Tarjan's Algorithm Makes On-the-Fly LTL Verification More Efficient

Jaco Geldenhuys and Antti Valmari

Tampere University of Technology, Institute of Software Systems
PO Box 553, FIN-33101 Tampere, FINLAND
{jaco,ava}@cs.tut.fi

Abstract. State-of-the-art algorithms for on-the-fly automata-theoretic LTL model checking make use of nested depth-first search to look for accepting cycles in the product of the system and the Büchi automaton. Here we present a new algorithm based on Tarjan's algorithm for detecting strongly connected components. We show its correctness, describe how it can be efficiently implemented, and discuss its interaction with other model checking techniques, such as bitstate hashing. The algorithm is compared to the old algorithms through experiments on both random and actual state spaces, using random and real formulas. Our measurements indicate that our algorithm investigates at most as many states as the old ones. In the case of a violation of the correctness property, the algorithm often explores significantly fewer states.

1 Introduction

Explicit-state on-the-fly automata-theoretic LTL model checking relies on two algorithms: the first for constructing an automaton that represents the negation of the correctness property, and the second for checking that the language recognized by the product of the system and the automaton is empty. This amounts to verifying that the system has no executions that violate the correctness property. An algorithm for converting LTL formulas to Büchi automata was first described in [26], and many subsequent improvements have been proposed [4,7, 10,11,19,21].

Checking the emptiness of the product automaton requires checking that none of its cycles contain any accepting states. One approach to this problem is to detect the *strongly connected components* (SCC) of the product. An SCC is a maximal set C of states such that for all $s_1, s_2 \in C$ there is a path from s_1 to s_2. An SCC is said to be *nontrivial* if it contains at least one such nonempty path, and, conversely, an SCC is *trivial* when it consists of a single state without a self-loop. The two standard methods of detecting SCCs are Tarjan's algorithm [1, 22], and the double search algorithm attributed to Kosaraju and first published in [20]. Both approaches often appear in textbooks.

Unfortunately, both algorithms have aspects that complicate their use in on-the-fly model checking. Tarjan's algorithm makes copious use of stack space,

K. Jensen and A. Podelski (Eds.): TACAS 2004, LNCS 2988, pp. 205–219, 2004.

```
    DFS(s)
1   MARK(⟨s, 0⟩)
2   for each successor t of s do
3     if ¬MARKED(⟨t, 0⟩) then
4       DFS(t)
5     endif
6   endfor
7   if ACCEPTING(s) then
8     seed := s ; NDFS(s)
9   endif

    NDFS(s)
10  MARK(⟨s, 1⟩)
11  for each successor t of s do
12    if ¬MARKED(⟨t, 1⟩) then
13      NDFS(t)
14    else if t = seed then
15      report violation
16    endif
17  endfor
```

Fig. 1. The nested depth-first search algorithm of [2]

while Kosaraju's algorithm needs to explore transitions backwards. Instead, state-of-the-art algorithms perform nested depth-first searches [2,15].

In this paper, we introduce a new algorithm for detecting accepting cycles. Although it is based on Tarjan's algorithm, its time and memory requirements are often smaller than those of its competitors, because it relies on a single depth-first search, and it tends to detect violations earlier.

The rest of this paper is organized as follows: In Section 2 we describe the standard nested depth-first search algorithm, and discuss its strengths and weaknesses. Section 3 contains our proposed new algorithm and details about its implementation, correctness, and measurements with random graphs and random and actual LTL formulas. Section 4 deals with heuristics for speeding up the detection of violations. Experimental results on a real system are described in Section 5, and, finally, Section 6 presents the conclusions.

2 The CVWY Algorithm

Courcoubetis, Vardi, Wolper, and Yannakakis [2] presented the nested depth-first search algorithm for detecting accepting cycles, shown in Figure 1. (In the rest of this paper we shall refer to this algorithm by the moniker "CVWY".) The algorithm is based on a standard depth-first search. When a state has been fully explored and if the state is accepting, a second search is initiated to determine whether it is reachable from itself and in this way forms an accepting cycle.

The algorithm is clearly suited to on-the-fly verification. For each state, only two bits of information is required to record whether it has been found during the first and during the second search. Even when Holzmann's bitstate hashing technique [14] is used, hash collisions will not cause the algorithm to incorrectly report a violation.

A disadvantage of CVWY is that it does not find violations before starting to backtrack. Because depth-first search paths can grow rather long, this implies that many states may be on the stack at the time the first violation is detected, thus increasing time and memory consumption, and producing long counterexamples. It is not easy to change this behaviour, because it is important for the

correctness of CVWY that, when checking an accepting state for a cycle, accepting states that are "deeper" in the depth-first search tree have already been investigated.

Tarjan's algorithm also detects completed SCCs starting with the "deepest", but it performs only a single depth-first search and can already detect states that belong to the same SCC "on its way down", which is why it needs to "remember" the states by placing them on a second stack. Such early detection of partial SCCs, and by implication, of accepting cycles, is desirable, because intuitively it seems that, not only could it reduce memory and time consumption, but could also produce smaller counterexamples.

