Minimization of counterexample in SPIN

Paul Gastin, Pierre Moro, Marc Zeitoun

Minimization of counter-examples 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 counter-example to some property in a finite state program. This algorithm is derived from SPIN's one, but it finds a counter-example faster than SPIN does. (In particular it still works in linear time.) Moreover, it does not require more memory than SPIN. We further propose another algorithm to compute a counter-example of minimal size. Again, this algorithm does not use more memory than SPIN does to approximate a minimal counter-example. The cost to find a counter-example 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 correctness of properties of hardware and software systems. When the program 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 counter-examples, that is, faulty execution traces of the system. The simpler the counter-example is, the easier it will be to locate, understand and fix the error. A counter-example can mean that the abstraction of the system (formalized as the model) is to coarse; several techniques allow to refine it, guided by the counter-example found by the model-checker. Since the refinement stage is done manually, it is very important to compute *small* counter-examples (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, 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 indeed equivalent to finding a reachable accepting state and a loop around it. A counter-example 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 counter-examples given by SPIN are always of this form. Our goal is to find short counter-examples. The first trivial remark is that we can reduce the length of a counter-example if we do not insist on the fact that

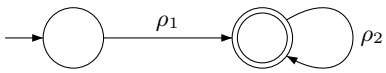


Fig. 1. An accepting path in a Büchi automaton

the loop starts from an accepting state. Hence, we consider counter-examples 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 $\rho_2\rho_3$ is a simple loop around this accepting state (see Figure 2). A minimal counter-example can then be defined as a path of this form, such that the length of ρ is minimal.

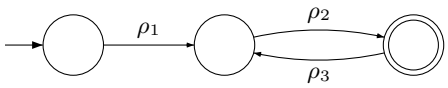


Fig. 2. An accepting path in a Büchi automaton

Finding a counter-example, 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 [6, 2].

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

SPIN has an option to reduce the size of counter-examples it finds. Yet, it does *not* provide the smallest one and results remain frequently 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 counter-example with 173 transitions. In this case, it is not difficult to see that an error occurs after 21 steps. One reason is that SPIN looks for counter-examples as a path to an accepting state followed be a loop around this state. The second reason is that SPIN's algorithm for reducing the size of counter-examples misses lots of them and therefore fails to find the shortest one. Our contribution is the following:

– We propose an algorithm to find a counter-example of a SPIN program in linear time. This algorithm is derived from SPIN's one, but finds a counter-example faster than SPIN does. Moreover, our algorithm does not use more memory than SPIN.

– We propose another algorithm to compute a counter-example of minimal size, once a first counter-example has been found. This algorithm does not use more memory than SPIN does with option `-i` when trying to reduce the size of counter-examples. The cost we have to pay for finding the shortest counter-example 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 21 states counter-example.
– We finally propose other improvements to SPIN's algorithm.

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

2 Finding the first counter-example

Let $\mathcal{A} = (S, E, s_0, F)$ be a Büchi automaton where S is a finite set of states, $E \subseteq S \times S$ is the transition relation, $s_0 \in S$ 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.

Recall that a path in an automaton is a sequence of states $s_1 s_2 \cdots s_k$ (also denoted $s_1, s_2, \ldots, s_k$) such that for all $i = 1, \ldots, k-1$ there is a transition from s_i to s_{i+1}. The empty path, with no transition, is denoted by ε. A loop is a path $s_1 s_2 \cdots s_k$ with $s_k = s_1$. A path $s_1 s_2 \cdots s_k$ is *simple* if $s_i \neq s_j$ for all $i \neq j$. A loop $s_1 s_2 \cdots s_k$ is a *cycle* if $s_1 s_2 \cdots s_{k-1}$ is a simple path. A loop (resp. a cycle) is *accepting* if it contains an accepting state. Finally, an *accepting path* is of the form $\gamma = s_0 \cdots s_k \cdots s_{k+\ell}$ where $s_0 \cdots s_{k+\ell-1}$ is a simple path and $s_k \cdots s_{k+\ell}$ is an accepting cycle. We call $s_0 \cdots s_k$ the *head* of γ. Note that an accepting path starts in the initial state. We also call *counter-example* an accepting path.

