Helsinki University of Technology Laboratory for Theoretical Computer Science

Research Reports 79

Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 79

Espoo 2003 HUT-TCS-A79

NESTED EMPTINESS SEARCH FOR GENERALIZED BÜCHI AUTOMATA

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 79
Teknillisen korkeakoulun tietojenkäsittelyteorian laboratorion tutkimusraportti 79
Espoo 2003 HUT-TCS-A79

NESTED EMPTINESS SEARCH FOR GENERALIZED BÜCHI AUTOMATA

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
FIN-02015 HUT
Tel. +358-0-451 1
Fax. +358-0-451 3369
E-mail: lab@tcs.hut.fi

ISBN 951-22-6667-9
ISSN 1457-7615

2003

Abstract: We generalize the classic explicit state emptiness checking algorithm for Büchi word automata (the "nested depth-first search") into Büchi automata with multiple acceptance conditions. Bypassing an explicit acceptance condition reduction improves the algorithm's worst case memory requirements. The generalized algorithm handles multiple unconditional and weak fairness constraints directly and is compatible with well-known probabilistic explicit state model checking techniques.

Keywords: Model checking, Büchi automata emptiness checking, nested depth-first search

CONTENTS

1 INTRODUCTION

The automata-theoretic verification framework [23] provides a powerful approach to the correctness analysis of finite-state reactive and concurrent systems, by, for example, capturing the expressive power of many temporal logics used for the specification of correctness requirements. In this framework, both the system and its correctness requirements are expressed as finite automata on infinite objects, such as *Büchi word automata*: finite automata operating on infinite strings, the acceptance of which is determined by a set of states that the automaton should visit infinitely often while processing a string. In some applications this set of accepting states is generalized into a *family* of accepting state sets such that a string is accepted only if the basic acceptance condition holds separately for each individual set in the family. For example, multiple acceptance conditions may arise in temporal logic model checking, where the correctness requirements are translated automatically into Büchi automata from some other formalism such as a linear time temporal logic formula [10].

Interpreted itself as an automaton, the (reachable) state space of the formal model of the system under investigation encodes a set of infinite strings corresponding to the computation paths that exist in the system. The correctness specification is represented as an automaton that recognizes a collection of undesirable computation paths violating the specification. Checking whether the system meets the correctness specification is done by taking the synchronous product of all concurrent system components with the specification automaton (see, for example, [5]) and testing whether the set of strings recognized by the resulting automaton, which accepts a string if and only if the system violates the specification, is empty.

The nonemptiness of a generalized Büchi automaton is equivalent to the existence of a cycle of states (reachable from an initial state of the automaton) that intersects all sets of accepting states. Although this can be decided with the help of basic graph algorithms, such as Tarjan's algorithm [22], the unavoidable state explosion problem has forced the development of more memory-efficient emptiness checking algorithms. In the special case of an automaton with only a single set of accepting states, a well-known alternative is to use the *nested depth-first search* algorithm [5] or one of its variants [11, 17, 6, 2], all of which implement the emptiness check as two interleaved depth-first traversals of a product structure constructed on-the-fly from the system components and the specification automaton. Multiple acceptance conditions are usually handled by transforming the specification automaton into an equivalent automaton with only a single acceptance condition [3, 5] before invoking the nested depth-first search algorithm. However, an explicit automaton transformation results in a linear worst case blow-up (in the number of acceptance conditions) in the size of the automaton. When aiming for extreme memory-efficiency, even this linear blow-up can have a nonnegligible impact on the memory requirements when composing the specification automaton with the concurrent system components (the product of which can be exponential in the number of components), for example, in a conventional hash table based implementation in which each product state descriptor consumes the same fixed amount of memory.

This paper presents a generalization of the basic nested depth-first search emptiness checking algorithm into Büchi automata with multiple acceptance conditions. By avoiding the blow-up caused by acceptance condition reduction, the generalized version of the algorithm allows a slight improvement in the algorithm's minimum worst case memory requirements. The generalized algorithm is compatible with well-known improvements to explicit state model checking such as probabilistic model checking techniques [15, 25, 21]. The generalized algorithm can also readily handle unconditional and weak fairness assumptions (see, for example, [9]) attached directly to the concurrent system components [1] without increasing the complexity of the specification automaton.

2 PRELIMINARIES

This section reviews the definition of generalized Büchi automata and the classic automata conversion used for handling generalized acceptance conditions, together with an example to illustrate why avoiding the conversion may be desirable in practice.