3 Cycle Detection with Tarjan's Algorithm

Tarjan's algorithm is often mentioned but dismissed as too memory consuming to be considered useful, as, for example, in [3]. In the presentation of Tarjan's algorithm in [1] states are placed on an explicit stack in addition to being placed on the implicit *procedural* stack—that is, the runtime stack that implements procedure calls. Moreover, a state remains on the explicit stack until its entire SCC has been explored. Only when the depth-first search is about to leave a state (in other words, the state has been fully explored) and the algorithm detects that it is the root of an SCC, the state and all its SCC members are removed from the explicit stack. In consequence, the explicit stack may contain several partial SCCs—many more states than the implicit depth-first stack. In addition, each state has two associated attributes: its depth-first number and its *lowlink* value; naturally this requires extra memory.

However, we believe it is wrong to dismiss Tarjan's algorithm so quickly. Firstly, it is true that the state space of the system under investigation often forms a single, large SCC. However, the product automaton in which the accepting SCC must be detected is often broken into many smaller SCCs by the interaction of the system with the Büchi automaton. Even when the system and the Büchi automaton each consist of a single SCC it is still possible that the product will have more than one component as the Büchi transitions are enabled and disabled by the values of the atomic propositions. Secondly, and most importantly, for the automata-theoretic approach to work it is unnecessary to compute the entire SCC of a violating accepting cycle—it suffices to *detect* a nontrivial SCC that contains an accepting state. And lastly, as we show below, much of the memory requirements which at first sight may seem daunting, can be avoided by a careful implementation of the algorithm.

While these arguments do not constitute a claim that Tarjan's algorithm is necessarily viable, they do open the door to investigating its potential.

3.1 The New Algorithm

An automata-theoretic verification algorithm based on Tarjan is shown in Figure 2. We have shown a lot of detail to be able to describe how memory is used

```
    Stack = record
 1    state      # state stored in stack entry
 2    lasttr     # last explored transition
 3    lowlink    # lowlink value of this entry
 4    pre        # DFS predecessor
 5    acc        # accepting state link
    endrecord

 6  Stack stack[0...]
 7  int top := −1          # top of SCC stack
 8  int dftop := −1        # top of DFS stack
 9  bool violation := false

    MAIN()
10  PUSH(initial state)
11  while ¬violation ∧ dftop ≥ 0 do
12    s := stack[dftop].state
13    t := next enabled transition of s
14    stack[dftop].lasttr := t
15    s' := successor of s by t, if any
16    if t = "none" then
17      POP()
18    else if ¬MARKED(s') then
19      PUSH(s')
20    else if s' is on stack then
21      k := position of s' on stack
22      LOWLINKUPDATE(dftop, k)
23    endif
24  endwhile
25  if violation then
26    report violation
27  endif

    PUSH(s)
28  MARK(s)
29  INCR(top)
30  stack[top].state := s
31  stack[top].lasttr := "none"
32  stack[top].lowlink := top
33  stack[top].pre := dftop
34  if ACCEPTING(s) then
35    stack[top].acc := top
36  else if dftop ≥ 0 then
37    stack[top].acc := stack[dftop].acc
38  else
39    stack[top].acc := −1
40  endif
41  dftop := top

    POP()
42  p := stack[dftop].pre
43  if p ≥ 0 then
44    LOWLINKUPDATE(p, dftop)
45  endif
46  if stack[dftop].lowlink = dftop then
47    top := dftop − 1
48  endif
49  dftop := p

    LOWLINKUPDATE(f, t)
50  if stack[t].lowlink ≤ stack[f].lowlink then
51    if stack[t].lowlink ≤ stack[f].acc then
52      violation := true
53    endif
54    stack[f].lowlink := stack[t].lowlink
55  endif
```

Fig. 2. New algorithm for detecting accepting cycles

efficiently. Our presentation differs from the original presentation [1,22] and from the presentation of nested search algorithms in that it is iterative and not recursive. This is only a minor difference, but it avoids a small overhead associated with non-tail recursion, makes it is easier to abort in the case of a violation, and does not impede the clarity of the presentation.

However, there are other, more significant differences—so many that we chose to prove the correctness of our algorithm from scratch in Section 3.2:

1. Tarjan's algorithm uses an implicit procedural stack to manage the depth-first search, and an explicit SCC stack to store partial SCCs. That the former is a subset of the latter is easy to see: A new state is inserted into both stacks when it is first encountered. Once it is fully explored it is removed from the depth-first stack, but remains on the SCC stack until its entire SCC can be removed. This makes it possible to use only a single stack and thread the depth-first stack through it by means of the *pre* field of the *Stack* structure (line 4), and a second pointer *dftop* (line 8) to the top element of the depth-first stack. It is an invariant property of the algorithm that $top \geq dftop$, and that $stack[k].pre < k$ for any $0 \leq k \leq top$. (Other, equivalent variations are presented in [8] and [9, Appendix D].)
2. The MARKED and MARK functions in lines 18 and 28 refer to the presence of a state in the state store. Similar routines occur in Tarjan's algorithm (states

are marked as "old" or "new") and in state space construction algorithms. It is important to note that once a state has been removed from the SCC stack, its stack attributes (such as *lowlink* and *pre*) are no longer required, and the information can be discarded.