In this section, we describe an algorithm finding the first counter-example. It is similar to the nested DFS described in [6], with two improvements: the first one concerns the detection of accepting cycles and the second one avoids revisiting unnecessarily some states. Both improvements are also useful when minimizing the size of the counter-example.

The algorithm 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_0.

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 (blue) 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 counter-example 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`. Note that SPIN also stores the current path in a global variable. Actually, this is not necessary since it may be obtained from the execution stack when the counter-example is found.

Each state $s \in S$ is represented by a structure and the algorithm only requires the following additional fields. The extra cost of these data is only 3 bits for each state (this is the same cost as for SPIN's algorithm).

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

When we write `for all` $s' \in E(s)$ in the algorithms (see *e.g.* Algorithm 1), we assume that the successors $\{s' \in S \mid (s, s') \in E\}$ 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 an 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 an accepting path, or there is an 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 an 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, inside `DFS_red`(s), this invariant is only affected at line 10 when the color of s is set to grey. When executing this statement, all successors of s are either black, grey, or red. Note that a red state is necessarily on the current path. Assume that there exists an accepting path ρ going through r and using no black or grey state. If $s \notin \rho$ then the invariant still holds after setting the color of s to grey in line 10. Assume now that $s \in \rho$ and let s' be the successor of s on the path ρ.

Algorithm 1 A version of the nested DFS algorithm

```
void DFS_blue (State s)
 1: B[i] = (i.color == blue);
 2: if (s'→is_in_cp and s' ∈ F) then exit with cp · s' as counter-example;
 3: else if s'→color = white then DFS_blue(s'); end if
 4: pop(cp); s→is_in_cp := false;
 5: if ∀s' ∈ E(s), s'→color = black then
 6:    s→color := black;
 7: else if s ∈ F then
 8:    DFS_red(s);
 9:    DFS_black(s);
10: end if

void DFS_red (State s)
 1: push(cp, s); s→is_in_cp := true; s→color := red;
 2: for all s' ∈ E(s) do
 3:    if (s'→is_in_cp and (s' ∈ F or s'→color = blue)) then
 4:       exit with cp · s' as counter-example;
 5:    else if s'→color = blue then
 6:       DFS_red(s');
 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 s' ∈ E(s) do
 3:    if s'→color ≠ black then DFS_black(s'); end if
 4: end for
```

Note that all paths using no black state and going from r to an accepting state must cross $cp(r)$ where $cp(r)$ is the current path when `DFS_red`(r) was called. This is due to the postfix order of the calls `DFS_red`(t) for $t \in F$.

Since s' is reachable from r and since following ρ (unwinding once its cycle if necessary) we can reach an accepting state from s', we find in ρ a subpath ρ' containing no accepting state and going from s' to some state s'' in $cp(r)$. Note that $s \notin \rho'$ since ρ is simple. Then, $\rho'' = cp(s') \cdot \rho'$ is an accepting path going through r, containing no black or grey state and such that $s \notin \rho''$. Hence the invariant still holds after setting the color of s to grey in line 10. □

Remark 1. One can prove that if a call `DFS_red`(r) with $r \in F$ terminates without finding a counter-example, 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 counter-example. But when the color of some state s is set to grey, then we do not know whether s is part of a counter-example or not. In other words, one can deduce that a grey state cannot be part of a counter-example 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 counter-example. 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.

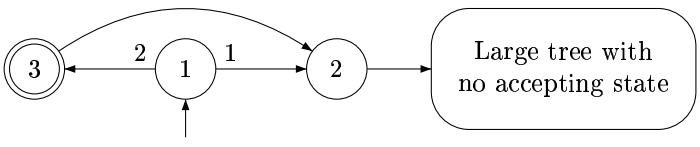
Proposition 1. *If a Büchi automaton $\mathcal{A}$ admits a counter-example, then Algorithm 1 finds a counter-example on input $\mathcal{A}$.*

2.1 Comparison with SPIN's algorithm

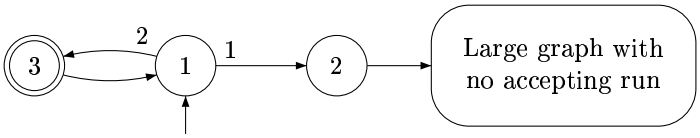
There are essentially two differences between our algorithm and SPIN's one. The first one is that SPIN does not paint states in black to avoid unnecessary revisits of states. More precisely, in SPIN's algorithm, lines 7 to 12 of `DFS_blue` are replaced with

`if` $s \in F$ `then` $r := s$; `DFS_red`(s); `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. Therefore the tree will not be revisited by `DFS_red`(3).



