Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 96
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 96
Espoo 2005 HUT-TCS-A96

A NOTE ON THE WORST-CASE MEMORY REQUIREMENTS OF GENERALIZED NESTED DEPTH-FIRST SEARCH

Heikki Tauriainen

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Helsinki University of Technology Laboratory for Theoretical Computer Science
Research Reports 96
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 96
Espoo 2005 HUT-TCS-A96

A NOTE ON THE WORST-CASE MEMORY REQUIREMENTS OF GENERALIZED NESTED DEPTH-FIRST SEARCH

Heikki Tauriainen

Helsinki University of Technology
Department of Computer Science and Engineering
Laboratory for Theoretical Computer Science

Teknillinen korkeakoulu
Tietotekniikan osasto
Tietojenkäsittelyteorian laboratorio

Distribution:
Helsinki University of Technology
Laboratory for Theoretical Computer Science
P.O.Box 5400
FI-02015 TKK
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-7874-X
ISSN 1457-7615

Multiprint Oy
Helsinki 2005

ABSTRACT: This report presents a new nested depth-first search algorithm for testing the emptiness of a language accepted by a generalized finite Büchi automaton. Given an automaton with n states, $m \geq 1$ sets of accepting transitions and s bits in a state descriptor, the algorithm decides whether the language accepted by the automaton is empty in at most $m + 1$ passes of the state space, using $n\bigl(s + \lceil\log_2(m+1)\rceil\bigr)$ bits of memory for the search hash table in the worst case. In addition to the standard search stacks, the algorithm also needs $O\bigl(m \log_2(nm)\bigr)$ bits of memory for bookkeeping.

KEYWORDS: Büchi automata, emptiness checking, nested depth-first search

CONTENTS

1 INTRODUCTION

This research report is a follow-up to the article [9], which proposes an algorithm for testing the emptiness of the language accepted by a finite Büchi automaton with multiple acceptance conditions (i.e., sets of "accepting" transitions), generalizing the classic *nested depth-first search* algorithm of Courcoubetis et al. [1]. The algorithm presented in the article [9] has better worst-case resource requirements (memory usage and number of states explored) than the alternative approach based on applying the classic algorithm to a nongeneralized automaton obtained by an intermediate automaton transformation [1, 3]. The reason for the improvement in the resource requirements is that the reduction in the number of acceptance conditions may increase the size of the automaton.

Similarly to the classic nested depth-first search algorithm, the generalized search proposed in the article [9] is based on a *top-level search* that initiates *second searches* in the automaton. The key observation leading to the improvement in resource usage is that these second searches never need to explore parts of the automaton which have not been found in the top-level search. As an optimization of the basic generalized algorithm, the article [9] proposes also a version of the algorithm ([9], Figure 6) which tries to further reduce the number of states explored by (essentially) combining second searches together if possible, guided by the accepting transitions "seen" during second searches. This optimization is called the *condition heuristic* in the article [9].

The generalized search algorithm associates each state of the automaton with an initially empty set of acceptance conditions that is populated during the search with conditions fulfilled by paths to the state from other states in the automaton. In this article, we develop the condition heuristic further by replacing the sets of acceptance conditions with *counters* that are incremented during the search according to an ordering given for the conditions. Consequently, the number of additional bits to be stored in the search hash table with each state reduces from linear to logarithmic in the number of acceptance conditions. However, because the change alters the behavior of the algorithm by possibly limiting the extent of second searches, it must be checked that the modified algorithm remains complete. We provide a new, more structured proof of the completeness of the algorithm. Additionally, we incorporate the useful optimization proposed by Couveur et al. [2] into the new algorithm to take full advantage of transition-based acceptance. (The generalized algorithm presented in the article [9] waits for all successors of a state to have been explored in the top-level search before considering second searches from the state. As suggested by Couvreur et al. [2], this is actually not necessary: a second search can be started from a transition immediately after it and all previously unvisited states reachable by it have been explored.)

2 EMPTINESS CHECKING ALGORITHM

Let $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ be a *generalized Büchi automaton* over an *alphabet* Σ (with a finite set of *states* Q, *transitions* $\Delta \subseteq Q \times \Sigma \times Q$, *initial* *state* $q_I \in Q$, and a finite nonempty set of *acceptance conditions* $\mathcal{F} = \{F_1, F_2, \ldots, F_n\} \subseteq 2^{\Delta}$, $\mathcal{F} \neq \emptyset$). The indices of the acceptance conditions induce an ordering between the conditions. The objective is to check whether the language accepted by $\mathcal{A}$ is not empty, i.e., whether $\mathcal{A}$ has a cycle that is reachable from its initial state q_I and *fulfills* each condition F_i (includes a transition from each F_i). The new variant of the language emptiness checking algorithm for generalized Büchi automata is shown in pseudocode in Figure 1. We present the algorithm in recursive form, from which it is easy to point out the relevant details.

The TOP-LEVEL-SEARCH procedure implements a simple depth-first search in the automaton. After processing a transition (line 7), a second search is started from the transition at line 9 [2] (in further discussion, the term "second search" refers to all operations caused by a call to the SECOND-SEARCH procedure made at line 9). Similarly to the basic generalized algorithm, the purpose of this search is to propagate information about the reachablility of states via paths which fulfill certain acceptance conditions. Each second search is restricted to states that have been found during the top-level search. The second search behaves almost identically to the basic version augmented with the condition heuristic ([9], Figure 6); the main change is in the condition on when to advance the search to another state in the automaton (lines 17–18). We shall discuss this change in more detail after introducing the main data structures used by the algorithm.

The algorithm uses the following (global) data structures:

$path\_stack$: This stack collects the states entered in the top-level search but from which the search has not yet backtracked. This stack is used only to facilitate the correctness proof in Section 3 and thus all references to it (lines 3 and 12) could be eliminated from the pseudocode.

$visited$: States found in the top-level search. When a state is added to this set, it will never be removed.