3. Line 20 omits a test made by Tarjan's algorithm to avoid the update for descendants of s that have already been investigated ("forward edges" in depth-first search terminology).
4. Tarjan's algorithm numbers states consecutively as they are found, whereas the algorithm in Figure 2 reuses the numbers of states that belong to completed SCCs.
5. When a transition from state f to state t is encountered, Tarjan's algorithm sometimes updates the *lowlink* of f with the depth-first number of t, and sometimes with the *lowlink* value of t. However, in our algorithm it is always the *lowlink* of t that is used for the update (lines 50–55). A similar change has been described in [17].
6. The most important addition to Tarjan's original algorithm is the *acc* field, defined in line 5, initialized in lines 34–40, and used in line 51. A stack entry's *acc* field keeps track of the shallowest (that is, closest to the top of the stack) accepting state on the depth-first path that leads to that stack entry.

Changes 1–4 improve the efficiency of Tarjan's algorithm. With the addition of changes 5 and 6 the new algorithm is able to tell early whether an SCC contains an accepting state.

3.2 Correctness

To show the correctness of the algorithm in Figure 2, we make use of colours. The colours are not in any way essential to the operation of the algorithm; they are simply mental tools that help us to understand that the algorithm is correct. A state can have one of the following colours:

- *White*: the state has not been found yet;
- *Grey*: the state is on the depth-first stack, in other words, it is *dftop* or $stack[dftop].pre$, or $stack[stack[dftop].pre].pre$, or ...;
- *Brown*: the state is still on the stack, but not on the depth-first stack; and
- *Black*: the state has been removed from the stack, in other words, its SCC has been completely explored.

The colour of a state can change from white to grey when it is first encountered by the depth-first search, from grey to brown when it has been fully explored but its SCC has not, and from grey or brown to black when it and its SCC have been completely explored.

The following invariants are maintained by the algorithm:

- *I1*: If a state is grey, then all stack states above it are reachable from it.
- *I2*: If a state is brown, then the topmost grey state below it on the stack exists, and is reachable from it.

– *I3*: If a state s_0 is on the stack and can reach a state below it on the stack, then there are $k > 0$ states s_1 to s_k such that $s_0 s_1 \dots s_k$ is a path, s_1 to s_k are grey or white and the *lowlink* value of s_k is smaller than the position of s_0 on the stack.

To show that the algorithm maintains these invariants, we need the following:

Lemma 1. *Any state on the stack can reach stack*[*k*]*.lowlink, where k is the position of the state on the stack.*

Proof. Via the path by which the *lowlink* value was propagated back to k. □

We now consider the three actions the algorithm may take:

– *Forwarding* (PUSH). When the algorithm explores a transition to a new state, a white state is painted grey. Invariant *I1* is maintained because the new *dftop* becomes *top*, so no states exist above it. Other grey states can reach the old *dftop* and thus also the new state through the transition just explored. As far as *I3* is concerned: If s_0 is painted grey, the path to the state below s_0, truncated at the first non-white state after s_0, meets the requirements. If any of the s_j on s_0's path is painted grey, *I3* remains valid. Invariant *I2* is not affected.
– *Backtracking, top state becomes brown* (POP, without line 47). The state in question is in position *dftop*. To cope with *I2*, we have to show that the topmost grey state below *dftop* exists and is reachable from *dftop*. Because *dftop* is going to be painted brown instead of black, we have *stack*[*dftop*]*.lowlink* < *dftop*. By Lemma 1, there is a path from *dftop* to some lower state s_1 on the stack. If s_1 is brown, let s_2 be the nearest grey state below it. From *I2* we know that it is reachable from s_1. If s_1 is not brown, then it is grey, in which case let $s_2 = s_1$. The topmost grey state other than *dftop* is either s_2, or, by *I1*, reachable from s_2. Therefore, *I2* also holds for the new brown state. As for *I3*: If s_0 becomes brown, the claim is not affected. If some s_j on s_0's path becomes brown, then s_{j+1} must be grey since a state is not backtracked from as long as it has white children. If s_{j+1} is below s_0 in the stack, then s_j has at most s_{j+1}'s position as its *lowlink* value, and thus qualifies as the new s_k for s_0. Otherwise, s_{j+1} is a grey state above (or at) s_0 in the stack, so following the depth-first stack from s_0 to s_{j+1} and then continuing along the earlier path constitutes a path as specified by *I3*. Invariant *I1* is not affected.
– *Backtracking, top states become black* (POP, with line 47). If any of the states that are to be painted black can reach a state below *dftop*, then, by *I1*, so can *dftop*. (Note that as most one grey state is painted black in this operation.) *I3* stays valid by the same reasoning as when the top state is painted brown. Invariants *I1* and *I2* are not affected.