The second difference is that SPIN only looks for accepting cycles around an accepting state. This means that in SPIN's algorithm, the test in line 3 of `DFS_blue` is replaced by `false` and the test in line 3 of `DFS_red` is replaced by $s' = r$ where r is the global variable storing the origin of the red DFS (see above). Again, the benefit of our method is illustrated with the automaton below. Both algorithms behave similarly until `DFS_red`(3) is called. When considering the successor $s' = 1$ of $s = 3$, our algorithm immediately find a counter-example. On the contrary, SPIN will call `DSF_red`(1) which revisits state 2 and the large graph before going to the successor 3 of 1 and finding the counter-example.



3 Finding a minimal counter-example

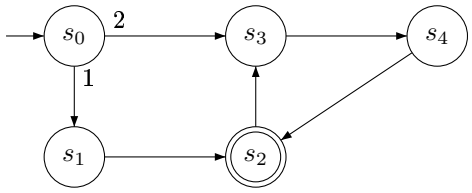
To find a minimal counter-example, we use a depth first search which does not necessarily stops when it reaches a state already visited. Indeed, reaching a state s with a distance to the initial state s_0 smaller than for the previous visit of s may lead to a shorter counter-example.

Therefore, in addition to the fields used in Algorithm 1, each state has an integer field `depth`, storing the minimal distance (from s_0) at which s has been seen on a current path. It is therefore an upper bound of the distance from the initial state s_0. It 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 counter-example found so far. It is initially empty. At the end of the algorithm, it will contain a minimal counter-example of the whole automaton.

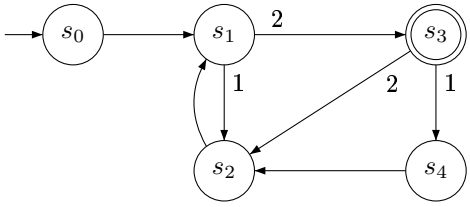
Algorithm 2 has two operating modes: a normal one where several criteria can make the algorithm backtrack, and a more careful one, where the visit can only stop when either the current path loops, or becomes longer than the size of the minimal counter-example found so far. In this mode, states may be revisited several times (in contrast with Algorithm 1 where a state is visited at most 3 times, its color switching to blue, then to red, and finally to black).

The current algorithm implemented in SPIN to find a small counter-example does not guarantee to find a minimal one. This algorithm is a variation of the Nested-DFS algorithm [2]. It visits a state either if it is new or if it is found more quickly than during the previous visits. To check this last condition, it maintains for each state the distance from the initial state s_0. Since the counter-examples are searched as a loop around an accepting state, one cannot hope to get a minimal example. But there is another problem with SPIN's algorithm: after finding the first counter-example, 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 counter-example. There are two cases where this is not appropriate and the minimal

counter-example is missed. The following examples illustrate these two cases. As before, the numbers on transitions indicate in which order they are visited.



With the automaton above, the first counter-example found is $s_0 \to s_1 \to s_2 \to s_3 \to s_4 \to s_2$. During this visit, the state depths are set as follows: $(s_0, 0)$, $(s_1, 1)$, $(s_2, 2)$, $(s_3, 3)$, $(s_4, 4)$. Then s_3 is reached from s_0 with depth 1. Since this is smaller than the previous depth of s_3 the visit continues to s_4 which is reached now at depth 2. Again, 2 is smaller than the previous depth of s_4 and the visit continues to s_2 with depth 3. But 3 is larger than the previous depth of s_2 and SPIN's algorithm would backtrack missing the shortest counter-example which is $s_0 \to s_3 \to s_4 \to s_2 \to s_3$. This example explains why we switch to `careful` mode in lines 11-12 of Algorithm 2.