$count$: A lookup table associating each state of the automaton with an index of an acceptance condition. Intuitively, if $count[q] = c$ holds for some $q \in Q$ and $1 \leq c \leq n$, then, for every acceptance condition $F_1, F_2, \ldots, F_c \in \mathcal{F}$, there exists a nonempty path to the state q in the automaton such that the path fulfills the condition. For all $q \in Q$, $count[q]$ will never decrease during the execution of the algorithm.

$\mathcal{I}_{\text{seen}}$: A set of indices of acceptance conditions "seen" in a path from the source state of a transition from which a second search was started at line 9 to the state referred to by the program variable q in a nested recursive call of the SECOND-SEARCH procedure. The conditions "seen" in this path include also the conditions that were associated with the source state of the search at the beginning of the search (line 8). In each nested call to the SECOND-SEARCH procedure, the algorithm first collects the indices of all previously "unseen" acceptance conditions fulfilled by the transition given as a parameter for the procedure (line 16). These indices are then added to the $\mathcal{I}_{\text{seen}}$ set later at line 21 before any nested recursive calls to the procedure.

Initialize: $\mathcal{A} := (\Sigma, Q, \Delta, q_I, \mathcal{F})$: Nondeterministic generalized Büchi automaton with $\mathcal{F} = \{F_1, F_2, \ldots, F_n\} \subseteq 2^{\Delta}$, $\mathcal{F} \neq \emptyset$;
$\mathit{path\_stack}$:= [empty stack];
$\mathit{visited} := \emptyset$;
$\mathit{count} := [q_1 \mapsto 0, \ldots, q_{|Q|} \mapsto 0]$;
$\mathit{depth} := 1$;
$\mathit{condition\_stack}$:= [stack initialized with the element $(0, 0)$].

```
TOP-LEVEL-SEARCH($q \in Q$)
  begin
    push $q$ on $\mathit{path\_stack}$;
    $\mathit{visited} := \mathit{visited} \cup \{q\}$;
    for all $t = (q, \sigma, q') \in \Delta$ do
      begin
        if ($q' \notin \mathit{visited}$) then TOP-LEVEL-SEARCH($q'$);
        $\mathcal{I}_{\text{seen}} := \{1, 2, \ldots, \mathit{count}[q]\}$;
        SECOND-SEARCH($t$);
        if ($\mathit{count}[q] = |\mathcal{F}|$) then report "The language of $\mathcal{A}$ is not empty";
      end;
    pop $q$ off $\mathit{path\_stack}$;
  end

SECOND-SEARCH($(q, \sigma, q') \in \Delta$)
  begin
    $\mathcal{I}_{\text{unseen\_fulfilled}} := \{1 \le i \le |\mathcal{F}| \mid (q, \sigma, q') \in F_i\} \setminus \mathcal{I}_{\text{seen}}$;
    $c := \max\{0 \le i \le |\mathcal{F}| \mid j \in \mathcal{I}_{\text{seen}} \cup \mathcal{I}_{\text{unseen\_fulfilled}}$ for all $1 \le j \le i\}$;
    if ($c > \mathit{count}[q']$) then
      begin
        $\mathit{count}[q'] := c$;
        $\mathcal{I}_{\text{seen}} := \mathcal{I}_{\text{seen}} \cup \mathcal{I}_{\text{unseen\_fulfilled}}$;
        for all $i \in \mathcal{I}_{\text{unseen\_fulfilled}}$ do push $(i, \mathit{depth})$ on $\mathit{condition\_stack}$;
        $\mathit{depth} := \mathit{depth} + 1$;
        for all $t = (q', \sigma', q'') \in \Delta$ such that $q'' \in \mathit{visited}$ do SECOND-SEARCH($t$);
        $\mathit{depth} := \mathit{depth} - 1$;
        $(i, d)$ := topmost element of $\mathit{condition\_stack}$;
        while ($d = \mathit{depth}$) do
          begin
            $\mathcal{I}_{\text{seen}} := \mathcal{I}_{\text{seen}} \setminus \{i\}$;
            pop $(i, d)$ off $\mathit{condition\_stack}$;
            $(i, d)$ := topmost element of $\mathit{condition\_stack}$;
          end
      end
  end
```

Figure 1: Generalized nested depth-first search algorithm. The emptiness check is started by calling the TOP-LEVEL-SEARCH procedure with the initial state q_I of the automaton as a parameter.

depth: An integer variable representing the number of transitions in a path from the source state of the transition from which a second search was started to the state referred to by the program variable q' in a nested recursive call of the SECOND-SEARCH procedure.

condition_stack: A stack used for recording a (partial) history of changes made to the $\mathcal{I}_{\text{seen}}$ set during a second search in the automaton. To simplify the presentation of the SECOND-SEARCH procedure, the stack is initialized with a sentinel element that will never be removed from the stack (and thus the stack can never be empty at lines 26 or 31).

To simplify the presentation of the algorithm, the global variables *depth* and $\mathcal{I}_{\text{seen}}$ could be modelled as parameters of the SECOND-SEARCH procedure (in which case also the variable *condition_stack* could be eliminated). However, we treat these variables as globals to facilitate the analysis of the algorithm's memory requirements.

In the original version of the condition heuristic [9], a second search in the automaton proceeds from a state q to its successor q' in the automaton if the set of acceptance conditions associated with the state q' does not include all conditions "seen" in a path from the source state of the search to the state q'. Because we now associate states of the automaton with counters instead of sets, we make the second search proceed from a state q to its successor q' only if the conditions "seen" in a path to q' include the next condition (or possibly several consecutive conditions in the acceptance condition ordering), the index of which is not yet recorded in the value of $count[q']$. At line 17, the algorithm finds the maximal index of an acceptance condition which has been "seen" along with all of its predecessors (in the acceptance condition ordering) during the second search; whether to proceed to a successor of q is then decided at line 18.

