Minimization of counterexamples in SPIN

Paul Gastin, Pierre Moro, and Marc Zeitoun

LIAFA, Univ. of Paris 7, Case 7014,
2 place Jussieu, F-75251 Paris Cedex 05, France
{gastin,moro,mz}@liafa.jussieu.fr

Abstract. We propose an algorithm to find a counterexample to some property in a finite state program. This algorithm is derived from SPIN's one, but it finds a counterexample faster than SPIN does. In particular it still works in linear time. Compared with SPIN's algorithm, it requires only one additional bit per state stored. We further propose another algorithm to compute a counterexample of minimal size. Again, this algorithm does not use more memory than SPIN does to approximate a minimal counterexample. The cost to find a counterexample of minimal size is that one has to revisit more states than SPIN. We provide an implementation and discuss experimental results.

1 Introduction

Model-checking is used to prove the correctness of properties of hardware and software systems. When the model is incorrect, locating errors is important to provide hints on how to correct either the system or the property to be checked. Model checkers usually exhibit counterexamples, that is, faulty execution traces of the system. The simpler the counterexample is, the easier it will be to locate, understand and fix the error. A counterexample can mean that the abstraction of the system (formalized as the model) is too coarse; several techniques can be used to refine the model, guided by the counterexample found by the model-checker. Since the refinement stage is done manually, it is very important to compute *small* counterexamples (ideally of minimal size) in case the property is not satisfied.

It is well-known that verifying whether a finite state system $\mathcal{M}$ satisfies an LTL property φ is equivalent to testing whether a Büchi automaton $\mathcal{A} = \mathcal{A}_{\mathcal{M}} \cap \mathcal{A}_{\neg\varphi}$ has no accepting run [8], where $\mathcal{A}_{\mathcal{M}}$ is a Kripke structure describing the system and $\mathcal{A}_{\neg\varphi}$ is a Büchi automaton describing executions that violate φ. It is easy, in theory, to determine whether a Büchi automaton has at least one accepting run. Since there is only a finite number of accepting states, this problem is equivalent to finding a reachable accepting state and a loop around it. A counterexample to φ in $\mathcal{M}$ can then be given as a path $\rho = \rho_1\rho_2$ in the Büchi automaton, where ρ_1 is a simple (loop-free) path from the initial state to an accepting state, and ρ_2 is a simple loop around this accepting state (see Figure 1). The model-checker SPIN[6,5] can find counterexamples by exploring on the fly the synchronized product of the system and the property. Our goal is

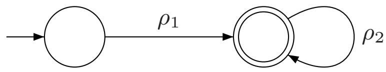


Fig. 1. An accepting path in a Büchi automaton

to find short counterexamples while sparing memory. The first trivial remark is that we can reduce the length of a counterexample if we do not insist on the fact that the loop starts from an accepting state. Hence, we consider counterexamples of the form $\rho = \rho_1\rho_2\rho_3$ where $\rho_1\rho_2$ is a path from the initial state to an accepting state, and $\rho_3\rho_2$ is a simple loop around this accepting state (see Figure 2). A minimal counterexample can then be defined as a path of this form, such that the length of ρ is minimal.

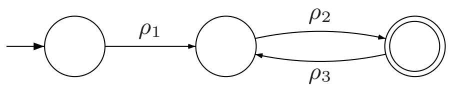


Fig. 2. An accepting path in a Büchi automaton

Finding a counterexample, even of minimal size, can of course be done in polynomial time using minimal paths algorithms based on breadth-first traversals. However, breadth-first traversals are not well-suited to detect loops. Moreover, the model of the system frequently comes from several components working concurrently, and the resulting Büchi automaton can be huge. Therefore, memory is a *critical resource* and, for instance, we cannot afford to store the minimal distance between all pairs of states. Therefore, we retain SPIN's approach and we use a depth-first search-like algorithm [7,3]. Depth-first traversals are well suited to detect loops, but they are not adapted for computing distances between states, which makes the problem more difficult than it first appears.

With this approach, there are actually two difficulties: the first one is to find *one* counterexample, the second one is to find a *small* counterexample, and ideally a *minimal one.*

SPIN has an option to reduce the size of counterexamples it finds. Yet, it does *not* provide the smallest one and results frequently remain too large and difficult to read, even when considering simple systems. For instance, on a natural liveness property on Dekker's mutual exclusion algorithm, SPIN provides a counterexample with 173 transitions. In this case, it is not difficult to see that an error occurs after 23 steps. The reason is that SPIN's algorithm for reducing the size of counterexamples misses lots of them and therefore fails to find the shortest one. Our contribution is the following:

– We propose an algorithm to find a counterexample of a Promela model in linear time. This algorithm is derived from SPIN's, but finds a counterexample faster than SPIN does. Moreover, compared with SPIN's algorithm, it only requires one additional bit per state stored.
– We propose another algorithm to compute a counterexample of minimal size, once a first counterexample has been found. This algorithm does not use more memory than SPIN does with option `-i` when trying to reduce the size of counterexamples. The cost of finding the shortest counterexample is to revisit more states than SPIN does.
– We have implemented a version of the last algorithm whose results are indeed much smaller than those given by SPIN. For instance, for Dekker's algorithm, it actually finds the 23 states counterexample.
– We finally propose other improvements to SPIN's algorithm.

The paper is organized as follows. In Section 2, we describe the algorithm to find a first counterexample and we prove its correctness. However, there is no guarantee that this counterexample is of minimal size. In Section 3, we present an algorithm finding a minimal counterexample. While explaining these algorithms, we exhibit various problems that may arise when computing a counterexample with the current SPIN algorithm. An implementation and experimental results are described in Section 4.

2 Finding the first counterexample

Let $\mathcal{A} = (S, E, s_1, F)$ be a Büchi automaton where S is a finite set of states, $E \subseteq S \times S$ is the transition relation, $s_1 \in S$ is the initial state and $F \subseteq S$ is the set of accepting states. Usually transitions are labeled with actions but since these labels are irrelevant for the emptiness problem, they are ignored in this paper. In pictures, the initial state is marked with an ingoing edge and accepting states are doubly circled. If a state has k outgoing transitions, we number them from 1 to k. Transitions from a state will be considered by the algorithms in the order given by their labels.