The second case is when an accepting state is on the current path. Then, even if no depth was reduced after finding the first counter-example, one should revisit already visited states. This situation is illustrated in the automaton above. The first counter-example found is $s_0 \to s_1 \to s_3 \to s_4 \to s_2 \to s_1$ and the state depths are $(s_0, 0)$, $(s_1, 1)$, $(s_2, 2)$, $(s_3, 2)$, $(s_4, 3)$. Now, when we reach s_2 from s_3 with the current path $s_0 \to s_1 \to s_2$, no depth has been reduced and again SPIN's algorithm would backtrack missing the shortest counter-example which is $s_0 \to s_1 \to s_3 \to s_2 \to s_1$. In this case, the relevant length that was reduced is the length from the accepting state s_3 to s_2 (from 2 to 1). Because memory is the most critical resource, it is not possible to store the length from a state to all accepting states. Therefore, we have to revisit states already visited. As above, this example explains why we switch to `careful` mode in line 7-8 of Algorithm 2.

Recall that in the classical depth-first algorithm, the spanning tree is the tree rooted at s_0 whose branches are the maximal values, for the prefix ordering, taken by the current path during the execution of the algorithm. Since the visit will sometimes proceed to already visited states, our algorithm will generate an acyclic spanning graph of the automaton, not a tree: a state can occur on two prefix-incomparable current paths from s_0.

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 counter-example 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 well as a global variable, which saves memory. Making it an argument of the procedure yields a simpler presentation of the algorithm.

Also, to keep the presentation simple, we describe the algorithm starting from a fresh input. However, one can also start from an automaton already tagged by Algorithm 1. Since no counter-example can go through a black state, this allows to backtrack in the depth-first search as soon as a black state is seen. Moreover, one can also bound the search by the upper bound for the size of the minimal counter-example produced by Algorithm 1.

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

- `int length(p)` returns the length of the path `p`. 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(`s`)` returns `true` iff `cp` $\cdot\, s$ 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 last accepting state of `cp` located before s. For instance, if the current path is $[s_0, s_1, s_2, s_3, s_4, s_5]$ and only s_1, s_4 are accepting, then this stack contains $[\bot, s_1, s_1, s_1, s_4, s_4]$ (where $\bot$ means that there is no accepting state). The function `is_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 first occurrence on the current path is smaller than the depth of the last accepting state on the current path (which we can recover from the stack of accepting states).
  - in $O(n)$-time otherwise, where n is the length of the current path. Nevertheless, the stack of accepting states still gives useful information to avoid visiting the current path. For instance, if $s\rightarrow$`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.

To prove the correctness of Algorithm 2, 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 can extend λ to paths starting at s_0 by letting $\lambda(s_0) = \varepsilon$ and $\lambda(s_0, s_1, \ldots, s_k) = \lambda(s_0, s_1)\lambda(s_1, s_2)\cdots\lambda(s_{k-1}, s_k)$. If γ and γ' are two paths starting at s_0, we sat that γ is lexicographically smaller than γ', noted $\gamma \prec_{\text{lex}} \gamma'$, 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 $\gamma = \gamma'$.

Since the algorithm revisits states, we first prove that it cannot loop forever.

Algorithm 2 Finding a minimal counter-example

```
void DFS_MIN (State s, Boolean mode)
 1: s→depth := min(length(cp), s→depth);
 2: push(cp, s);
 3: for all s' ∈ E(s) do
 4:   if (mce = ε or (length(cp) + 1 < length(mce))) then
 5:     if s' ∈ cp then
 6:       if closes_accepting(s') then mce := cp.s'; end if
 7:     else if (mode = careful or s' ∈ F) then
 8:       DFS_MIN(s', careful);
 9:     else if s'→depth = ∞ then
10:       DFS_MIN(s', mode);
11:     else if ((s'→depth > length(cp) + 1) and mce ≠ ε) then
12:       DFS_MIN(s', careful);
13:     end if
14:   end if
15: end for
16: pop(cp);
```