2.1 Resource requirements

Clearly, the *count* table associates each state of the automaton with an integer that will never exceed the number of acceptance conditions (denoted by $|\mathcal{F}|$ in the following) in the automaton. Because entering a state q' during a second search (line 19) implies incrementing the value of $count[q']$, it is easy to see that the algorithm needs at most $|\mathcal{F}| + 1$ passes of the state space of the automaton. Similarly, it is easy to see that the worst-case bounds for the sizes of search stacks remain the same as in the article [9] (at most $|Q|$ elements in the top-level search stack, and, because of the condition heuristic, at most $|Q| \cdot |\mathcal{F}| + 1$ elements in the implicit recursion stack used for nested calls of the SECOND-SEARCH procedure). If the *count* table is represented as a hash table indexed with state descriptors, each of which consumes s bits of memory, the table has to store $|Q| \cdot \big(s + \lceil \log_2(|\mathcal{F}| + 1) \rceil\big)$ bits in the worst case. (The *visited* set can be combined with this table by inserting states to the table as they are entered in the top-level search.)

The $\mathcal{I}_{\text{seen}}$ set can be represented as a bit vector with $|\mathcal{F}|$ bits. Because elements are added to *condition_stack* only when there are conditions with indices still missing from the $\mathcal{I}_{\text{seen}}$ set, it is easy to see that this stack will never contain more than $|\mathcal{F}| + 1$ elements. Note that this bound would not be as obvious if the *depth* and $\mathcal{I}_{\text{seen}}$ variables were modelled as parameters of the SECOND-SEARCH procedure: in this case the information stored in *condition_stack* would be implicit in the activation records of nested recursive procedure calls (and would amount to $|Q| \cdot |\mathcal{F}| + 1$ instead of $|\mathcal{F}| + 1$ elements in the worst case). It is straightforward to check that the integer *depth*, the $\mathcal{I}_{\text{seen}}$ set and the elements in the *condition_stack* stack need $O\big(|\mathcal{F}| \log_2(|Q| \cdot |\mathcal{F}|)\big)$ bits for their representation in the worst case.

Additionally, the search procedures need (in a standard way) to keep track on how to generate the next transition starting from a state in the loops at lines 5–11 and line 24 such that the information is preserved over nested recursive calls of the procedures.

3 CORRECTNESS OF THE ALGORITHM

In this section we prove the correctness of the algorithm. We use notation and terminology from the article [9], Section 2.

Let $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ ($\mathcal{F} \neq \emptyset$) be a generalized Büchi automaton given as input for the algorithm. It is easy to see that the algorithm will start a second search at line 9 from each transition, the source state of which is reachable from the initial state q_I of the automaton, and the second search is started exactly once from each of these transitions. Therefore the transitions are processed in some well-defined order: we write $t \leq t'$ ($t, t' \in \Delta$) to indicate that the second search from t is started at line 9 before the second search from t' (or $t = t'$). (Other inequalities between transitions are defined in the obvious way.) Additionally, we write *visited*(t) and *path_stack*(t) to refer to the contents of the *visited* set and the top-level search stack (respectively) at the beginning of a second search from the transition t; note that these data structures remain unchanged in all recursive calls to the SECOND-SEARCH procedure. We also write *path_stack*(q) to denote the contents of the top-level search stack at line 13 of the algorithm when the top-level search is about to backtrack from a state $q \in Q$.

It is easy to check (by induction on the number of nested recursive calls of the SECOND-SEARCH procedure) that, if $depth = d$ and $\mathcal{I}_{\text{seen}} = \mathcal{I}$ hold for an integer d and a set of indices of acceptance conditions $\mathcal{I} \subseteq \big\{1, 2, \ldots, |\mathcal{F}|\big\}$ when the algorithm is about to enter the loop at line 24, then *depth* and $\mathcal{I}_{\text{seen}}$ will have these same values at the beginning of each iteration of the loop.

3.1 Soundness

In this section we prove the soundness of the algorithm. We first formalize the following simple fact about the states in the top-level search stack.

Lemma 1 *Let q and q' be states in path_stack such that $q = q'$, or q was pushed before q' on the stack. Then, q' is reachable from q in $\mathcal{A}$.*

Proof: The result holds trivially if $q = q'$. Otherwise q' was pushed on the stack in a recursive call of the top-level search procedure (started at line 7 when q was on top of the stack), and it is easy to see that there exists a nonempty path from q to q' in the automaton. □

The soundness of the algorithm is based on the fact that every state in the top-level search stack known to be reachable via a path fulfilling an acceptance condition is actually in a cycle that fulfills the condition.

Lemma 2 *Let $t \in \Delta$ be a transition from which the algorithm starts a second search at line 9. Assume that $count[q] \geq n$ holds for some $q \in path\_stack(t)$ and $1 \leq n \leq |\mathcal{F}|$ during a second search started at line 9 from the transition t. Then, the automaton contains a cycle that visits the state q and fulfills the acceptance condition F_n.*

Proof: We claim that there exists an integer $1 \leq k < \omega$, states $q_0, q_1, \ldots, q_k \in Q$ (with $q_0 = q$) and transitions $t_0, t_1, \ldots, t_k \in \Delta$ such that for all $0 \leq i < k$, q_i and q_{i+1} are reachable from each other via nonempty paths in the automaton, $t_i \geq t_{i+1}$ for all $0 \leq i < k$, and q_{k-1} is reachable from q_k via a path that fulfills the acceptance condition F_n. Clearly, if such sequences exist, then q_0 $(= q)$ is in a cycle that fulfills the acceptance condition F_n, and the result follows.

We prove the claim by constructing sequences with the required properties. Let $q_0 = q$, and let $t_0 = t$. Because $count[q_0] \geq n > 0$ holds during the second search from t_0, there exists a transition $t_1 \in \Delta$, $t_1 \leq t_0$, such that $count[q_0]$ was updated for the first time to a value greater than or equal to n during a second search started at line 9 from the transition t_1. Let $q_1 \in Q$ be the source state of t_1.

Assume that q_i and t_i have been defined for some $1 \leq i < \omega$ such that $count[q_{i-1}]$ is updated for the first time to a value greater than or equal to n during a second search started at line 9 from the transition $t_i \leq t_{i-1}$. If $count[q_i] \geq n > 0$ already holds at the beginning of this second search, there exists a transition $t' \in \Delta$, $t' < t_i$, such that $count[q_i]$ was updated for the first time to a value greater than or equal to n during a second search started from the transition t' (with source state $q' \in Q$). In this case we let $q_{i+1} = q'$ and $t_{i+1} = t'$ and continue the construction.