A *path* in an automaton is a sequence of states $\gamma = t_1 t_2 \cdots t_k$ (also denoted $t_1, t_2, \ldots, t_k$) such that for all $i < k$ there is a transition from t_i to t_{i+1}. We call k the *length* of γ, and we denote it by $|\gamma|$. The empty path, with no state, is denoted by ε and it has length 0. We say that γ is *simple* if $t_i \neq t_j$ for all $i \neq j$.

A *loop* is a path $t_1 t_2 \cdots t_k$ with $t_k = t_1$. A loop is *accepting* if it contains an accepting state. A loop $t_1 t_2 \cdots t_k$ is a *cycle* if $t_1 t_2 \cdots t_{k-1}$ is a simple path.

An *accepting path*, or *counterexample*, is of the form $\gamma = s_1 \cdots s_k \cdots s_{k+\ell}$ where $s_1 \cdots s_k$ is a path starting from the initial state and $s_k \cdots s_{k+\ell}$ is an accepting loop. Abusing the language, we say that γ is a *simple accepting path* if in addition $s_1 \cdots s_k \cdots s_{k+\ell-1}$ is simple.

In this section, we describe an algorithm finding the first counterexample. It is similar to the nested DFS described in [3,1,7], with an improvement that avoids revisiting some states unnecessarily. This improvement is also useful when minimizing the size of the counterexample.

Algorithm 1 A version of the nested DFS algorithm: the color-DFS

```
void DFS_blue (State s)
1: push(cp, s); s→is_in_cp := true; s→color := blue
2: for all t ∈ E(s) do
3:   if (t→is_in_cp and t ∈ F) then exit with cp · t as counterexample
4:   else if (t→color = white) then DFS_blue(t) end if
5: end for
6: pop(cp); s→is_in_cp := false
7: if (∀t ∈ E(s), t→color = black) then
8:   s→color := black
9: else if (s ∈ F) then
10:   DFS_red(s)
11:   DFS_black(s)
12: end if

void DFS_red (State s)
1: push(cp, s); s→is_in_cp := true; s→color := red
2: for all t ∈ E(s) do
3:   if (t→is_in_cp and (t ∈ F or t→color = blue)) then
4:     exit with cp · t as counterexample
5:   else if (t→color = blue) then
6:     DFS_red(t)
7:   end if
8: end for
9: pop(cp); s→is_in_cp := false
10: s→color := grey
   /*
    * Note that line 10 of DFS_red is not part of the actual algorithm.
    * Its purpose is simply to clarify the correctness proof.
    * Therefore there are actually only four colors as stated in the
    * description above.
    */

void DFS_black (State s)
1: s→color := black
2: for all t ∈ E(s) do
3:   if (t→color ≠ black) then DFS_black(t) end if
4: end for
```

Algorithm 1 uses 4 colors to mark states: *white* < *blue* < *red* < *black*. (We also mark states in grey, but this is just for simplifying the proof.) The color of a state can *only increase*. At the beginning, all states are white and the algorithm `DFS_blue` is called on the initial state s_1.

Two DFSs alternate, the *blue* and *red* ones. The blue DFS is used to locate reachable accepting states and to start red DFSs from these accepting states in postfix order with respect to the covering tree defined by the blue DFS. A red DFS starts (and interrupts the blue one) whenever one pops an accepting state in the blue DFS. A red DFS only visits blue states, that is states already visited by the blue DFS. We will show that if a red DFS initiated from an accepting state r terminates without finding a counterexample then no state reachable from r may be part of an accepting path. Hence, the color of all states reachable from r may be set to black. This is the purpose of the black DFS.

The DFSs used define, at any time, a current path from the initial state to the current state. For convenience, this current path is stored in a global variable `cp`. Actually, this is not necessary with our recursive presentation, since it may be obtained as a by-product of the execution stack when the counterexample is found. (For efficiency, SPIN uses an iterative implementation of the DFS, and stores the current path in a global variable.)

Each state $s \in S$ is represented by a structure and the algorithm requires the following additional fields. The extra cost of these data is only 3 bits for each state, while the nested DFS implemented in SPIN only needs 2 bits per state.

- `Color color` initially `white`.
- `Boolean is_in_cp` initially `false`. This flag is used to test in constant time whether a state belongs to the current path.

When we write `for all` $t \in E(s)$ in the algorithms (see *e.g.* Algorithm 1), we assume that the successors $\{t \in S \mid (s,t) \in E\}$ of s are returned in a fixed order, which is in particular the same in `DFS_blue` and `DFS_red`. This fact is important for the correctness of Algorithm 1. We establish simultaneously the following invariants.

Lemma 1. *(1) Invariant for* `DFS_blue`*: no black state is part of a simple accepting path and all states reachable from a black state are also black.*
(2) Invariant for `DFS_red` *initiated from* `DFS_red`(r) *with* $r \in F$*: either no state reachable from* r *is part of a simple accepting path, or there is a simple accepting path going through* r *and using no black or grey state.*

Proof. (1) During `DFS_blue`(s), if we execute line 8 then all successors of s are black and the result is clear by induction. Now, assume that we execute line 11. Then `DFS_red`(s) was executed completely and the color of s is grey. Using (2) (with $r = s$) we deduce that no state reachable from s is part of a simple accepting path. Hence, after executing `DFS_black`(s), the invariant is still satisfied.
(2) This is the difficult part. First, note that when entering `DFS_red`(r) there are no grey states and we get property (2) directly from (1). Now, this invariant may only be affected by the execution of line 10 inside some `DFS_red`(s). When executing this statement, all successors of s are either black, grey, or red. Note that a red successor of s is necessarily on the current path between r and s since the states on `cp(r)` are still blue, where `cp`(r) is the current path when `DFS_red`(r) was called.

Assume that there exists a simple accepting path α going through r and using no black or grey state. Note that all paths using no black state and going from r to an accepting state must cross `cp`$(r) \cdot r$. This is due to the postfix order of the calls `DFS_red`(t) for $t \in F$. Since we can reach an accepting state, following

α from r, unwinding α once if necessary, we get a path β from r to $cp(r) \cdot r$ using no black or grey state. The path $\texttt{cp}(r) \cdot \beta$ is a simple accepting path using no black or grey state.