Lemma 2. *Any state on the stack can reach all states above it on the stack.*

Proof. A state on the stack is either grey or brown. If the state is grey, then the lemma holds by *I1*. If it is brown, then we first invoke *I2* and then *I1*. □

Lemma 3. *If a state is black, then all of its descendants are black.*

Proof. Since black states do not ever again change colour, it suffices to consider the case when new states are being painted black. If any of the newly painted states can reach a state below the then *dftop*, then, by *I1*, so can *dftop*. It then has a path as described by *I3*. Because all the states above *dftop* are brown, already *dftop*'s s_1 must be below it on the stack. Therefore, the condition on line 46 cannot hold, yielding a contradiction. Thus the newly painted states can reach only each other, or other black states which were painted earlier on. □

Finally we are in a position to prove that the algorithm operates correctly:

Theorem 1. *If the algorithm announces a violation, then there is an accepting cycle.*

Proof. By Lemma 1 t can reach $stack[t].lowlink$. When a violation is reported, by Lemma 2, $stack[t].lowlink$ can reach $stack[f].acc$ which can reach f. The transition that caused the announcement is from f to t, closing the cycle. □

Not only does the algorithm not raise false alarms, but it also has the property that it reports a violation as early as possible.

Theorem 2. *The algorithm reports a violation as soon as possible.*

Proof. Consider a nontrivial cycle with an accepting state A. If the algorithm does not terminate earlier, the following will happen. Eventually every transition of the cycle will be constructed by line 15. When the last transition of the cycle is constructed, then *dftop* is grey, and all other states in the cycle are on the stack. (They can be neither black by Lemma 3, nor white, because all transitions in the cycle are found.) Because no violation has been announced, the algorithm has never assigned to any $stack[k].lowlink$ a value smaller that $stack[k].acc$ (so no accepting state is brown). The cycle contains a transition $B \longrightarrow C$ such that B is A or above A on the stack, and C is A or below A. When the first such transition is found, the algorithm executes line 22 and announces a violation, because B's $lowlink \geq B$'s $acc \geq A$'s position $\geq C$'s position $\geq C$'s $lowlink$. □

We have now demonstrated how an on-the-fly verification algorithm based on Tarjan can be efficiently implemented, and correctly detects accepting cycles. The question that remains is, does the extra cost of keeping backtracked states on the stack outweigh the benefit of finding errors early on?

Table 1. Comparison of the new algorithm and CVWY for random graphs and random and real LTL formulas

Edge probability	Random formulas			Formulas from literature			Combined		
0.001			*7133*			*2949*			*10082*
NEW	8.71	6.74	6.83	9.68	7.30	7.86	8.99	6.91	7.13
CVWY	41.69	30.14	37.83	36.62	26.21	31.82	40.21	28.99	36.07
0.01			*7267*			*3039*			*10306*
NEW	6.15	4.08	4.67	6.40	3.85	5.50	6.22	4.01	4.91
CVWY	25.59	15.31	21.62	24.52	13.93	20.45	25.28	14.90	21.27
0.1			*7832*			*3131*			*10963*
NEW	6.38	2.90	4.10	5.69	1.62	4.74	6.19	2.53	4.28
CVWY	78.82	32.70	72.58	65.33	24.66	58.40	74.96	30.40	68.53
0.5			*8150*			*3168*			*11318*
NEW	6.02	2.16	3.20	5.11	1.08	3.89	5.76	1.86	3.40
CVWY	92.54	47.56	84.66	80.52	35.18	70.68	89.18	44.09	80.75
0.9			*8222*			*3177*			*11399*
NEW	6.04	2.06	3.00	5.57	1.11	4.30	5.91	1.80	3.36
CVWY	88.90	47.22	80.51	81.78	35.89	71.26	86.91	44.07	77.93
Combined			*38604*			*15464*			*54068*
NEW	6.62	3.50	4.29	6.45	2.93	5.22	6.57	3.33	4.55
CVWY	66.98	35.18	60.80	58.32	27.31	51.03	64.51	32.93	58.01

3.3 Comparison with CVWY by Measurements

To investigate the impact of retaining partial SCCs on the stack, the new algorithm was compared to CVWY using random graphs and both random and actual LTL formulas. The procedure described in [23] was used to generate 360 random formulas, another 94 formulas were selected from the literature (the 12 in [7], the 27 in [21], the 55 in [5]), and a further 36 formulas were taken from a personal collection of troublesome formulas. Also the negations of these formulas were added to the list, bringing the total to 980. No attempt was made to remove duplicate LTL formulas. Each formula was converted to a Büchi automaton using the LTL2BA program [10], and the number of states in the resulting Büchi automata ranged from 1 to 177. Using another procedure from [23], 75 random 100-state graphs were generated. The graph generation algorithm selects one state as the root and ensures that every other state is reachable from it.

Every graph was checked against every LTL formula; Table 1 shows the outcome of the comparison. The three major columns contain the results for the random formulas, the human-generated formulas and the combined set, respectively. The major rows divide the results according to the connectedness of the graphs; each row describes 15 graphs that were generated with the same transition probability, while the last row contains the combined results for all 75 graphs. Within each cell, the small number in italics indicates in how many cases violations were found. For both the new and the CVWY algorithm three numbers indicate the number of unique states reached, the number of transi-

tions explored, and the maximum size of the stack, averaged over the instances of violations, and expressed as a percentage of the total number of states or transitions in the product of the graph and the Büchi automaton. This means that every number in the last column is the sum of its counterparts in the first two columns, weighted with the number of violations.