Lemma 2. *Algorithm 2 stops on any input.*

Proof. First observe that the test at line 5 guarantees that the current path `cp` remains simple. That is, `DFS_MIN` will not be called on a state that would close the current path. Second, each call to `DFS_MIN` makes the current path greater in the lexicographic ordering: let $\mathtt{cp}_1 = s_0 s_1 \cdots s_\ell$ be the value of the current path after statement 2 in the call `DFS_MIN`(s_ℓ) and consider the next call to `DFS_MIN`. After popping $k \geq 0$ states, the algorithm pushes $s'_{\ell-k+1}$ on the current path. By definition of the transition labeling λ, $s'_{\ell-k+1}$ is the successor of $s_{\ell-k}$ such that $\lambda(s'_{\ell-k+1}) = \lambda(s_{\ell-k+1}) + 1$. Hence, the new value of `cp` is $\mathtt{cp}_2 = s_0 s_1 \cdots s_{\ell-k} s'_{\ell-k+1}$ and so $\lambda(\mathtt{cp}_2) > \lambda(\mathtt{cp}_1)$. We conclude that the algorithm stops on any input: there is a finite number of simple paths in a finite automaton, `cp` takes its values in this finite set and each recursive call makes it greater. □

Let s be a state and let $\gamma(s)$ be the simple path between s_0 and s which is minimal in the lexicographic ordering. Observe that $\gamma(s)$ is a prefix of a branch of the spanning tree in the classical depth-first search algorithm.

Lemma 3. *Let s be a state visited by the algorithm while* `mce` *is empty. Let $\delta(s)$ be the value of the current path* `cp` *after line 2 of the first call to* `DFS_MIN`(s). *Then $\delta(s) = \gamma(s)$.*

Proof. By definition of $\gamma(s)$, we have $\gamma(s) \preceq_{\text{lex}} \delta(s)$. Assume that $\gamma(s) \neq \delta(s)$ and let $\gamma(s) = \gamma \cdot t \cdot \gamma'$ where γ is the longest prefix of $\gamma(s)$ stored into `cp` during the algorithm. Observe that $\gamma(t) = \gamma \cdot t$. After pushing s onto `cp`, t is examined as a successor of the last state of γ. (Since `mce` is empty, the test line 4 succeeds.) Since $\gamma(s)$ is simple, the test at line 5 for $s' = t$ fails. By definition of t, all subsequent tests (lines 7, 9, 11) have to fail. In particular, t→`depth` is

not infinite, so t was already visited by the algorithm, that is, while the current path was lexicographically smaller than $\gamma \cdot t = \gamma(t)$, a contradiction with the definition of $\gamma(t)$. □

Let $\mathcal{S}$ be the finite set of accepting paths. Since the lexicographic ordering is total, we can define a sequence $(\gamma_i)_{0 \leq i \leq k}$ as follows:

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

where $|\gamma|$ denotes the length of γ. By construction, the last element γ_k of this sequence is an accepting path of minimal length. Note that the sequence $\gamma_0, \ldots, \gamma_k$ is increasing in the lexicographic ordering and decreasing in length.

Lemma 4. *Let $\gamma = \gamma' \cdot s$ be the value of the current path after line 2 in a call to* `DFS_MIN`(s) *and assume that the algorithm is in careful mode. Let μ be the value of* `mce` *at this point. Assume that there exists $\delta = \min_{\prec_{lex}}\{\rho \mid \rho = \gamma\rho', \rho$ is accepting and $|\rho| < |\mu|\}$. Then δ is the next value assigned to* `mce`.

Proof. Since the algorithm is in careful mode after the call `DFS_MIN`(s), it remains in that mode at least as long as s is not popped from `cp`. In that mode, the visit proceeds until either the current path becomes longer than `mce` (line 4), or a cycle is found (line 5). Indeed, if these conditions are meet, the test at line 7 always succeeds. We can now conclude since the sequence of values taken by `cp` after each `push` (line 2) is increasing. □

Proposition 2. *The successive values taken by the variable* `mce` *during the execution of Algorithm 2 are $\varepsilon, \gamma_0, \ldots, \gamma_k$.*

Proof. Clearly, all values of `mce` are accepting paths. Hence, if there is no counter-example, the algorithm ends up with an empty value for `mce`, which is correct.

Notice also that the search remains bounded by the length of the shortest accepting paths found so far (line 4). Since, as noted above, the current path can only increase (in the lexicographic ordering) at each recursive call, the sequence of paths assigned to `mce` is necessarily a subsequence of $\gamma_0, \ldots, \gamma_k$. We shall show that no element of this sequence can be missed.

Assume that there exists at least one accepting path. We show that the first accepting path in the lexicographic ordering, that is, γ_0, is the first value assigned to `mce` during the execution of Algorithm 2.