If $s \notin \beta$ then the invariant still holds after setting the color of s to grey in line 10. Assume now that $s \in \beta$ and let t be the successor of s on the path β. The color of t must be red. Let v be the last state of β whose color is red and write $\beta = \beta_1 v \beta_2$. Since the color of v is red, it is on the current path between r and s and $\texttt{cp}(r) \cdot r$ is a prefix of $\texttt{cp}(v) \cdot v$. Therefore, $\texttt{cp(v)} \cdot v\beta_2$ is a simple accepting path using no grey or black states and does not contain s. Therefore, the invariant still holds after setting the color of s to grey in line 10. □

Remark 2. One can prove that if a call `DFS_red`(r) with $r \in F$ terminates without finding a counterexample, then all states reachable from r are black or grey. Therefore, in line 10 of `DFS_red`(s), we could set the color of s to black directly and remove line 11 (the call to `DFS_black`) in `DFS_blue`. This modification is fine if we are only interested in finding the first counterexample. But when the color of some state s is set to grey, then we do not know whether s is part of a counterexample or not. In other words, one can deduce that a grey state cannot be part of a counterexample only when the initial call `DFS_red`(r), with $r \in F$, terminates. In order to avoid revisiting unnecessarily some states, the minimization algorithm presented in Section 3 can use the fact that a black state cannot be part of a counterexample. This is why we do not use this modification.

Since the algorithm visits a state at most 3 times, Algorithm 1 terminates. Moreover, one gets as a corollary of Lemma 1 the following statement.

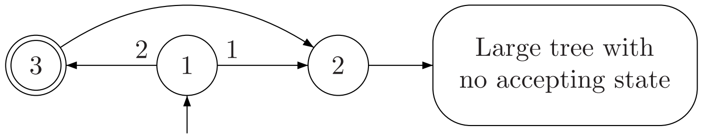
Proposition 3. *If a Büchi automaton $\mathcal{A}$ admits a counterexample, then Algorithm 1 finds a counterexample on input $\mathcal{A}$.*

2.1 Comparison with SPIN's algorithm

The difference between our algorithm and SPIN's is that SPIN does not paint states in black to avoid unnecessary revisits of states. More precisely, in SPIN's algorithm, lines 7 to 12 of `DFS_blue` are replaced with

$$\texttt{if } s \in F \texttt{ then } r := s\texttt{; DFS\_red}(s) \texttt{ endif}$$

where r is a global variable used to memorize the origin of the red DFS. To illustrate the benefit of black states, consider the automaton below. Recall that the transition labels indicate in which order successors are considered by the DFSs. With SPIN's algorithm, the large tree is visited twice. The first visit is started with `DFS_blue`(2) and the second one with `DFS_red`(3). With our algorithm, when `DFS_blue`(2) terminates, state 2 is black. Indeed, `DFS_blue` is called recursively on each state of the tree accessible from state 2. All leaves of this tree, which have no successor, are marked black at lines 7–8, and this propagates back to state 2. Therefore the tree will not be revisited by `DFS_red`(3).



3 Finding a minimal counterexample

To find a minimal counterexample, we use a depth-first search [2] which does not necessarily stop when it reaches a state already visited. Indeed, reaching a state s with a distance to the initial state s_1 smaller than for the previous visit of s may lead to a shorter counterexample.

Therefore, in addition to the fields used in Algorithm 1, each state has an integer field `depth`, storing the smallest length of current paths on which that state occurred. This field remains infinite as long as the state has not been visited, and it can only decrease during the algorithm. We also use an additional variable `mce`, a stack of states containing the minimal counterexample found so far. It is initially empty. At the end of the algorithm, it will contain a minimal counterexample of the whole automaton.

3.1 SPIN's algorithm

The current algorithm implemented in SPIN to find a small counterexample is a variation of the nested DFS algorithm [7]. It carries on the visit below a state either if the state is new or if it is found more quickly than during the previous visits. (And, before popping an accepting state, it looks for a loop from that state.) This algorithm cannot guarantee to find a minimal counterexample. The reason is that, after finding the first counterexample, SPIN backtracks whenever it reaches a state with a path longer than the stored distance to the initial state. This is due to the false intuition that using a longer path will never yield a shorter counterexample. There are two cases where this is not appropriate and the minimal counterexample is missed. The following examples illustrate these two cases. As before, transition labels indicate in which order they are visited.

In the automaton of Fig. 3, the first counterexample found is $s_1 s_2 s_3 s_4 s_5 s_6 s_3$.

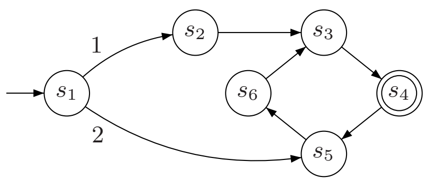


Fig. 3. Missing the minimal counterexample: case 1

After this visit, the state depths are set as follows: $(s_1, 1)$, $(s_2, 2)$, $(s_3, 3)$, $(s_4, 4)$, $(s_5, 5)$, $(s_6, 6)$. SPIN's algorithm then backtracks and s_5 is reached from

s_1 with depth 2. Since this is smaller than the previous depth of s_5 the visit proceeds to s_6 which is reached now at depth 3, and then to s_3, reached at depth 4. But 4 is greater than the previous depth of s_3 and SPIN's algorithm would backtrack missing the shortest counterexample which is $s_1 s_5 s_6 s_3 s_4 s_5$.

The second case is when an accepting state is on the current path. Then, even if no depth was reduced after finding the first counterexample, one should revisit already visited states. An example is shown in Fig. 4.

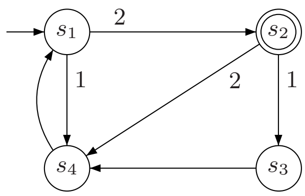


Fig. 4. Missing the minimal counterexample: case 2

The first counterexample found (during the second depth-first search from s_2) is $s_1 s_2 s_3 s_4 s_1$ and the state depths are $(s_1, 1)$, $(s_2, 2)$, $(s_3, 3)$, $(s_4, 2)$. Now, when we reach s_4 from s_2 with the current path $s_1 s_2 s_4$, no depth has been reduced and again SPIN's algorithm would backtrack missing the shortest counterexample which is $s_1 s_2 s_4 s_1$. In this case, the relevant length that was reduced is the length from the accepting state s_2 to s_4 (from 2 to 1). Because memory is the most critical resource, it is not possible to store the length from all accepting states to each state. Therefore, we have to revisit states already visited.