2.1 Definitions

A (nondeterministic) generalized Büchi automaton is a tuple $(\Sigma, Q, \Delta, q_I, \mathcal{F})$, where Σ is a finite nonempty set called the *alphabet*, Q is the finite set of *states*, $\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*, $q_I \in Q$ is the *initial state*, and $\mathcal{F} \subseteq 2^Q$ is the set of (generalized) *acceptance conditions*.

Let $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ be a generalized Büchi automaton. If $(q, \sigma, q') \in \Delta$ holds for some $\sigma \in \Sigma$, we call q' an *immediate successor* of q (denoted by $q \rightarrow q'$). A *path* in the automaton is a sequence of states $x = (q_i)_{i=1}^{n}$ ($n \in \mathbb{N} \cup \{\omega\}$) such that, for all $i \geq 1$ (and $i < n$ if $n < \omega$), q_{i+1} is an immediate successor of q_i. If $n \geq 2$, the path is *nontrivial*. If $x = (q_i)_{i=1}^{n}$ is a finite path in $\mathcal{A}$, we say that q_n is *reachable* from q_1 (via x) in $\mathcal{A}$ (denoted by $q_1 \rightarrow^* q_n$). When x is nontrivial, we occasionally use the notation $q_1 \rightarrow^+ q_n$ for reachability to emphasize this property.

Let x be a path in $\mathcal{A}$. If $n < \omega$, x *fulfills* the acceptance condition $F \in \mathcal{F}$ iff $q_i \in F$ holds for some $1 \leq i \leq n$. If $n = \omega$, define the set $\inf(x) = \{q \in Q \mid \forall i \geq 1 : \exists j > i : q_j = q\}$ of states occurring infinitely many times in x. In this case we say that the path x fulfills the acceptance condition $F \in \mathcal{F}$ iff $\inf(x) \cap F \neq \emptyset$.

An (infinite) *word* over the alphabet Σ is a sequence of symbols $w = (\sigma_i)_{i=1}^{\omega}$, where $\sigma_i \in \Sigma$ holds for all $i \geq 1$. A *run of $\mathcal{A}$ over w* is an infinite path $r = (q_i)_{i=1}^{\omega}$ in the automaton such that $q_1 = q_I$ and $(q_i, \sigma_i, q_{i+1}) \in \Delta$ holds for all $i \geq 1$. We say that r is *accepting* iff r fulfills all acceptance conditions $F \in \mathcal{F}$. If $\mathcal{A}$ has an accepting run over an infinite word w, we say that $\mathcal{A}$ *accepts* w. The set of infinite words accepted by the automaton $\mathcal{A}$ is called the *language recognized by $\mathcal{A}$*. $\mathcal{A}$ is *empty* iff the language recognized by it is empty.

2.2 Acceptance Condition Reduction

Any generalized Büchi automaton can be transformed into an automaton that recognizes the same language as the original automaton but uses only a single acceptance condition. For example, the well-known construction of [5] transforms an automaton $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ with state set $Q = \{q_1, q_2, \ldots, q_n\}$ (where $q_I = q_1$) and a nonempty[1] set of generalized acceptance conditions $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$ into another automaton $\mathcal{A}' = (\Sigma, Q', \Delta', q_I', \mathcal{F}')$, where $Q' = \{q_{(i,j)}\}_{i=1,j=1}^{i=n,j=m}$, $\Delta' = \bigcup_{k=1}^{m} \big(\{(q_{(i,k)}, \sigma, q_{(j,k)}) \mid (q_i, \sigma, q_j) \in \Delta, q_i \notin F_k\} \cup \{(q_{(i,k)}, \sigma, q_{(j,1+(k \bmod |\mathcal{F}|))}) \mid (q_i, \sigma, q_j) \in \Delta, q_i \in F_k\}\big)$, $q_I' = q_{(1,k)}$, and $\mathcal{F}' = \{\{q_{(i,k)} \mid q_i \in F_k\}\}$, for some fixed $1 \leq k \leq m$.

When applied to an automaton $\mathcal{A}$ with n states and $m \geq 1$ generalized acceptance conditions, the transformation yields an automaton $\mathcal{A}'$ with nm states and one acceptance condition such that $\mathcal{A}'$ and $\mathcal{A}$ recognize the same language. The construction gives an $O(nm)$ worst case lower bound for the number of states in an automaton obtained by acceptance condition reduction. As shown in the following example, this lower bound is optimal, i.e., the linear blow-up in the number of states in the automaton cannot be avoided in the general case. (We explicate this well-known result in automata theory only for illustration.)