By repeating the construction, we obtain a sequence of states $q_0, q_1, q_2, \ldots$ and a sequence of transitions $t_0 \geq t_1 > t_2 > \cdots$ such that for all $i = 0, 1, 2, \ldots$, $count[q_i]$ was updated for the first time to a value greater than or equal to n in a second search started from the transition t_{i+1} with source state q_{i+1}. Because the automaton is finite, the sequence of transitions is finite, and because $count[q'] = 0$ initially holds for all $q' \in Q$, there exists an index k such that $count[q_k] < n$ holds at the beginning of a second search from the transition t_k.

Let $0 \leq i < k$. Because $count[q_i]$ is updated for the first time to a value greater than or equal to n during a second search started from the transition $t_{i+1} \leq t_i$, q_i is reachable from the source state q_{i+1} of t_{i+1} via a nonempty path, and $q_i \in visited(t_{i+1})$ holds. On the other hand, because $t_{i+1} \leq t_i$ holds, the top-level search could not have backtracked from the state q_i before backtracking from q_{i+1}, and thus $q_i \in path\_stack(t_{i+1})$ holds. Because q_{i+1} is on top of $path\_stack$ during the second search from t_{i+1}, it follows by Lemma 1 that q_{i+1} is reachable from q_i, and thus the states are reachable from each other via nonempty paths in the automaton.

Because $count[q_k] < n$ holds at the beginning of the second search from t_k, $n \notin \mathcal{I}_{\text{seen}}$ holds at line 16 when the second search procedure is called at line 9. Because $count[q_{k-1}]$ is nevertheless updated to a value greater than or equal to n during the second search from t_k (at line 20), it is easy to see that n must be inserted to $\mathcal{I}_{\text{unseen\_fulfilled}}$ during the search, and thus the second search from t_k must reach q_{k-1} via a path that contains a transition which fulfills the acceptance condition F_n. Therefore q_{k-1} and q_k are in a cycle that fulfills the condition F_n, and the result follows. $\square$

It is now easy to prove the soundness of the algorithm using Lemma 2.

Theorem 1 *Let $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ ($\mathcal{F} \neq \emptyset$) be the nondeterministic generalized Büchi automaton given as input for the algorithm. Let $t \in \Delta$ be a transition from which the algorithm starts a second search at line 9. If $count[q] = |\mathcal{F}|$ holds during this search for a state $q \in path\_stack(t)$, then $\mathcal{A}$ contains an accepting cycle reachable from the initial state q_I. In particular, this holds if the algorithm reports that the language of $\mathcal{A}$ is not empty.*

Proof: Because the top-level search is started from the initial state q_I of the automaton, $q_I \in path\_stack(t)$ certainly holds, and there exists a (possibly empty) path from q_I to q in $\mathcal{A}$ by Lemma 1. Because $count[q] \geq i$ holds for all $1 \leq i \leq |\mathcal{F}|$, it follows by Lemma 2 that for all $1 \leq i \leq |\mathcal{F}|$, the automaton contains a cycle that visits the state q and fulfills the acceptance condition F_i. Because all of these cycles share the state q, the cycles can be merged together to obtain an accepting cycle for the automaton. The soundness of the algorithm now follows from the condition at line 10 of the top-level search procedure since the program variable q refers to the topmost state of $path\_stack$ at this point. $\square$

3.2 Completeness

We now turn to the completeness of the algorithm. Again, we start by listing several basic facts about the behavior of the algorithm for future reference.

Lemma 3 *Let $t = (q, \sigma, q') \in \Delta$ be a transition from which the algorithm starts a second search at line 9. Then, $q' \in visited(t)$.*

Proof: The result follows from the condition at line 7 of the algorithm: if $q' \notin visited$ holds at this point, the algorithm calls the TOP-LEVEL-SEARCH procedure recursively for the state q', and q' is added to the $visited$ set at line 4 of the algorithm before a second search from the transition t. $\square$

Lemma 4 *Let $t \in \Delta$ be a transition from which the algorithm starts a second search at line 9. If there exists a state $q \in visited(t)$ that has a successor $q' \in Q \setminus visited(t)$, then $q \in path\_stack(t)$.*

Proof: Let q be a state satisfying the assumptions. Because $q \in visited(t)$ holds, the top-level search has entered q. If the search had also backtracked from q, then the algorithm would have reached line 12 with q on top of the top-level search stack. Therefore the algorithm would have started a second search from all transitions having q as its source state, in particular, from a transition $(q, \sigma, q') \in \Delta$. But then $q' \in visited$ would hold at the beginning of the search from t by Lemma 3, which is a contradiction. Therefore, the top-level search has not yet backtracked from q, and $q \in \mathit{path\_stack}(t)$ holds. □

Lemma 5 *Let $(q, \sigma, q') \in \Delta$ be a transition for which the algorithm calls the second search procedure at line 9 or 24 such that $c \geq n$ holds at line 18 for some $0 \leq n \leq |\mathcal{F}|$. Then, $\mathit{count}[q'] \geq n$ holds when this call returns.*

Proof: If $\mathit{count}[q'] < n$ holds at line 18, $\mathit{count}[q']$ is updated to the value $c \geq n$ at line 20 of the algorithm, and the result follows from the fact that $\mathit{count}[q']$ never decreases during the execution of the algorithm. □

Lemma 6 *Assume that the algorithm reaches line 20 during a second search started from a transition $t \in \Delta$ such that $c \geq n$ holds for some $1 \leq n \leq |\mathcal{F}|$, and the program variable q' refers to a state $q \in Q$. The second search procedure will be called recursively for each transition having q as its source state such that $c \geq n$ holds at line 18 at the beginning of each call.*

Proof: The result holds trivially if q has no successors. Otherwise, let $t' = (q, \sigma, q')$ be a transition having q as its source state. It is easy to see from line 20 that $\mathit{count}[q] \geq n$ holds at the end of the second search from t.