To cope with these cases, Algorithm 2 has two operating modes: a normal one where several criteria can make the algorithm backtrack, and a more careful one, where the visit can only stop when either the current path loops, or becomes longer than the size of the minimal counterexample found so far. In this mode, states may be revisited several times. If the algorithm enters in careful mode while pushing a state s on the current path, it remains in this mode until that occurrence of s is popped off the current path.

In the example of Fig. 3, we would switch to `careful` mode at lines 11–12 of Algorithm 2 when visiting s_5 for the second time, because the field $s_5 \rightarrow$`depth` gets reduced. In the example of Fig. 4, we would switch to `careful` mode at lines 7–8 when visiting s_2, an accepting state.

The important fact is that being careful only in these two situations is sufficient to catch a minimal counterexample.

3.2 An algorithm finding the minimal counterexample

Algorithm 2 is again presented by a recursive procedure which tags states while visiting them. Its first argument is the state to be visited. Its second argument is the mode, initially `normal`, used for the visit. When we detect that some counterexample might be missed in that mode, we switch to the careful mode by calling the procedure with `careful` as the second argument. The mode could be implemented as a global variable, which saves memory. Making it an argument of the procedure yields a simpler presentation of the algorithm.

Algorithm 2 Finding a minimal counterexample

```
void DFS_MIN(State s, Boolean mode)
push(cp, s)
s→depth := min(length(cp), s→depth)
for all t ∈ E(s) do
  if (mce = ε or (length(cp) + 1 < length(mce))) then
    if t ∈ cp then
      if closes_accepting(t) then mce := cp.t end if
    else if (mode = careful or t ∈ F) then
      DFS_MIN(t, careful)
    else if t→depth = ∞ then
      DFS_MIN(t, mode)
    else if ((t→depth > length(cp) + 1) and mce ≠ ε) then
      DFS_MIN(t, careful)
    end if
  end if
end for
pop(cp)
```

In the description of Algorithm 2, we use the following functions:

- `int length(p)` returns the length of the path `p` (*i.e.*, its number of states). Since we only use it with `cp` and `mce` as arguments, one can maintain their lengths in two global variables, hence we may assume that this call requires $O(1)$-time.
- `Boolean closes_accepting(`t`)` returns `true` iff `cp` $\cdot\, t$ is an accepting path (assuming that `cp` itself is not accepting). To implement this function, one can use another stack of states recording, for each state s of the current path `cp` the depth of the last accepting state of `cp` located before s. For instance, if the current path is $[s_1, s_2, s_3, s_4, s_5, s_6]$ and only s_2, s_5 are accepting, then this stack contains $[0, 2, 2, 2, 5, 5]$ (where 0 means that there is no accepting state). The function `closes_accepting` can then be implemented:
  - in $O(1)$-time if we accept to store the depth of each state on the current path. To check that a state closing a cycle creates an accepting cycle, one checks that the depth of its occurrence on the current path is smaller than the depth of the last accepting state on the current path.
  - in $O(n)$-time otherwise, where n is the length of the current path. Nevertheless, the additional stack still gives useful information to avoid visiting the current path. For instance, if s→`depth` (which will be smaller than the depth of s in `cp`) is larger than the depth of the last accepting state on the current path, or if there is no accepting state on it, we know that s does not close an accepting path.

3.3 Correctness of the algorithm

To prove that Algorithm 2 is correct, we introduce the lexicographic ordering on paths starting from the initial state of the automaton. Recall that if a state has k outgoing transitions, they are labeled from 1 to k according to the order in which they will be processed by the algorithm. Let $\lambda : S \times S \to \mathbb{N}$ assigning to each edge its labeling. We extend λ to paths starting at s_1 by letting $\lambda(s_1) = \varepsilon$ and $\lambda(s_1, s_2, \ldots, s_n) = \lambda(s_1, s_2)\lambda(s_2, s_3)\cdots\lambda(s_{n-1}, s_n)$. If γ and γ' are two paths starting at s_1, we say that γ is lexicographically smaller than γ', denoted $\gamma \prec_{\text{lex}} \gamma'$, if $\lambda(\gamma)$ is lexicographically smaller than $\lambda(\gamma')$ (with the usual order over $\mathbb{N}$). We let $\gamma \preceq_{\text{lex}} \gamma'$ iff $\gamma \prec_{\text{lex}} \gamma'$ or $\gamma = \gamma'$.

The first observation is that the algorithm discovers paths in increasing lexicographic order. In other words, each call to `DFS_MIN` makes the current path greater in the lexicographic ordering.

Lemma 4. *Let α and β be the values of* `cp` *after two consecutive executions of line 1 of Algorithm 2. Then, $\alpha \prec_{lex} \beta$.*

Proof. First observe that the test at line 5 guarantees that the current path `cp` remains simple: `DFS_MIN` will not be called on a state that would close the current path. Let $\alpha = s_1 s_2 \cdots s_\ell$. Then either no state is popped before the next execution of line 1, and β is of the form $\alpha s_{\ell+1}$, hence $\alpha \prec_{\text{lex}} \beta$. Or $1 \leq k < \ell$ states are first popped, and the algorithm then pushes t on the current path. By definition of the transition labeling λ, t is a successor of $s_{\ell-k}$ such that $\lambda(s_{\ell-k}, s_{\ell-k+1}) < \lambda(s_{\ell-k}, t)$. Hence, the new value of `cp` is $\beta = s_1 s_2 \cdots s_{\ell-k} t$ and $\alpha \prec_{\text{lex}} \beta$. □

Corollary 5. *Algorithm 2 halts on any input.*

Proof. There is a finite number of simple paths in a finite graph, `cp` takes its values in this finite set and each recursive call makes it greater. □