Let $\gamma_0 = \delta_0\delta_1 r\delta_2$, where δ_0 is the head of γ_0, where $\delta_1 r \delta_2$ is its cycle, and where r is its last accepting state. The situation is depicted in Figure 3.

We claim that $\gamma(r) = \delta_0\delta_1 r$. Indeed, we have $\gamma(r) \preceq_{\text{lex}} \delta_0\delta_1 r$ by definition of $\gamma(r)$. If $\gamma(r) \neq \delta_0\delta_1 r$, then either $\gamma(r)$ is a strict prefix of $\delta_0\delta_1 r$, which is impossible since r does not appear in $\delta_0\delta_1$; or, by definition of the lexicographic ordering, there exists a transition (x, y) in $\gamma(r)$ and (u, v) in $\delta_0\delta_1 r$ such that $\gamma(r) = \gamma \cdot (x, y) \cdot \gamma'$, $\delta_0\delta_1 r = \gamma \cdot (z, t) \cdot \delta'$ with $\lambda(u, v) < \lambda(z, t)$. We deduce that $\gamma(r)\delta_2\delta_1 r \prec_{\text{lex}} \delta_0\delta_1 r\delta_2 = \gamma_0$. Let t be the first repeated state on $\gamma(r)\delta_2\delta_1 r$. Since

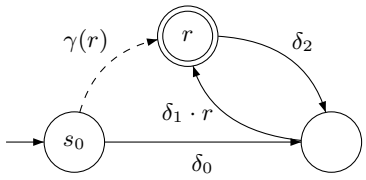


Fig. 3. Finding γ_0

r is repeated, t is well-defined. Since γ_0 is an accepting path, $\delta_2\delta_1$ is simple, so the first occurrence of t has to be in $\gamma(r)$ and its second occurrence in $\delta_2\delta_1 r$. Hence, $\gamma(r)\delta_2\delta_1 r = \gamma(r)\gamma t\gamma'$, whence $\gamma(r)\gamma t$ is an accepting path, in contradiction with $\gamma(r)\gamma t \prec_{\text{lex}} \gamma_0 = \min_{\prec_{\text{lex}}} \mathcal{S}$.

Hence, $\gamma_0 = \gamma(r)\delta_2\delta_1 r$. Since $\gamma(r)$ is the smallest path from s_0 to r in the lexicographic ordering, it is visited by the algorithm by Lemma 3. Since r is final, the algorithm switches to the careful mode and discovers γ_0 by Lemma 4.

Assume now that `mce` has been set successively to $\gamma_0, \ldots, \gamma_{i-1}$ by the algorithm, with $i \leq k$. We show that the algorithm finds γ_i as the next counterexample. Assume by contradiction that there exists a state on γ_i which is not visited by the algorithm, and let t be the first of these states: set $\gamma_i = \gamma_i' \cdot t \cdot \gamma_i''$, where `cp` takes the value γ_i' and `DFS_MIN`(t) is not called after this assignment. Since `mce` is not empty (by the induction hypothesis, γ_0 has been found) and $|\gamma_i| < |\gamma_{i-1}|$, the test at line 4 succeeds on $s' = t$. Next, t does not close γ_i' otherwise, γ_i'' would be empty and `mce` would be updated correctly. All subsequent tests fail, hence in particular, the mode is not careful and t was already visited. If the depth of t on $\gamma_i' t$ is smaller than its stored depth `t→depth`, then the test at line 11 would succeed and `DFS_MIN`(t) would be called. Hence, t was visited with a depth smaller than $|\gamma_i' t|$ on a path δ, from s_0 to t, such that $\delta \prec_{\text{lex}} \gamma_i' \cdot t$ and $|\delta| < |\gamma_i' \cdot t|$. We distinguish two cases, according whether t belongs to the head or to the loop of γ_i (see Figures 4 and 5).

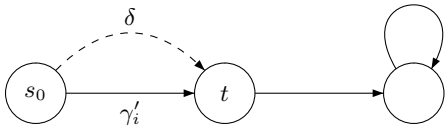


Fig. 4. First case: t on the head of γ_i

In the first case, $\delta\gamma_i''$ is an accepting path of smaller than γ_i (in size and in the lexicographic ordering), a contradiction. (Since it follows immediately from the definition of $(\gamma_i)_i$ that there is no accepting path both smaller than γ_i in size and in the lexicographic ordering.)