Example. *Let Σ_n denote a finite alphabet with n distinct symbols $\sigma_1, \ldots, \sigma_n$ together with an extra symbol $\#$ different from each σ_i. For each $n \geq 2$, define a set of infinite strings $\mathcal{L}_n$ over Σ_n characterized by the expression*

$$\mathcal{L}_n = \Big(((\cup_{i=2}^{n} \sigma_i)\#^n)^* \sigma_1 \#^n ((\cup_{\substack{i=1 \\ i \neq 2}}^{n} \sigma_i)\#^n)^* \sigma_2 \#^n ((\cup_{\substack{i=1 \\ i \neq 3}}^{n} \sigma_i)\#^n)^* \sigma_3 \#^n \cdots \cdots ((\cup_{i=1}^{n-1} \sigma_i)\#^n)^* \sigma_n \#^n \Big)^{\omega},$$

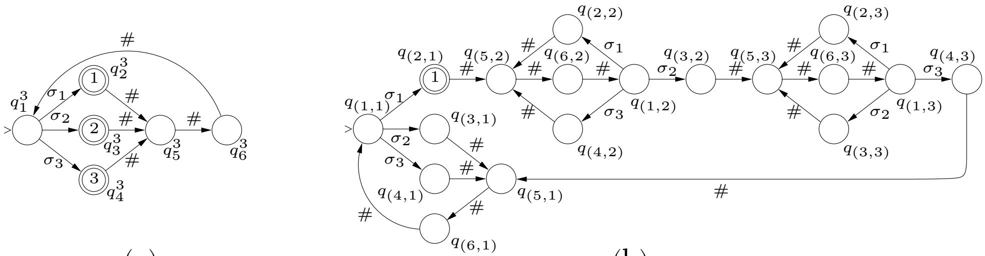
i.e., the set of infinite strings built from $(n+1)$-symbol "blocks" over Σ, each of which consists of one of the symbols σ_i ($1 \leq i \leq n$) followed by exactly n $\#$'s, with the additional constraint that each σ_i has to occur in the resulting string infinitely many times.

The set of strings $\mathcal{L}_n$ can be recognized by the $2n$-state generalized Büchi automaton $\mathcal{A}_n = (\Sigma_n, Q_n, \Delta_n, q_I^n, \mathcal{F}_n)$, where $Q_n = \{q_1^n, q_2^n, q_3^n, \ldots, q_{2n}^n\}$, $q_I^n = q_1^n$, the transition relation $\Delta_n = \big(\bigcup_{i=1}^{n} \{(q_1^n, \sigma_i, q_{i+1}^n), (q_{i+1}^n, \#, q_{n+2}^n)\}\big) \cup \big(\bigcup_{i=n+2}^{2n} \{(q_i^n, \#, q_{(i \bmod 2n)+1}^n)\}\big)$, and $\mathcal{F}_n = \bigcup_{i=2}^{n+1} \{\{q_i^n\}\}$. As a concrete example, Fig. 2.2 (a) depicts the automaton $\mathcal{A}_3$.

By the above construction, there exists a Büchi automaton with $2nm$ (with $m = |\mathcal{F}_n| = n$) states and a single acceptance condition recognizing the language $\mathcal{L}_n$. Figure 2.2 (b) shows the result when the conversion is applied to the automaton $\mathcal{A}_3$. Although the result could be simplified, there is a lower bound for the size of the automaton: we argue that any automaton (with a single acceptance condition) that recognizes the same language always has more than nm states.

Let $\mathcal{A} = (\Sigma_n, Q, \Delta, q_I, \{F\})$ be an automaton that recognizes $\mathcal{L}_n$ using only one acceptance condition F, and let $w \in \mathcal{L}_n$. Thus, $\mathcal{A}$ accepts w, and there exists an accepting run r of $\mathcal{A}$ on w and a state $q \in F \cap \inf(r)$. Let u

[1] If $\mathcal{F} = \emptyset$, $\mathcal{A}'$ can be defined simply as $\mathcal{A}' = (\Sigma, Q, \Delta, q_I, \{Q\})$.



(a) (b)

Fig. 1: (a) Generalized Büchi automaton $\mathcal{A}_3$. (b) Automaton obtained from $\mathcal{A}_3$ by acceptance condition reduction. States associated with different acceptance conditions are indicated by numbers in the double circles

be a finite prefix of w such that $\mathcal{A}$ reaches the state q after reading u in the run r. Because $q \in \inf(r)$, q is reachable from itself in the automaton via a nontrivial path.