Since Algorithm 2 discovers an increasing sequence of paths in the lexicographic ordering, it is natural to introduce the following sequence $(\gamma_i)_{0 \leq i \leq p}$. Let $\mathcal{S}$ be the finite set of simple accepting paths. Recall that a simple accepting path is of the form $\alpha s \beta s$ with $\alpha s \beta$ simple and $s\beta \cap F \neq \emptyset$. Since the lexicographic ordering is total, we can define a sequence $(\gamma_i)_{0 \leq i \leq p}$ as follows:

$$\begin{cases} \gamma_0 \quad = \min_{\prec_{\text{lex}}} \mathcal{S} & \text{if } \mathcal{S} \neq \emptyset \\ \gamma_{i+1} = \min_{\prec_{\text{lex}}} \{\gamma \in \mathcal{S} \mid |\gamma| < |\gamma_i|\} & \text{if } \{\gamma \in \mathcal{S} \mid |\gamma| < |\gamma_i|\} \neq \emptyset \end{cases}$$

where $|\gamma|$ denotes the length of γ. By construction, the last element γ_p of this sequence is an accepting path of minimal length. Note that the sequence $\gamma_0, \ldots, \gamma_p$ is increasing in the lexicographic ordering and decreasing in length. For $\alpha, \beta \in \mathcal{S}$, we let $\alpha \sqsubseteq \beta$ if $\alpha \preceq_{\text{lex}} \beta$ and $|\alpha| \leq |\beta|$. We shall use the following simple fact.

Fact 1 *Each γ_i is $\sqsubseteq$-minimal in $\mathcal{S}$.*

The following proposition states in particular that Algorithm 2 is correct, since the last value taken by `mce` is precisely γ_p. It also shows what would be the behavior of a variant of our algorithm which outputs the successive values of `mce`.

Proposition 6. *The successive values taken by the variable* `mce` *during the execution of Algorithm 2 are* $\gamma_{-1} = \varepsilon, \gamma_0, \ldots, \gamma_p$.

Proposition 6 is a direct consequence of Proposition 7 below. Given a path α, we let $\min(\alpha) = \min\{i \mid \alpha \text{ is a prefix of } \gamma_i\}$ and $\max(\alpha) = \max\{i \mid \alpha \text{ is a prefix of } \gamma_i\}$. By convention, $\min(\alpha) = \infty$ and $\max(\alpha) = -\infty$ if α is not a prefix of some γ_i.

Proposition 7. *Let δs be a strict prefix of γ_k with $k = \min(\delta s) < \infty$. Assume that at the beginning of a call* `DFS_MIN(s,mode)`*, we have* $\texttt{cp} = \delta$ *and that for all prefixes $\delta_1 r$ of δs we had* $\texttt{mce} = \gamma_{\min(\delta_1 r)-1}$ *at the beginning of the call* `DFS_MIN(r,_)`*. Then, at the end of the call* `DFS_MIN(s,mode)`*, we have* $\texttt{mce} = \gamma_{\max(\delta s)}$*. Moreover, whenever the variable* `mce` *is updated, it is switched from some $\gamma_{\ell-1}$ to γ_ℓ with $\ell \geq 0$.*

The proof of this proposition in turn uses Lemma 8.

Proof. Let $T = \{t \in E(s) \mid \min(\delta st) < \infty\}$. Write $T = \{t_1, \ldots, t_n\}$ with $\lambda(s, t_i) < \lambda(s, t_{i+1})$ for all $1 \leq i < n$. We use an induction on $|\gamma_k| - |\delta s| \geq 1$.

Claim. If before line 4 when considering $t_i \in E(s)$ we have $\texttt{mce} = \gamma_{\min(\delta st_i)-1}$ then after line 14 of this iteration we have $\texttt{mce} = \gamma_{\max(\delta st_i)}$.

Let $t = t_i$ and $\ell = \min(\delta st)$. Either $\ell = 0$ and $\texttt{mce} = \varepsilon$ or $|\texttt{cp}| + 1 = |\delta st| \leq |\gamma_\ell| < |\texttt{mce}|$ and the test line 4 succeeds.

The first case is when $t \in \texttt{cp} = \delta s$. Then, we have $\gamma_\ell = \delta st$ and t closes an accepting path. Therefore, `mce` is updated to γ_ℓ. For any other successor v of s with $\lambda(s, v) > \lambda(s, t)$, we have $|\delta sv| = |\gamma_\ell| = |\texttt{mce}|$, hence the test line 4 fails. Therefore, the value of `mce` remains γ_ℓ until the end of the call $\texttt{DFS\_MIN(}s\texttt{,\_)}$. Moreover, from $\gamma_\ell = \delta st$ we deduce that $i = n$ and $\gamma_\ell = \max(\delta st_n) = \max(\delta s)$ which proves the claim.

The second case is when $t \notin \texttt{cp} = \delta s$. All hypotheses of Lemma 8 are fulfilled, hence $\texttt{DFS\_MIN(}t\texttt{,\_)}$ is called. When $\texttt{DFS\_MIN(}t\texttt{,\_)}$ is called, δst is a strict prefix of γ_ℓ, $\texttt{cp} = \delta s$, $\texttt{mce} = \gamma_{\ell-1} = \gamma_{\min(\delta st)-1}$ and for all prefixes $\delta_1 r$ of δs we had $\texttt{mce} = \gamma_{\min(\delta_1 r)-1}$ at the beginning of the call $\texttt{DFS\_MIN(}r\texttt{,\_)}$. Therefore, the hypotheses of Proposition 7 are fulfilled and since $|\gamma_\ell| - |\delta st| < |\gamma_k| - |\delta s|$ we get by induction that $\texttt{mce} = \gamma_{\max(\delta st)}$ at the end of the call $\texttt{DFS\_MIN(}t\texttt{,\_)}$. The claim is proved.

Now, we show by induction on i that before line 4 when considering $t_i \in E(s)$ we have $\texttt{mce} = \gamma_{\min(\delta st_i)}$.

Note that $k = \min(\delta s) = \min(\delta st_1)$. By definition of γ_k, no successor t of s with $\lambda(s, t) < \lambda(s, t_1)$ may be such that δst is on a simple accepting path of length less than $|\gamma_{k-1}|$ (with the convention $|\gamma_{-1}| = \infty$). Hence, the value of `mce` remains γ_{k-1} until $t_1 \in E(s)$ is considered. The property holds for $i = 1$.