For example, the 720 random formulas (360 randomly generated and negated) were checked against the 15 random graphs with a transition probability of 0.001. Of the 10800 products, 7133 contained violations. The new algorithm reported a violation (there may be several in each product) after exploring on average 8.71% of the states and 6.74% of the transitions of the product. During the search, the stack contained a maximum of 6.83% of the states on average. (This refers to the total number of states on the combined stack described in Section 3.1, not just the depth-first stack.) In contrast, the CVWY algorithm reports a violation after exploring on average 41.69% of the states and 30.14% of the transitions, and during the search the stack contained at most 37.83% of the states on average. (This includes the usual depth-first and the nested depth-first search stacks.)

The product automaton explored by the CVWY algorithm may, in some sense, have up to twice as many states and transitions as that explored by the new algorithm. Each state s has a depth-first version $\langle s, 0\rangle$ and, if the state is reachable from an accepting state, a nested depth-first version $\langle s, 1\rangle$, and the same holds for the transitions. However, in our opinion the figures as reported in the table give an accurate idea of the memory consumption of the algorithms (indicated by the percentage of states and maximum stack size) and the time consumption (indicated by the percentage of transitions). When states are stored explicitly, it is possible to represent the nested version of each state by storing only one additional bit per state. In this case, the CVWY figures for states may be divided by two before reading the table, to get a lower estimate.

The results clearly demonstrate that the new algorithm is faster (i.e., explores fewer transitions) and more memory efficient (i.e., stores fewer visited states and uses less stack space) than the CVWY algorithm, *for the formulas and random graphs investigated*. From experience we know that results on random graphs can be misleading; results for actual models and formulas are discussed in Section 5.

3.4 State Storage and Partial Orders

The new algorithm is fully compatible with bitstate hashing [14]; in fact, while the CVWY algorithm stores two bits per state, the new algorithm needs only a single bit. As we explain in the next subsection, the stack bit can be avoided if states can be searched for in the stack. The algorithm also works with state space caching [13]. Only the states on the depth-first stack need to be preserved; the other stack states are replaceable.

Partial order (or stubborn set) reduction techniques have become well-known and widely used [12,18,25]. Unfortunately, the CVWY algorithm has a drawback in this regard: Because states are visited more than once during the nested depth-first search, the reduction may cause the algorithm to ignore transitions that lead to an acceptance state. This was pointed out in [15]; the authors proposed

a modification that not only corrects the problem, but also improves the performance of the algorithm slightly. (We shall refer to the modified algorithm as "HPY".) However, the modification requires extra information about transition reductions to be stored along with each state. This information can be reduced to a single bit, but at the expense of a loss in reduction. The new algorithm avoids these problems by never investigating a state more than once.

3.5 Stack Issues

While each stack element of the new algorithm carries three extra fields of information (*pre*, *lowlink*, and *acc*), the space required to store this information is small compared to the size of the states in a large system. Furthermore, since every state encountered by the algorithm is inserted in the state store (by the MARK function), the stack only needs to record a reference to the state's position in the store; it is not necessary to duplicate the state on the stack. When many partial-SCC states are retained on the stack, memory requirements can be made less severe if such lightweight stack elements are used. Note that the *lasttr* field does not count as extra information. In a recursive implementation like that in Figure 1, it is stored in a local variable on the procedural stack.

Another important implementation issue is finding states on the stack. If states are stored explicitly, there is the option of storing its stack position along with each state. As with the extra fields of the stack, the extra space required for this is usually relatively small compared to the size of states. Alternatively, a supplementary data structure such as a hash table or binary tree could help to locate states in the stack. This is necessary when bitstate hashing is used.

A significant difference between the new and the older algorithms is that CVWY/HPY can store stack information in sequential memory, which can be swapped to disk. The new algorithm, on the other hand, needs to store the *lowlink* fields of stack states in random access memory. Unfortunately, the impact of this difference depends on the structure of the model, and is difficult to judge

4 Heuristics

It is not difficult to construct a scenario where the new algorithm fares worse than the CVWY algorithm. Consider the Büchi automaton B and the system S in Figure 3. The product has exactly the same shape as S except that the state marked β is accepting, and forms the single accepting cycle.

Assuming that transitions are explored left-to-right, the CVWY algorithm detects the accepting cycle after exploring the four states of the subgraph rooted at state α. Its stack reaches a maximum depth of four and at the time of the detection, its stack contains three states (two regular states and one nested-search state). The new algorithm, on the other hand, also explores the four states of the subgraph rooted at α, but, because they form an SCC rooted at the initial state, these states remain on the stack. At the time of detecting the accepting cycle, the new algorithm's stack contains all six states of the product.