Assume that $q' \in \mathit{visited}(t)$ holds. Denote by $\mathcal{I}$ the contents of the $\mathcal{I}_{\text{seen}}$ set at the beginning of the first iteration of the loop at line 24 when the program variable q' refers to the state q with $c \geq n$. Because $c \geq n$, it is easy to see that $\{1, 2, \ldots, n\} \subseteq \mathcal{I}$ holds at this point. Because $\mathcal{I}_{\text{seen}} = \mathcal{I}$ holds at the beginning of each iteration of the loop (and therefore, in particular, at the beginning of the recursive call of the second search procedure for the transition t'), line 17 guarantees that $c \geq n$ holds at line 18 in the beginning of this call.

If $q' \notin \mathit{visited}(t)$, the algorithm has not yet started a second search from the transition t' (Lemma 3), and thus $t' > t$ holds. Furthermore, $q \in \mathit{path\_stack}(t)$ holds by Lemma 4, i.e., the top-level search has not backtracked from the state q before the second search from t. Clearly, a second search is started from the transition t' before the top-level search backtracks from q. Because $\mathit{count}[q] \geq n$ holds at the end of the second search from t, it follows from the initialization of the $\mathcal{I}_{\text{seen}}$ set at line 8 that $c \geq n$ will hold at line 18 when the second search is started from the transition t' at line 9. □

In the following results, we shall refer to a maximal strongly connected component $C \subseteq Q \cup \Delta$ that contains a state reachable from the initial state of the automaton. Clearly, the top-level search will in this case explore all states in the component, and thus there exists a state $\hat{q} \in C \cap Q$ that is the first state of C pushed on the top-level search stack. Because the elements of this stack are accessed in "last in, first out" order, it is easy to see that $\hat{q}$ is the last state of C from which the top-level search backtracks.

Our goal is to show that if the component C contains an accepting cycle (i.e., a cycle that fulfills all acceptance conditions in $\mathcal{F}$), then there exists a state $q \in C \cap Q$ in the component (namely, the first state $\hat{q}$ of the component entered in the top-level search) for which $\mathit{count}[q] = |\mathcal{F}|$ holds when the top-level search is about to backtrack from the state q. Clearly, this implies that the algorithm will report that the language accepted by the automaton is not empty (at the latest) at line 10 with $\hat{q}$ on top of *path_stack*. (Because C contains an accepting cycle, C contains at least one transition having $\hat{q}$ as its source state, and thus it is easy to see that the algorithm will in fact reach line 10 with $\hat{q}$ on top of the top-level search stack before the top-level search backtracks from $\hat{q}$.)

In general, we shall be interested in finding states $q \in C \cap Q$ for which $count[q] \geq n$ holds for some $1 \leq n \leq |\mathcal{F}|$ when the top-level search is about to backtrack from the state q at line 13. Given a nonempty path in the automaton, the last state (in the path) for which $count[q] \geq n$ holds at the end of a second search started in the first state of the path is guaranteed to have this property unless the path ends in the state q.

Lemma 7 *Let $q_0, q_1, \ldots, q_k \in Q$ be the list of consecutive states in a nonempty path from the state q_0 to the state q_k in the automaton for some $1 \leq k < \omega$. Assume that there exists a maximal index $0 \leq \ell \leq k$ for which $count[q_\ell] \geq n$ holds for some $1 \leq n \leq |\mathcal{F}|$ at the end of a second search from a transition $t \in \Delta$ having q_0 as its source state (line 10). If $\ell < k$ holds, then q_ℓ is a state for which $count[q_\ell] \geq n$ already holds when the top-level search backtracks from the state q_ℓ. More precisely, in this case $count[q_\ell] \geq n$ actually holds before the algorithm starts a second search from any transition from q_ℓ to $q_{\ell+1}$.*

Proof: Because $n \geq 1$, there exists a transition $t' \in \Delta$, $t' \leq t$, such that the second search started from t' reached line 20 of the algorithm such that the program variable q' referred for the first time to the state q_ℓ with $c \geq n$. Clearly, $q_\ell \in visited(t')$ holds, and the top-level search had entered q_ℓ at some point. By Lemma 6, the algorithm calls the second search procedure recursively for a transition from q_ℓ to $q_{\ell+1}$ such that $c \geq n$ holds at line 18 at the beginning of this call. If $q_{\ell+1} \in visited(t')$ holds, such a call occurs during the second search started from the transition t'. But then $count[q_{\ell+1}] \geq n$ would hold at the end of the call by Lemma 5, and therefore also at the end of the second search from t. This contradicts the maximality of ℓ, however. It follows that $q_{\ell+1} \notin visited(t')$ holds, and by Lemma 4, $q_\ell \in path\_stack(t')$. Thus the top-level search backtracks from the state q_ℓ after the second search from t', and $count[q_\ell] \geq n$ holds when this occurs. The second claim follows from Lemma 3: because $q_{\ell+1} \notin visited(t')$, no second search can have been started at line 9 from a transition from q_ℓ to $q_{\ell+1}$ before updating $count[q_\ell]$ to a value greater than or equal to n (in the second search from t'). □

Using Lemma 7, we can prove the following result, whose corollary then shows that the reachability information (i.e., the knowledge on reachablity via paths fulfilling certain acceptance conditions) stored in the *count* table propagates towards the first state of the maximal strongly connected component C entered in the top-level search.

Lemma 8 *Let C be a maximal strongly connected component of the automaton, and let $\hat{q} \in C \cap Q$ be the first state of the component entered in* *the top-level search. Let $q \in C \cap Q$, $q \neq \hat{q}$, be another state in the component such that $count[q] \geq n$ holds for some $1 \leq n \leq |\mathcal{F}|$ when the top-level search is about to backtrack from the state q at line 13. There exists a state $q' \in C \cap Q$, $q' \neq q$, such that the top-level search backtracks from q' after backtracking from q, and $count[q'] \geq n$ holds when this occurs at line 13.*

Proof: Because q and $\hat{q}$ belong to the same maximal strongly connected component, there exists a nonempty path from q to $\hat{q}$ in the automaton. Clearly, all states in this path are contained in C. Let $q_0, q_1, \ldots, q_k \in C \cap Q$ ($1 \leq k < \omega$, $q_0 = q$, $q_k = \hat{q}$) be the list of consecutive states in this path. Because $count[q_0] \geq n$ holds when the top-level search is about to backtrack from the state q_0 at line 13, there exists a maximal index $0 \leq \ell \leq k$ such that $count[q_\ell] \geq n$ holds at this point. Clearly, $count[q_\ell] \geq n$ holds already at the end of a second search started at line 9 from the last transition $t \in \Delta$ (with source state q_0) which was processed in the loop between lines 5 and 11 (and such a transition exists because C contains a nonempty path from q_0 to q_k).