Assume now that the property holds for some $i < n$. From the claim, we get $\texttt{mce} = \max(\delta st_i)$ after the iteration for $t_i \in E(s)$. Let $q = \max(\delta st_i)$. Note that $\min(\delta st_{i+1}) = \max(\delta st_i) + 1 = q + 1$. By definition of γ_{q+1} and of the set T, no successor v of s with $\lambda(s, t_i) < \lambda(s, v) < \lambda(s, t_{i+1})$ may be such that δsv is on a simple accepting path of length less than $|\gamma_q|$. Hence, the value of $\texttt{mce}$ remains γ_q until $t_{i+1} \in E(s)$ is considered and the property still holds for $i + 1$.

Finally, before line 4 when considering $t_n \in E(s)$ we have $\texttt{mce} = \gamma_{\min(\delta st_n)}$. Using the claim, we get $\texttt{mce} = \max(\delta st_n)$ after the iteration for $t_n \in E(s)$. Note that $q = \max(\delta st_n) = \max(\delta s)$. By definition of the set T, no successor v of s with $\lambda(s, t_n) < \lambda(s, v)$ may be such that δsv is on a simple accepting path of length less than $|\gamma_q|$. Hence, the value of $\texttt{mce}$ remains γ_q until the end of $\texttt{DFS\_MIN(}s\texttt{,mode)}$ and the proposition is proved. □

The proof of the next lemma uses auxiliary results (Lemmas 10 and 11 below) on paths that are totally independent of the algorithm.

Lemma 8. *Let δst be a simple path with $\ell = \min(\delta st) < \infty$. Assume that, while considering the successor t of s in DFS_MIN(s,_), we have $\texttt{cp} = \delta s$ and $\texttt{mce} = \gamma_{\ell-1}$ and that for all prefixes $\delta_1 r$ of δs we had $\texttt{mce} = \gamma_{\min(\delta_1 r)-1}$ at the beginning of the call DFS_MIN(r,_). Then, DFS_MIN(t,_) is called.*

Proof. We let $\alpha' = \delta s$. Assume first that $\ell = 0$, so that $\alpha' t$ is a prefix of γ_0. Since $\texttt{mce} = \gamma_{-1} = \varepsilon$, the test line 4 succeeds. Since $\alpha' t$ is simple and $\texttt{cp} = \alpha'$, the test line 5 fails. Assume that the test line 7 fails. Then, the mode of the algorithm is necessarily $\texttt{normal}$, and in particular there is no accepting state on $\texttt{cp} = \alpha'$. Moreover, t is not accepting: $\alpha' t \cap F = \emptyset$. If the test line 9 also fails, then t has already been visited along a simple path that we denote $\beta' t$. By Lemma 4, we have $\beta' t \prec_{\text{lex}} \alpha' t$. This situation is impossible by Lemma 11. Hence the test line 9 must succeed and $\texttt{DFS\_MIN(}t\texttt{,\_)}$ is called in this case.

Assume now that $\ell \neq 0$. We have $|\texttt{cp}| + \texttt{1} = |\alpha'\texttt{t}| \leq |\gamma_\ell|$ since $\alpha' t$ is a prefix of γ_ℓ. Further, $|\gamma_\ell| < |\gamma_{\ell-1}|$ by definition of the sequence $(|\gamma_i|)_i$. Since $\texttt{mce} = \gamma_{\ell-1}$, we deduce that the test line 4 succeeds. Assume that the test line 7 fails. We show as before that $\alpha' t \cap F = \emptyset$. Assume that the test line 9 also fails. Then, there exists a simple path $\beta' t$ such that $\beta' t \prec_{\text{lex}} \alpha' t$ and $|\beta' t| = t \rightarrow \texttt{depth}$. If $|\beta' t| > |\gamma_\ell|$, then $t \rightarrow \texttt{depth} = |\beta' t| > |\alpha' t| = |\texttt{cp}| + 1$ and $\texttt{DFS\_MIN(}t\texttt{,\_)}$ is called on line 11.

Assume now that $|\beta' t| \leq |\gamma_\ell|$. We want to apply Lemma 10 with $\alpha = \gamma_\ell$. Recall that γ_ℓ is a simple accepting path which is $\sqsubseteq$-minimal in $\mathcal{S}$. Note that $\alpha' t$ is a prefix of α which satisfies property (1) of Lemma 10 and it remains to show that $\alpha' t$ is minimal with this property. Since the algorithm is still in mode $\texttt{normal}$ (the test line 7 failed), for all prefixes $\alpha'_1 r$ of α', we had $r \rightarrow \texttt{depth} = \infty$ when the call to $\texttt{DFS\_MIN(}r\texttt{,normal)}$ was made. By hypothesis, at the beginning of this call, we had $\texttt{mce} = \gamma_{\ell_1}$ where $\ell_1 = \min(\alpha'_1 r)$. Assume that there is a simple path $\beta'_1 r \prec_{\text{lex}} \alpha'_1 r$. Since $\beta'_1 r$ was not visited before $\alpha'_1 r$, we have $|\beta'_1 r| \geq |\gamma_{\ell_1}| \geq |\gamma_\ell| = |\alpha|$ and property (1) of Lemma 10 does not hold for $\alpha'_1 r$. Therefore, $\alpha' t$ is the shortest prefix of $\alpha = \gamma_\ell$ satisfying (1) and Lemma 10 implies that $|\beta'| > |\alpha'|$. We conclude as above that `DFS_MIN(`t`,_)` is called on line 11. $\square$

Lemma 9. *Let $\delta = \alpha s \beta s$ be a path with $s\beta \cap F \neq \emptyset$, αs simple and $\alpha s \cap \beta = \emptyset$. We can construct a simple accepting path $\delta' = \alpha' s' \beta' s'$ such that $|\delta'| \leq |\delta|$ and αs is a prefix of $\alpha' s'$.*