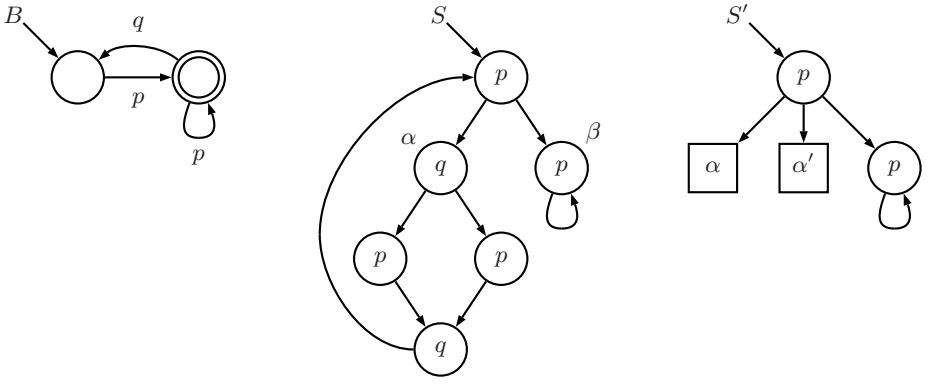

Fig. 3. A difficult case for the new algorithm

The situation is even worse if the system contains two such subgraphs, as does S' in Figure 3. In this case, CVWY explores the subgraphs at α and α', but its stack reaches a maximum size of only four. The new algorithm retains all states on the stack, so that it contains ten states when the accepting cycle is found.

If only the transition leading to β were explored first, both algorithms would detect the offending cycle after exploring just two states (plus an extra nested-search state for CVWY). This suggests the use of heuristics to guide the algorithms to detect accepting cycles more quickly. Ten heuristics were investigated, and the results are shown in Table 2. The meaning of the three columns is the same as in Table 1, as is the meaning of the three numbers (states, transitions, maximum stack size) given per experiment. Only the performance for the new algorithm is described in the table, and the first line—which agrees with the last NEW line of Table 1—shows the case where no heuristics are used.

The following heuristics were investigated: +DEPTH (−DEPTH) selects those transitions that lead to the deepest (shallowest) Büchi SCC first, +ACCEPT (−ACCEPT) selects those transitions that move closest to (furthest away from)

Table 2. Effect of heuristics on the new algorithm

Heuristic	Random formulas			Formulas from literature			Combined		
NONE	6.62	3.50	4.29	6.45	2.93	5.22	6.57	3.33	4.55
+DEPTH	6.72	3.34	4.28	7.88	3.46	6.29	7.05	3.37	4.85
−DEPTH	11.40	5.95	8.98	14.86	7.47	13.33	12.39	6.38	10.23
+ACCEPT	13.04	6.82	10.15	16.47	8.16	14.71	14.02	7.20	11.46
−ACCEPT	7.62	3.62	5.80	10.25	4.97	8.97	8.37	4.01	6.71
+STAY	12.03	6.15	9.76	16.27	8.04	14.65	13.24	6.69	11.16
−STAY	8.30	3.99	6.10	12.06	5.63	10.55	9.37	4.46	7.37
+TRUE	9.03	4.39	6.58	12.91	5.95	11.03	10.14	4.83	7.85
−TRUE	13.16	6.50	10.84	17.17	8.27	15.27	14.31	7.00	12.11

an accepting state first, +STAY (−STAY) selects those transitions that stay within the same Büchi SCC first (last), and +TRUE (−TRUE) selects those transitions that are labelled with the formula "true" first (last). If there are any ties between transitions, the order in which the transitions appear in the input is followed.

As the table shows, none of the heuristics performed better than using no heuristic at all. This is disappointing, but it does not mean that heuristics do not work. Rather, the problem might be that some heuristics work well for some Büchi automata, and poorly for others. Suggestions for heuristics search based on system transitions have been made in [13,16], and, more recently, in [6,27]. The new algorithm presented in this paper can accommodate all these suggestions.

5 Experiments with Actual Models

We have implemented a model of the echo algorithm with extinction for electing leaders in an arbitrary network, as described in [24, Chapter 7]. Three variations of the model behave in different ways:

- *Variation 1*: After a leader has been elected and acknowledged by the other nodes, the leader abdicates and a new election is held. The same node wins every election.
- *Variation 2*: A leader is elected and abdicates, as in Variation 1. However, a counter keeps track of the previous leader and gives each node a turn to win the election.
- *Variation 3*: As in Variation 2, each node gets a turn to become leader. However, one node contains an error that disrupts the cycle of elections.

Each of the variations was modelled with the SPIN system and its state space, reduced with partial orders, was converted to a graph for input by our implementation of the cycle detection algorithms. This is not the way the algorithms would be used in practice—cycle detection normally runs concurrently with the generation of the state space—but it facilitates making the experiments without having to implement the new algorithm in SPIN.

The results of our comparison are shown in Table 3. The first column contains the names of the formulas which are given explicitly below the table; a cross to the left of the formula name indicates that a violation of the property was detected. The column marked "Product" gives the number of states and transitions in the product automaton, and the columns marked "NEW", "HPY", and "CVWY" give the number of states, transitions and the maximum stack size for the new algorithm, the algorithm in [15] (HPY), and the algorithm in [2] (CVWY), respectively. As mentioned in Section 3.4, the HPY algorithm improves on the performance of the CVWY algorithm in some of the cases.