If $\ell = k$, the result follows immediately (with $q' = \hat{q}$) by the choice of $\hat{q}$. Otherwise, by Lemma 7, $count[q_\ell] \geq n$ holds at the beginning of each second search starting at line 9 from a transition from q_ℓ to $q_{\ell+1}$ (and still when the top-level search is about to backtrack from the state q_ℓ). Let $t' \in C \cap \Delta$ be a transition from q_ℓ to $q_{\ell+1}$. If the top-level search had backtracked from q_ℓ before backtracking from q_0 (or if $q_\ell = q_0$), a second search would have been started from the transition t'. But then, because $count[q_\ell] \geq n$ holds at the beginning of this second search (and because $q_{\ell+1} \in visited(t')$ necessarily holds by Lemma 3 at this point), it follows by Lemma 5 that $count[q_{\ell+1}] \geq n$ would hold when the top-level search backtracks from the state q_0. This, however, contradicts the maximality of ℓ. Therefore, $q_\ell \neq q_0$ holds, and the top-level search backtracks from q_ℓ only after backtracking from q_0 ($= q$). The result now follows with $q' = q_\ell$. □

Corollary 1 *Let C be a maximal strongly connected component of the automaton, and let $\hat{q} \in C \cap Q$ be the first state of the component entered in the top-level search. Let $q \in C \cap Q$ be a state in the component such that $count[q] \geq n$ holds for some $1 \leq n \leq |\mathcal{F}|$ when the top-level search is about to backtrack from q at line 13. Then, $count[\hat{q}] \geq n$ holds when the top-level search is about to backtrack from $\hat{q}$.*

Proof: The result holds trivially if $q = \hat{q}$. Let thus $q \neq \hat{q}$. Because $count[q] \geq n \geq 1$ holds when the top-level search is about to backtrack from the state q at line 13, then, by Lemma 8, there exists a state $q' \in C \cap Q$, $q' \neq q$, from which the top-level search backtracks after backtracking from the state q, and $count[q'] \geq n$ holds when this occurs.

By repeating the argument, we can now construct a sequence of pairwise distinct states $q, q', \ldots \in C \cap Q$ such that for any state q'' in the sequence, the top-level search backtracks from the state q'' only after backtracking from all its predecessors in the sequence, and $count[q''] \geq n$ holds when the top-level search backtracks from q''. Because C is finite and $\hat{q}$ is the last state in C from which the top-level search backtracks, the sequence ends with the state $\hat{q}$, and the result follows. □

On the other hand, when the top-level search is about to backtrack from the first state $\hat{q}$ of the component C entered in the top-level search, the following relationship holds between $count[\hat{q}]$ and the values stored in the $count$ table for other states in the component.

Lemma 9 *Let C be a maximal strongly connected component of the automaton, and let $\hat{q} \in C \cap Q$ be the first state of the component entered in the top-level search. Assume that $count[\hat{q}] = n$ holds for some $0 \leq n \leq |\mathcal{F}|$ when the top-level search backtracks from $\hat{q}$ at line 13. Then, $count[q] \geq n$ holds for all $q \in C \cap Q$ at this point.*

Proof: The result holds trivially if $n = 0$. For the rest of the proof, we assume that $n > 0$ holds. Let $q \in C \cap Q$. Because q and $\hat{q}$ belong to the same maximal strongly connected component, there exists a path from $\hat{q}$ to q in the component. We proceed by induction on the number of transitions in a path from $\hat{q}$ to q.

If the path contains no transitions, then $q = \hat{q}$, and the result holds for the state q trivially.

Assume that the result holds for all states in C that are reachable from $\hat{q}$ via a shortest path of $0 \leq k < \omega$ transitions, and let $q \in C \cap Q$ be a state that is reachable from $\hat{q}$ via a shortest path $(t_i)_{i=0}^{k}$ of $k + 1$ transitions ($t_i = (q_i, \sigma_i, q_{i+1}) \in \Delta$ for all $0 \leq i \leq k$, $q_0 = \hat{q}$, and $q_{k+1} = q$). Clearly, q_k is reachable from $\hat{q}$ via a shortest path having k transitions, and because $\hat{q}, q \in C$, $q_k \in C$ holds also. By the induction hypothesis, $count[q_k] \geq n > 0$ holds when the top-level search backtracks from $\hat{q}$. This implies the existence of a transition $t \in \Delta$ such that $count[q_k]$ was first updated to a value greater than or equal to n during a second search started (at line 9) from the transition t. It follows that the algorithm reached line 20 such that the program variable q' referred to the state q_k with $c \geq n$. Let $t' \in C \cap \Delta$ be a transition from q_k to q. By Lemma 6, the algorithm will call the second search procedure recursively for the transition t' such that $c \geq n$ holds at line 18 at the beginning of this call. Furthermore, by the choice of $\hat{q}$, this call occurs before the top-level search backtracks from the state $\hat{q}$. By Lemma 5, it follows that $count[q] \geq n$ holds when the top-level search backtracks from $\hat{q}$.

The result now follows by induction for all states $q \in C \cap Q$. □

A maximal strongly connected component C that contains an accepting cycle includes a transition from each set of accepting transitions. We wish to use Corollary 1 to show that the existence of these transitions forces the condition at line 10 to hold in the first state of C entered in the top-level search. However, Corollary 1 does not directly refer to accepting transitions. We therefore need the following technical result, which establishes a connection between the existence of a transition (in C) which fulfills the n^{th} acceptance condition ($1 \leq n \leq |\mathcal{F}|$) and a state $q \in C \cap Q$ for which $count[q] \geq n$ holds when the top-level search is about to backtrack from the state.

