal{B}$ is accepting for φ can be done by checking whether $\mathcal{B}$, seen as an interpretation, is a model for φ. So, overall for Markov chains, deciding positivity for a generic acceptance condition φ can be done in polynomial time.

Proposition 7. *Deciding* $\Pr^{\mathcal{M}}(\varphi) > 0$ *for a Markov chain* $\mathcal{M}$ *and an Emerson-Lei acceptance* φ *can be done in polynomial time.*

Markov decision processes combine probabilities with nondeterminism:

Definition 2. *A* Markov decision process *is a tuple* $\mathcal{M} = (S, Act, P, \iota, \mathrm{AP}, \ell)$*, where*

- S *is a finite set of states,*
- Act *is a finite set of actions,*
- $P : S \times Act \times S \to [0,1]$ *is the transition probability function satisfying:*

$$\sum_{s' \in S} P(s, \alpha, s') \in \{0,1\} \qquad \textit{for all } s \in S,\ \alpha \in \mathrm{Act},$$

- $\iota : S \to [0,1]$ *is the initial distribution satisfying* $\sum_{s \in S} \iota(s) = 1$*, and*
- $\ell : S \to 2^{\mathrm{AP}}$ *is a labeling function.*

Since an MDP contains nondeterminism, we need to resolve the nondeterminism to obtain probabilities. Therefore schedulers are introduced to choose the current nondeterministic action in dependence of the history. Formally, it is a function $\mathfrak{s} : (S \times Act)^* \times S \to Act$. With a fixed scheduler, an MDP behaves purely probabilistic and so we can reason about probabilities of path events. For an ω-regular language L and a scheduler $\mathfrak{s}$, we denote the probability of L under $\mathfrak{s}$ by $\mathrm{Pr}^{\mathcal{M},\mathfrak{s}}(L)$. Analogously, we write $\mathrm{Pr}^{\mathcal{M},\mathfrak{s}}(L)$ for the probability of an MDP $\mathcal{M}$ to satisfy an acceptance condition φ. In practice, one looks for the maximal probability $\mathrm{Pr}^{\mathcal{M},\max}(L)$ or minimal probability $\mathrm{Pr}^{\mathcal{M},\min}(L)$, yielding a maximal or minimal scheduler.

The analog of an accepting cycle in an automaton is an accepting end-component in an MDP. An end-component is an SCC closed under probabilistic choice. Similar to an SCC in an automaton, that can contain several sub cycles, an end-component can contain several sub end-components. This also results in NP-completeness for checking whether an end-component satisfies an accepting condition.

C.1 Benchmark: Mutual Exclusion Protocol

Section 5 uses the following LTL formulas, published by Hahn et al. [23, Tab. 2].

$$\begin{aligned}
\psi_3 &= \mathsf{GF}a \wedge \mathsf{GF}b \wedge \mathsf{GF}c \wedge \mathsf{GF}d\\
\psi_4 &= (\mathsf{GF}a' \vee \mathsf{FG}\neg b') \wedge (\mathsf{GF}b' \vee \mathsf{FG}\neg c')\\
\psi_6 &= (\mathsf{GF}a' \vee \mathsf{FG}\neg b') \wedge (\mathsf{GF}b' \vee \mathsf{FG}\neg c') \wedge (\mathsf{GF}c' \vee \mathsf{FG}\neg a')\\
\psi_8 &= (\mathsf{GF}\neg a \vee \mathsf{GF}a' \vee \mathsf{FG}a'') \wedge \mathsf{GF}\neg a' \wedge \mathsf{GF}a''\\
\psi_9 &= (\mathsf{G}\neg a \vee \mathsf{G}\neg b \vee \mathsf{G}\neg c) \wedge (\mathsf{FG}\neg a'' \vee \mathsf{GF}b'' \vee \mathsf{GF}c'') \wedge (\mathsf{FG}\neg b'' \vee \mathsf{GF}a'' \vee \mathsf{GF}c'')\\
\psi_{10} &= \mathsf{FG}\neg a' \vee \mathsf{FG}\neg b' \vee \mathsf{GF}c' \vee (\mathsf{FG}\neg a \wedge \mathsf{GF}b \wedge \mathsf{GF}c)
\end{aligned}$$

The atomic propositions have the following meaning:

a: $p_1 = 10$	b: $p_2 = 10$	c: $p_3 = 10$	d: $p_4 = 10$
a': $p_1 = 0$	b': $p_2 = 0$	c': $p_3 = 0$	
a'': $p_1 = 1$	b'': $p_2 = 1$	c'': $p_3 = 1$	