Therefore, t belongs to the cycle of γ_i (see Figure 5). In this case, δ followed by the cycle is again smaller than γ_i in both the lexicographic ordering and the size ordering. Hence, if it is an accepting path, we get a contradiction. Otherwise,

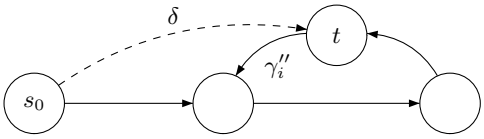


Fig. 5. Second case: t on the cycle of γ_i

this implies that there is some state appearing both in δ and in the loop. Arguing as for the proof that γ_0 is found, we find an accepting path smaller than γ_i in both the lexicographic ordering and the size ordering, which concludes the proof. □

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 1) in order to find a first counter-example. The second algorithm on the other hand finds the shortest counter example 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 counter-example is found (if any) and then switch to the second algorithm to find the shortest counter-example. 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 counter-example 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 counter-example. 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 [5]). Therefore all accepting paths are of odd length and when minimizing the size of a counter-example 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 counter-examples exist are gathered in Table 1. In this case we compare our algorithm with `SPIN -i` which tries to reduce the size of the counter-example. Clearly `SPIN -i` does not find the shortest counter-example while we have proved in Section 3 that our algorithm does. For each experiment, we show, in addition to the size of the minimal counter-example 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 not indicated is roughly proportional to the number of states matched. We see that as expected our algorithm needs to revisit more states that SPIN's one in order to really find the minimal counter-example.

		SPIN -i	Contrex
Peterson 1	counter-example length	55	19
	states stored	80	75
	states matched	1968	9469
	time	0.030s	0.040s
Peterson 2	counter-example length	33	19
	states stored	80	80
	states matched	1389	4244
Deckker 1	counter-example length	173	21
	states stored	539	417
	states matched	48593	$2.5 * 10^6$
	time	0m0.270s	12.850s
Hyman	counter-example length	97	17
	states stored	123	157
	states matched	7389	40913

Table 1. Experiments for various algorithms when a counter-example does exist

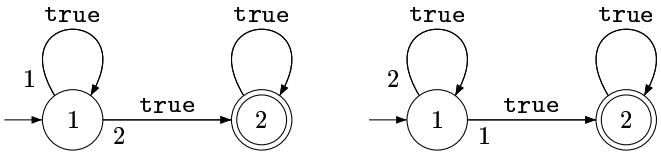
There are two versions of Peterson's and Dekker's algorithms. They only differ by the order in which the transitions are considered by the DFS algorithms. We obtained the best version just by reordering by hand the transitions of the LTL-automata. As we can see, the impact on the efficiency is important both for our algorithm and SPIN's one.

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 counter-example, without trying to minimize its length, which is more efficient than SPIN's one. It avoids unnecessary revisits of states and hence finds a counter-example more quickly. It also finds a shorter counter-example 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 one before searching for a minimal counter-example should improve the performance of our second algorithm.

Finding a shortest counter-example 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.

Experiments have shown that the order in which transitions are considered by DFS algorithms influences strongly the performances. This is easy to explain. Assume that an LTL formula is described by the automata below which only differ by the order in which transitions from state 1 will be considered in DFS algorithms. With the first automaton the whole model of the system will be explored since the loop on state 1 is visited first. On the contrary the second automaton may find a counter-example immediately. It seems that ordering the transitions of the LTL automaton so that accepting states are visited as soon as possible gives good results. This should be further investigated and automatized.



Finally, it would be interesting to find ways to minimize the length of the counter-example with respect to the model and the LTL specification. Instead, existing algorithms search for counter-examples 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 counter-example for the LTL formula.

References

1. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.
2. 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.
3. 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.
4. 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.
5. G. Holzmann. *The SPIN model-checker*. Addison-Wesley, 2003.
6. G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, Rutgers, Piscataway, NJ, 1996. American Mathematical Society.
7. G. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 2(3):270–278, November 1999.
8. Peterson. Myths about the mutual exclusion problem. In *Info. Proc. Lett*, 1981.

9. 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.