Lemma 10 *Let C be a maximal strongly connected component of the automaton, and let $\hat{q} \in C \cap Q$ be the first state of the component entered in the top-level search. Let $t_F = (q_F, \sigma_F, q'_F) \in C \cap \Delta \cap F_{n+1}$ be a transition that fulfills the acceptance condition F_{n+1} for some $0 \leq n < |\mathcal{F}|$. Assume that,* *before the top-level search backtracks from the state $\hat{q}$, the* SECOND-SEARCH *procedure is called at line 9 or 24 of the algorithm with t_F as a parameter such that $c \geq n$ holds at the beginning of this call at line 18. There exists a state $q \in C \cap Q$ such that $count[q] \geq n+1$ holds when the top-level search is about to backtrack from the state q at line 13.*

Proof: Clearly, the call to the SECOND-SEARCH procedure with t_F as a parameter occurs during a second search started at line 9 from a transition $t \in \Delta$ such that there exists a (possibly empty) path from the source state of t to the state q_F in the automaton. Because $c \geq n$ holds at line 18 and $t_F \in F_{n+1}$, it actually follows that $c \geq n+1$ holds at line 18, and thus $count[q'_F] \geq n+1$ holds at the end of the second search from t by Lemma 5.

Because $q'_F \in C$ holds, there exists a nonempty path from q'_F to $\hat{q}$ in the automaton, and this path visits only states in C. Thus there exists a nonempty path from the source state q_0 of t to $\hat{q}$ in the automaton. Let $q_0, q_1, \ldots, q_k \in Q$ ($1 \leq k < \omega$, $q_k = \hat{q}$, $q_i = q'_F$ for some $1 \leq i \leq k$, and $q_j \in C$ for all $i \leq j \leq k$) be the list of consecutive states in this path. Because $count[q'_F] \geq n+1$ holds at the end of the second search from t, there exists a maximal index $i \leq \ell \leq k$ such that $count[q_\ell] \geq n+1$ holds at this point.

If $\ell = k$, then the result follows (with $q = \hat{q}$) by the choice of $\hat{q}$. Otherwise $count[q_\ell] \geq n+1$ holds when the algorithm is about to backtrack from the state q_ℓ by Lemma 7, and the result holds with $q = q_\ell$ because $q_\ell \in C$. $\square$

We can now prove the completeness of the algorithm.

Theorem 2 *Let $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ (where $\mathcal{F} = \{F_1, F_2, \ldots, F_n\} \subseteq 2^\Delta$ for some $1 \leq n < \omega$) be the nondeterministic generalized Büchi automaton given as input for the algorithm. If $\mathcal{A}$ contains an accepting cycle, the algorithm reports that the language of $\mathcal{A}$ is not empty.*

Proof: Because the automaton contains an accepting cycle, there exists a maximal strongly connected component $C \subseteq Q \cup \Delta$ that contains a transition from each F_i for all $1 \leq i \leq |\mathcal{F}|$. Let $\hat{q}$ be the first state of C entered in the top-level search.

Assume that $count[\hat{q}] = m < n$ holds when the top-level search is about to backtrack from the state $\hat{q}$. By Lemma 9, it follows that $count[q] \geq m$ holds for all $q \in C \cap Q$ at this point.

Because C contains an accepting cycle, there exists a transition $t_{m+1} \in C \cap \Delta \cap F_{m+1}$ which fulfills the acceptance condition F_{m+1}. We claim that the algorithm calls the second search procedure at line 9 or 24 for the transition t_{m+1} such that $c \geq m$ holds at line 18 at the beginning of this call. This is clear if $m = 0$, because a second search is started from every transition in C before the top-level search backtracks from the state $\hat{q}$. If $m > 0$, then, because $count[q] \geq m$ holds for all $q \in C \cap Q$ before the top-level search backtracks from the state $\hat{q}$ (but $count[q] = 0$ holds for all $q \in C \cap Q$ initially), the algorithm reached line 20 such that the program variable q' referred to the source state of t_{m+1} with $c \geq m$ before the top-level search backtracks from $\hat{q}$. In this case the second search procedure will be called for the transition t_{m+1} (before the top-level search backtracks from $\hat{q}$) such that $c \geq m$ holds at line 18 at the beginning of this call by Lemma 6.

Because $c \geq m$ holds at line 18 when the second search procedure is called with the transition $t_{m+1} \in F_{m+1}$ as a parameter, it now follows by Lemma 10 that there exists a state $q_{m+1} \in C \cap Q$ such that $count[q_{m+1}] \geq m + 1$ holds before the top-level search backtracks from q_{m+1}. But then, by Corollary 1, $count[\hat{q}] \geq m + 1$ holds when the top-level search backtracks from the state $\hat{q}$. This contradicts the assumption that $count[\hat{q}] = m < n$ holds at this point.

It follows that $count[\hat{q}] \geq n$ necessarily holds when the top-level search backtracks from $\hat{q}$ (and $count[\hat{q}] = n$, because $c \leq n$ clearly holds always at line 20, the only location where the value of $count[\hat{q}]$ may change). Therefore, because $\hat{q} \in C$ is the source state of at least one transition, the algorithm will report that the language of the automaton is not empty at line 10 before the top-level search backtracks from $\hat{q}$ (at the latest, after a second search from the last transition examined in the loop between lines 5 and 11 with $\hat{q}$ on top of *path_stack*). □

4 DISCUSSION

The proposed algorithm was obtained by modifying the condition heuristic presented in the article [9]. This heuristic is based on keeping track of all acceptance conditions "seen" in a path to the state currently referred to by the program variable q in a second search. This is not strictly necessary for completeness, however: it is actually sufficient to keep track of only the maximal index of a condition that has been "seen" along with all its predecessors in the acceptance condition ordering. Using this observation, the algorithm could be simplified by replacing the $\mathcal{I}_{\text{seen}}$ set with a counter representing this maximal index and using *condition_stack* to record the history of changes to this counter: when the value of the counter changes, the old value of the counter could then be pushed on or popped off the stack in a single operation instead of iterating through indices of acceptance conditions. However, because this change makes second searches more dependent on the condition ordering, the simplified algorithm would in some cases have to explore more states and transitions than the algorithm in Figure 1 before detecting an accepting cycle (for example, in an automaton that consists of a ring of transitions in which the conditions occur in "reverse" order).