Proof. Assume that δ is not simple. Let t be the first state occuring twice in β. Then, we write $\beta = \beta_1 t \beta_2 t \beta_3$ with $t \notin \beta_1 \beta_2$. If $s\beta_1 t \beta_3 \cap F \neq \emptyset$ then we let $\delta' = \alpha s \beta_1 t \beta_3 s$. The path δ' still satisfies the hypotheses of the lemma and we have $|\delta'| < |\delta|$. Hence we can conclude by induction. Otherwise, we let $\delta' = \alpha s \beta_1 t \beta_2 t$. Again, δ' still satisfies the hypotheses of the lemma and $|\delta'| < |\delta|$. Since αs is a prefix of $\alpha s \beta_1 t$, we can again conclude by induction. $\square$

Lemma 10. *Let $\alpha \in \mathcal{S}$ be a simple accepting path which is $\sqsubseteq$-minimal in $\mathcal{S}$. Assume that there exists a prefix $\alpha' t$ of α satisfying*

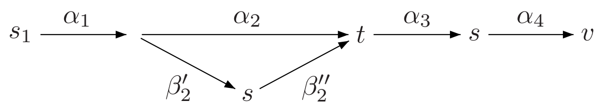
$$\alpha' t \cap F = \emptyset,\ \beta' t \prec_{lex} \alpha' t \text{ and } |\beta' t| \leq |\alpha| \text{ for some simple path } \beta' t. \qquad (1)$$

For the shortest prefix $\alpha' t$ of α satsisfying (1) we have $|\beta'| > |\alpha'|$.

Proof. Let α_1 be the greatest common prefix of α' and β'. We write $\alpha' = \alpha_1 \alpha_2$ and $\beta' = \alpha_1 \beta_2$. Note that the transition between the last state of α_1 and the first state of β_2 is strictly smaller than the transition between the last state of α_1 and the first state of α_2. Hence, for all nonempty prefixes β_2' of β_2 and α_2' of α_2, we have $\alpha_1 \beta_2' \prec_{\text{lex}} \alpha_1 \alpha_2'$.

Assuming by contradiction that $|\beta_2| \leq |\alpha_2|$ we will build a simple accepting path β with $\beta \prec_{\text{lex}} \alpha$ and $|\beta| \leq |\alpha|$, a contradiction with the $\sqsubseteq$-minimality of α.

Write $\alpha = \alpha' t \alpha''$ and let s be the first state on $\beta_2 t$ which occurs also on $t\alpha''$. We write $\beta_2 t = \beta_2' s \beta_2''$ and $\alpha'' = \alpha_3 s \alpha_4$ with $s \notin \alpha_3$. Below, whenever we state that a path is simple, this follows from the definition of s and α_3 and from the fact that α and $\beta' t$ are simple.



1. Assume that $\alpha_3 s$ contains a final state. Then, $\delta = \alpha_1 \beta_2' s \beta_2'' \alpha_3 s$ is an accepting path which is not necessarily simple. Yet, $\beta' t = \alpha_1 \beta_2' s \beta_2''$ is simple and $\alpha_1 \beta_2' s \cap \alpha_3 = \emptyset$ since α is simple and by definition of s and α_3. Applying Lemma 9, we obtain a simple accepting path β with $|\beta| \leq |\delta| \leq |\alpha|$ and $\alpha_1 \beta_2' s$ is a prefix of β. We deduce that $\beta \prec_{\text{lex}} \alpha$ as announced.

Assume now that $\alpha_3 s \cap F = \emptyset$ and let v be the last state of α. By definition of an accepting path, v is also the seed of the accepting loop.

2. We first show that v does not occur in α_2. Assume by contradiction that $\alpha_2 = \alpha_2' v \alpha_2''$ and consider the simple path $\delta = \alpha_1 \beta_2' s \alpha_4 = \delta' v$. We have $\delta' v \prec_{\text{lex}} \alpha_1 \alpha_2' v$ and $|\delta' v| \leq |\alpha|$, a contradiction with the fact that t is the first such state.

3. If v does not occur in $t\alpha_3$ then we let $\beta = \alpha_1 \beta_2' s \alpha_4$.

4. If v occurs in $t\alpha_3$ then we write $t\alpha_3 = \alpha_3' v \alpha_3''$ and we let $\beta = \alpha_1 \beta_2' s \alpha_4 \alpha_3'' s$.

In both cases, we can check that β is a simple accepting path and that $\beta \prec_{\text{lex}} \alpha$ and $|\beta| \leq |\alpha|$ as desired. $\square$

Lemma 11. *If $\alpha' t$ is a prefix of γ_0 with $\alpha' t \cap F = \emptyset$ then there is no simple path $\beta' t$ with $\beta' t \prec_{lex} \alpha' t$.*

Proof. Assume by contradiction that there exists a prefix $\alpha' t$ of γ_0 such that $\alpha' t \cap F = \emptyset$ and $\beta' t \prec_{\text{lex}} \alpha' t$ for some simple path $\beta' t$. In the following, we assume that $\alpha' t$ is the shortest such prefix of γ_0. We will buid a simple accepting path β with $\beta \prec_{\text{lex}} \gamma_0$, a contradiction with the definition of γ_0.

We proceed exactly as in the proof of Lemma 10. The only difference is in case (2) when v occurs in α_2. Here we let $\beta = \alpha_1 \beta_2' s \alpha_4 \alpha_2'' t \alpha_3 s$. Note that we may have $|\beta| > |\gamma_0|$ but we can show that β is a simple accepting path with $\beta \prec_{\text{lex}} \gamma_0$. $\square$

3.4 Remarks on the algorithm

To keep the presentation simple, we have described the algorithm starting from a fresh input. However, one can also start from an automaton already tagged by Algorithm 1. Since no counterexample can go through a black state, this allows us to backtrack in the depth-first search as soon as a black state is seen. This shortens obviously the search by cutting useless parts of the automaton.