The arbitrary network specified in the model variations comprised three nodes numbered 0, 1, and 2. This explains why property B ("if ever, in the distant enough future, there is no leader, node 3 will be elected") and E ("node 3 is eventually elected once") are not satisfied by any of the models. Properties A, C, D, F, and G deal with the election of node 0, and properties H and I say

Table 3. Results of checking property ϕ for leader election in an arbitrary network

	Variation 1: 25714 states, 32528 transitions											
	ϕ	Product		NEW			HPY			CVWY		
	A	51053	96406	51053	96406	14097	51053	128342	14097	51053	128342	14097
×	B	51849	100521	422	436	422	1159	1227	1151	1159	1227	1151
	C	12081	15198	12081	15198	199	12081	30395	199	12081	30395	199
	D	51053	96406	51053	96406	14097	51053	128342	14097	51053	128342	14097
×	E	25714	32529	389	390	389	642	657	639	692	730	639
	F	51053	96406	51053	96406	14097	51053	128342	14097	51053	128342	14097
	G	25714	32529	25714	32529	14097	25714	32529	14097	25714	32529	14097
×	H	53988	88553	610	624	610	19649	24417	1511	19649	24417	1511
×	I	53988	88553	610	624	610	19649	24417	1511	19649	24417	1511

	Variation 2: 51964 states, 65701 transitions											
	ϕ	Product		NEW			HPY			CVWY		
×	A	104779	204228	26742	49909	13841	35786	87539	1472	35786	87539	1472
×	B	105599	208455	886	900	886	3016	3168	3008	3016	3168	3008
	C	12081	15198	12081	15198	199	12081	30395	199	12081	30395	199
	D	103541	195893	103541	195893	40347	103541	260986	40347	103541	260986	40347
×	E	51964	65702	853	854	853	1570	1585	1567	1620	1658	1567
	F	103541	194225	103541	194225	40347	103541	259318	40347	103541	259318	40347
×	G	92140	118238	1122	1123	1122	1803	1805	1803	1803	1805	1803
×	H	293914	552899	1567	1581	1567	56254	69736	4269	56281	69777	4295
	I	132763	222806	132763	222806	40347	211890	377639	40347	211890	377639	40347

	Variation 3: 40158 states, 51115 transitions											
	ϕ	Product		NEW			HPY			CVWY		
×	A	81167	160470	904	906	904	2133	2222	2132	2133	2222	2132
×	B	81987	164697	904	906	904	2133	2222	2132	2133	2222	2132
	C	12081	15198	12081	15198	199	12081	30395	199	12081	30395	199
×	D	79929	152135	12777	16125	697	13239	31784	1159	13239	31784	1159
×	E	40158	51116	697	698	697	1159	1161	1159	1159	1161	1159
×	F	79929	150467	12777	16125	697	13239	31784	1159	13239	31784	1159
×	G	66450	86258	903	904	903	1365	1367	1365	1365	1367	1365
×	H	229269	435261	30917	57851	2009	142993	271769	2516	169718	315617	2896
×	I	169793	312465	37798	63689	14688	150362	281812	2292	193866	340240	2993

$A \equiv \Diamond\Box(n \,\mathrm{U}\, \ell_0)$
$B \equiv \Diamond\Box(n \,\mathrm{U}\, \ell_3)$
$C \equiv \Diamond\ell_0$
$D \equiv \Box\Diamond\ell_0$
$E \equiv \Diamond\ell_3$
$F \equiv \Box(n \Rightarrow \Diamond\ell_0)$
$G \equiv \Box(n \vee \ell_0)$
$H \equiv \Diamond\Box((n \vee \ell_0 \vee \ell_1 \vee \ell_2) \wedge L(0,1) \wedge L(1,2) \wedge L(2,0))$
$I \equiv \Diamond\Box((n \vee \ell_0 \vee \ell_1 \vee \ell_2) \wedge L(0,2) \wedge L(1,0) \wedge L(2,1))$

$n \equiv$ there is no leader
$\ell_x \equiv$ process x is the leader
$L(x,y) \equiv \ell_x \Rightarrow (\ell_x \,\mathrm{U}\, (n \,\mathrm{U}\, \ell_y))$

that, from some point onward, the elected leader follows the cycles 0–1–2 and 0–2–1, respectively.

In all cases the number of states and transitions explored by the new algorithm were the same as or less than those explored by the others. In three cases, variation 1 H and I and 2 H, the new algorithm explored more than 30 times fewer states and transitions. In two cases, 2 A and 3 I, the new algorithm required more stack entries than the other algorithms. As discussed in Section 3.5, which algorithm wins in these two cases depends on implementation details, but the new algorithm is clearly the overall winner.