We stress that the completeness of the new algorithm nevertheless depends critically on the ability to take acceptance conditions "seen" in second searches into account when updating the *count* table during the searches. For example, it is not possible to obtain an even simpler algorithm by modifying the basic generalized search algorithm from [9] (i.e., nested search without condition heuristic) to use counters: because the basic algorithm ignores all acceptance conditions encountered in a second search that were not "seen" already at the beginning of the search (or in the transition from which the search was started), it is easy to find examples in which the ordering chosen for the conditions interferes with the updating of counters in a way that makes the algorithm fail to detect the existence of an accepting cycle in an automaton even if such a cycle exists.

In comparison with the condition heuristic from [9], advancing second

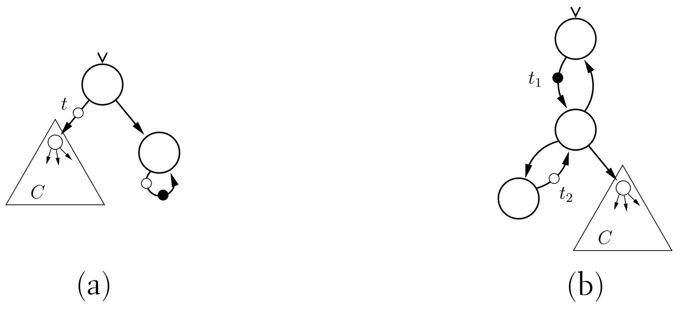

Figure 2: Automata for showing neither of the original and new algorithms to perform universally better than the other. In the figures, C can be an arbitrarily large (finite) component of the automaton. The transitions marked with • fulfill an acceptance condition that precedes the condition containing the transitions marked with ∘ in the ordering for the conditions.

searches in the automaton by incrementing counters associated with states of the automaton may sometimes reduce the total number of states and transitions explored in second searches. For example, if the component C of the automaton shown in Figure 2 (a) contains no accepting transitions, the new algorithm avoids starting a second search from the transition t, whereas the original algorithm proposed in the article [9] will perform a redundant second search in case the top-level search explores the left part of the automaton before entering the state with the self-loop. On the other hand, using a stricter condition for advancing second searches may also have negative effects on performance as illustrated in Figure 2 (b). If the top-level search explores the source state of the transition t_2 before entering the component C, the original algorithm can (with the condition heuristic) detect the existence of an accepting cycle immediately after a second search from t_2 without ever entering the component C. Because of the ordering chosen for the acceptance conditions, the new algorithm will not detect this cycle until after a second search from the transition t_1; however, at this point also all states in C have been explored at least once (in the top-level search). It is apparent from this example that, similarly to automata degeneralization constructions needed for applying the classic nested depth-first search algorithm to generalized automata, the performance of the new algorithm depends on the ordering chosen for the acceptance conditions.

Because the new algorithm differs from the basic algorithm with the condition heuristic only in the search strategy used for second searches, almost all notes from the article [9] concerning the construction of accepting cycles and the compatibility of the algorithm with enhancements suggested in the literature apply directly to the new algorithm. Although the existence of an accepting cycle in an automaton can be reported immediately when incrementing $count[q]$ to $|\mathcal{F}|$ at line 20 for a state q currently in the top-level search stack (using an additional bit of memory stored with each state in the *count* table to record its presence in the stack [6]), constructing an actual cycle that fulfills all acceptance conditions is not possible in general without extra effort [2, 9]. The algorithm can be made compatible with partial order reduction [7] identically to the classic nested depth-first search algorithm [6], and with probabilistic state exploration techniques [5, 8, 10] or state space caching [4] in a more limited way [9].[1] The algorithm can also be made to support weights to speed up the detection of accepting cycles as suggested in the article [2]; however, in comparison with the basic generalized algorithm, the more restrictive condition used for advancing second searches may reduce opportunities for applying this optimization successfully. The only extension from [9] not supported by the new algorithm is the detection of cycles fulfilling a given number of acceptance conditions since the new algorithm does not treat the conditions symmetrically.

ACKNOWLEDGMENTS

This research was supported financially by Academy of Finland (project number 211025) and Helsinki Graduate School in Computer Science and Engineering (HeCSE). The author is also grateful to Keijo Heljanko for insightful comments and suggestions for improving the presentation.

REFERENCES

[1] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.

[2] J.-M. Couvreur, A. Duret-Lutz, and D. Poitrenaud. On-the-fly emptiness checks for generalized Büchi automata. In P. Godefroid, editor, *Proceedings of the 12th International* SPIN *Workshop on Model Checking of Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 169–184. Springer-Verlag, 2005.

[3] E. A. Emerson and A. P. Sistla. Deciding full branching time logic. *Information and Control*, 61(3):175–201, 1984.

[4] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. *Formal Methods in System Design*, 7(3):227–241, 1995.

[5] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, 1991.

[6] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors, *Proceedings of the Second Workshop on the* SPIN *Verification System*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.

[7] D. A. Peled. Combining partial order reductions with on-the-fly model checking. *Formal Methods in System Design*, 8(1):39–64, 1996.

[1]With probabilistic state exploration techniques, a separate hash table is needed to keep track of the exact values of *count* for states currently in the top-level search stack to ensure the soundness of the algorithm [9]. To preserve the number of state space traverals required in the worst case, changes to *count*[*q*] in this table should be made to the imperfectly hashed lookup table only if the previous value stored in the imperfectly hashed table thus increases.

[8] U. Stern and D. Dill. A new scheme for memory-efficient probabilistic verification. In R. Gotzhein and J. Bredereke, editors, *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV 1996)*, pages 333–348. Kluwer, 1996.

[9] H. Tauriainen. Nested emptiness search for generalized Büchi automata. *Fundamenta Informaticae*, 2005. In press.

[10] P. Wolper and D. Leroy. Reliable hashing without collision detection. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV 1993)*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 1993.

ISBN 951-22-7874-X
ISSN 1457-7615