Moreover, one can also bound the search by the size of the counterexample produced by Algorithm 1. More precisely, Algorithm 2 is well suited for bounded model-checking. One can give it a bound B for the depth of the research, and it would find successively the counterexamples $\gamma_\ell, \gamma_{\ell+1}, \ldots, \gamma_p$ where ℓ is the first index such that $|\gamma_\ell| < B$. This amounts only to changing the test of line 4 by

$$\big(\texttt{mce} = \varepsilon \ \texttt{and} \ (\texttt{length}(\texttt{cp}) + 1 < B)\big) \ \texttt{or} \ \big(\texttt{length}(\texttt{cp}) + 1 < \texttt{length}(\texttt{mce})\big)$$

Although the time consumption of the algorithm is high, the algorithm outputs all counterexamples of the sequence γ_i when they are discovered, and the user can stop the search at any time. For instance, Dekker's algorithm produces about 80 counterexamples (and the first of these counterexamples is found by the linear

4 Implementation and experimental results

The algorithm presented in Section 2 is quite efficient and visits each state at most twice (in view of Remark 2) in order to find a first counterexample. The second algorithm on the other hand finds the shortest counterexample at the expense of revisiting states much more often. In the worst case, its time complexity is exponential. In order to get the best of the two, we start with the first algorithm until a first counterexample is found (if any) and then switch to the second algorithm to find the shortest counterexample. For this, before switching to the minimization algorithm, we have to pop the execution stack until the first (oldest) accepting state is reached.

In the prototype used to obtain the experimental results presented below, we actually used SPIN's algorithm for finding the first counterexample instead of our algorithm presented in Section 2. Then we switch to our minimization algorithm of Section 3. The reason is that more in-depth changes have to be carried out on SPIN's code to implement our algorithm of Section 2 and our primary goal was just to minimize the size of the counterexample. We are currently implementing the algorithm of Section 2 and since it is always more efficient than SPIN's one, more improvements can be expected.

In the synchronized product between the model and the LTL automaton built by SPIN, there is a strict alternation between transitions of the model and transitions of the LTL automaton (see [6]). Therefore all accepting paths are of odd length and when minimizing the size of a counterexample we can set the bound to `length(mce)-2` instead of `length(mce)-1` for an arbitrary Büchi automaton. This trivial optimization is important for our algorithm since it may revisit states quite often.

We have conducted experiments for various algorithms and specifications. Experiments for which the model does not satisfy the specification and counterexamples exist are gathered in Table 1.

		`SPIN -i`	`Contrex`
Peterson	counterexample length	55	19
	states stored	80	85
	states matched	1968	9469
	computation time	0m0.030s	0m0.070s
Dekker	counterexample length	173	23
	states stored	539	543
	states matched	48593	$2.5 * 10^6$
	computation time	0m0.240s	0m11.420s
Dijkstra (3 users)	counterexample length	5	5
	states stored	211258	209687
	states matched	1.96928e+09	654246
	computation time	71m27.700s	0m1.780s
Hyman	counterexample length	97	17
	states stored	123	157
	states matched	7389	40913
	computation time	0m0.080s	0m0.210s

Table 1. Experiments for various algorithms when a counterexample does exist

We compare our algorithm with `SPIN -i` which tries to reduce the size of the counterexample. Clearly `SPIN -i` does not find the shortest counterexample while we have proved in Section 3 that our algorithm does. The automata of the specifications (never-claims) have been generated by the tool `LTL2BA` [4] both for the verification with `SPIN -i` and with our algorithm. For each experiment, we show, in addition to the size of the minimal counterexample found, the number of *different* states visited by the algorithms (states stored). The last information (states matched) is the number of states (re)visited during the algorithm. Here, each time a state is (re)visited this counter is incremented. The execution time which is indicated is the user time (the system time is negligible in all cases) obtained on a Pentium III 700Mhz with 1Gb of RAM and 1Gb of cache. We see that as expected our algorithm needs to revisit more states that SPIN's one in order to really find the minimal counterexample.

The next table shows, for random graphs of sizes 20. 100 and 1000, the average counterexample length, number of matched states and of counterexamples found.

		`SPIN -i`/ nested-DFS	`Contrex`/ nested-DFS	`Contrex`/ color-DFS
20 states	counterexample length	3,2	3,2	3,2
	states matched	56,2	72,6	69,6
	number of counterexamples	2,8	3,4	3,4
100 states	counterexample length	10,8	10,2	10,2
	states matched	290,8	531	522
	number of counterexamples	7	7,6	7,6
1000 states	counterexample length	9,6	8,6	8,6
	states matched	2385	2727,6	2727
	number of counterexamples	19,4	20,4	20,4

Table 2. Experiments for random graphs

5 Conclusion and open problems

The main contribution of this paper is the algorithm presented in Section 3 which finds a shortest accepting path in a Büchi automaton. It has been implemented and the comparison with SPIN's algorithm clearly demonstrates its superiority. We also proposed an algorithm to find a counterexample, without trying to minimize its length, which is more efficient than SPIN's. It avoids unnecessary revisits of states and hence finds a counterexample more quickly. It also finds a shorter counterexample because it does not insist on having a cycle starting from an accepting state. We plan to implement this algorithm and to compare it experimentally with SPIN. Further, this algorithm detects states that cannot be part of an accepting path (black states). Hence, using it instead of SPIN's before searching for a minimal counterexample should improve the performance of our second algorithm.

Finding a shortest counterexample is time consuming because we need to revisit states many times. A general goal for improving the efficiency is to detect more states that need not be revisited.

Another important issue is to be able to deal with partial order reductions. While the first nested-DFS algorithm [3] failed in the presence of partial order reductions, the version of [7] is able to cope with some reductions. We need to investigate whether our algorithms can handle partial order reductions with reasonable memory requirements.

Finally, it would be interesting to find ways to minimize the length of the counterexample with respect to the model and the LTL specification. Instead, existing algorithms search for counterexamples for the model and *a specific automaton* associated with the LTL specification. It is often the case that this specific automaton is not optimal for finding a short counterexample for the LTL formula.

Acknowledgment. The authors wish to thank the anonymous referees for their careful reading of the submitted version of the paper and for their comments.

References

1. E. M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 1999.
2. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
3. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Computer-aided verification '90 (New Brunswick, NJ, 1990)*, volume 3 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 207–218. Amer. Math. Soc., Providence, RI, 1991.
4. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, number 2102 in Lect. Notes Comp. Sci., pages 53–65. Springer, 2001.
5. G. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
6. G. Holzmann. *The SPIN model-checker*. Addison-Wesley, 2003.
7. G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32, Rutgers, Piscataway, NJ, 1996. American Mathematical Society.
8. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.