6 Conclusions

We have presented an alternative to the CVWY [2] and HPY [15] algorithms for cycle detection in on-the-fly verification with Büchi automata. Our algorithm produces a counterexample as soon as an ordinary depth-first search has found every transition of the cycle. Thus it is able to find counterexamples quicker than CVWY and HPY, which need to start backtracking first. Also, it never investigates a state more than once, making it compatible with other verification techniques that rely on depth-first search. It sometimes requires a lot of stack space, but our measurements indicate that this drawback is usually outweighed by its ability to detect errors quickly.

Acknowledgments. The work of J. Geldenhuys was funded by the TISE graduate school and by the Academy of Finland.

References

1. A. V. Aho, J. E. Hopcroft, & J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
2. C. Courcoubetis, M. Y. Vardi, P. Wolper, & M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. In *Proc. 2nd Intl. Conf. Computer-Aided Verification*, LNCS #531, pp. 233–242, Jun 1990.
3. C. Courcoubetis, M. Y. Vardi, P. Wolper, & M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design* 1(2/3), pp. 275–288, Oct 1992.
4. M. Daniele, F. Giunchiglia, & M. Y. Vardi. Improved automata generation for linear time temporal logic. In *Proc. 11th Intl. Conf. Computer-Aided Verification*, LNCS #1633, pp. 249–260, Jul 1999.
5. M. B. Dwyer, G. S. Avrunin, & J. C. Corbett. Property specification patterns for finite-state verification. In *Proc. 2nd ACM Workshop Formal Methods in Software Practice*, pp. 7–15, Mar 1998.
   Related site: http://www.cis.ksu.edu/santos/spec-patterns, Last updated Sept 1998.
6. S. Edelkamp, S. Leue, & A. Lluch Lafuente. Directed explicit-state model checking in the validation of communication protocols. Tech. Rep. 161, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Oct 2001.

7. K. Etessami & G. J. Holzmann. Optimizing Büchi automata. In *Proc. 11th Intl. Conf. Concurrency Theory*, LNCS #1877, pp. 154–167, Aug 2000.
8. J. Eve & R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Informatica* 8, pp. 303–314, 1977.
9. H. N. Gabow. Path-based depth-first search for strong and biconnected components. Tech. Rep. CU–CS–890–99, Dept. Computer Science, Univ. of Colorado at Boulder, Feb 2000.
10. P. Gastin & D. Oddoux. Fast LTL to Büchi automata translation. In *Proc. 13th Intl. Conf. Computer-Aided Verification*, LNCS #2102, pp. 53–65, Jul 2001.
11. R. Gerth, D. Peled, M. Y. Vardi, & P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th IFIP Symp. Protocol Specification, Testing, and Verification*, pp. 3–18, Jun 1995.
12. P. Godefroid. *Partial-order Methods for the Verification of Concurrent Systems, An Approach to the State-explosion Problem*. PhD thesis, University of Liège, Dec 1994. Also published as LNCS #1032, 1996.
13. P. Godefroid, G. J. Holzmann, & D. Pirottin. State space caching revisited. In *Proc. 4th Intl. Conf. Computer-Aided Verification*, LNCS #663, pp. 175–186, Jun 1992.
14. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, 1991.
15. G. J. Holzmann, D. Peled, & M. Yannakakis. On nested depth first search. In *Proc. 2nd Spin Workshop*, pp. 23–32, Aug 1996.
16. F. J. Lin, P. M. Chu, & M. T. Liu. Protocol verification using reachability analysis: the state space explosion problem and relief strategies. *Computer Communication Review* 17(5), pp. 126–135, Oct 1987.
17. E. Nuutila & E. Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters* 49(1), pp. 9–14, Jan 1994.
18. D. Peled. All from one, one for all: On model checking using representatives. In *Proc. 5th Intl. Conf. Computer-Aided Verification*, LNCS #697, pp. 409–423, Jun 1993.
19. K. Schneider. Improving automata generation for linear temporal logic by considering the automaton hierarchy. In *Proc. Logic for Programming, Artificial Intelligence, and Reasoning*, LNCS #2250, pp. 39–54, Dec 2001.
20. M. Sharir. A strong-connectivity algorithm and its application in data flow analysis. *Computer and Mathematics with Applications* 7(1), pp. 67–72, 1981.
21. F. Somenzi & R. Bloem. Efficient Büchi automata from LTL formulae. In *Proc. 12th Intl. Conf. Computer-Aided Verification*, LNCS #1855, pp. 248–267, Jun 2000.
22. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing* 1(2), pp. 146–160, Jun 1972.
23. H. Tauriainen. A randomized testbench for algorithms translating linear temporal logic formulae into Büchi automata. In *Proc. Workshop Concurrency, Specifications, and Programming*, pp. 251–262, Sept 1999.
24. G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2000.
25. A. Valmari. A stubborn attack on state explosion. *Formal Methods in System Design* 1(1), pp. 297–322, 1992.
26. P. Wolper, M. Y. Vardi, & A. P. Sistla. Reasoning about infinite computation paths. In *Proc. Symp. Foundations of Computer Science*, pp. 185–194, Nov 1983.
27. C. H. Yang & D. L. Dill. Validation with guided search of the state space. In *Proc. 35th ACM/IEEE Conf. Design Automation*, pp. 599–604, Jun 1998.