Consider any shortest nontrivial path from q to itself in the automaton, and let v be the string composed of the successive transition labels occurring on the path. It is clear that $\mathcal{A}$ accepts the string uv^ω. Because $\mathcal{A}$ recognizes $\mathcal{L}_n$, it follows that $uv^\omega \in \mathcal{L}_n$. Hence each σ_i must occur at least once in v. In the simplest case, each σ_i occurs in v exactly once, and v is of the form $\#^k\sigma_{\rho(1)}\#^n\sigma_{\rho(2)}\#^n \cdots \sigma_{\rho(n)}\#^{n-k}$ for some $0 \leq k \leq n$ and some permutation ρ of $\{1, \ldots, n\}$. Clearly, v has $n^2 + n > n^2 = nm$ symbols. Because v was formed from the transition labels on a shortest *path from q to itself, all states occurring on the path between its endpoints are distinct. It follows that $\mathcal{A}$ has $\Omega(nm)$ states as argued.* ■

In practice, the acceptance condition reduction is traditionally done either explicitly using methods similar to the construction presented in [5], or implicitly with a counter while constructing the on-the-fly synchronous product of the specification automaton with the system components. An explicit reduction is usually further followed by simplification of the nongeneralized automaton, for which several techniques have been proposed in the literature (see, for example, [7, 8, 14]). However, as shown by the above example, the linear blow-up caused by the reduction cannot be always countered by simplification. Thus, although the major cause of the state explosion admittedly lies in the product of the system components, methods for bypassing the acceptance condition reduction may still be of practical relevance for minimizing the total number of product states that need be explored (and thus stored in memory) in the worst case: after all, the nested depth-first search algorithm is always forced to perform an exhaustive search in the product space whenever the product is empty (i.e., when the system satisfies the correctness specification).

3 EMPTINESS CHECKING ALGORITHM

This section presents a new variant of the nested depth-first search algorithm. This variant is directly applicable to Büchi automata with multiple acceptance conditions.

3.1 Outline

The generalized algorithm is shown in Fig. 2. For simplicity, we assume that $\mathcal{F} \neq \emptyset$; otherwise the emptiness check reduces to the problem of finding a reachable cycle in the automaton, in which case a nested search is not needed.

As usual, the algorithm scans the (product) automaton using a depth-first search that drives another interleaved search, which is started from the states in the automaton in depth-first search post-order. The algorithm uses a depth-first search stack *path*, and the *processed* set stores the states already visited during the main depth-first search. Additionally, each state of the automaton has a *label*, which stores (partial) information on the acceptance conditions fulfilled on some nontrivial path to that state in the automaton. The nested search procedure propagates these labels forward in the automaton starting from a given state (and is not necessarily bound to depth-first search mode). Unlike the classic algorithm, the nested search of the generalized algorithm is not allowed to enter states not yet visited in the main search.

The algorithm also uses a function $\mathsf{hash}\colon Q \to H$ that maps each state $q \in Q$ to its *hash value* $\mathsf{hash}(q)$ chosen from a set of hash values H. If the function is one-to-one, we say that the hash function is *perfect*. Lines 12–14, 16 and 27–28 of the algorithm make the effect of hashing explicit (in a practical implementation, the hashing is implicit in the tests for set or hash table membership, and thus each of these loops reduces to a single hash table operation).

The states in the *path* stack are always hashed perfectly such that any changes made to the label of a state $q \in path$ will not carry over to the label of any state $q' \in path \setminus \{q\}$. This is essential for the soundness of the algorithm, and can be implemented in practice, for example, by using a separate perfect hash table for the labels of the states currently in the *path* stack. The purpose of lines 12–14 is to ensure a simple upper bound for the running time of the algorithm by updating any changes made into the label of the state from which the algorithm is about to backtrack back into the (imperfect) hash table in an overapproximative way.

3.2 Complexity

We consider only the memory used for storing visited states and their labels, since this information usually dominates the memory requirements of an explicit state exploration algorithm. In a basic hash table based implementation, where each product state descriptor is stored into the table in its entirety, the label of each state stored in the hash table consumes $|\mathcal{F}|$ bits of memory. A straightforward implementation of the check for *path* stack membership requires one additional bit of memory per state. If the states are inserted into the hash table only as they are first entered during the main search, the presence of a state in the *processed* set can be inferred from its presence in the hash table. The algorithm will thus require (at a minimum) $n(s + |\mathcal{F}| + 1)$ bits of memory for the data in the hash table, where n is the number of states in the automaton and s is the number of bits in each state descriptor. This

Input: A Büchi automaton $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ with $\mathcal{F} \neq \emptyset$.
Output: "TRUE" (if hash is a perfect hash function, if and) only if $\mathcal{A}$ has an accepting run and "FALSE" otherwise.

Initialize: $label := [q_1 \mapsto \emptyset, \ldots, q_{|Q|} \mapsto \emptyset]$; $path := \emptyset$; $processed := \emptyset$;

```
emptinessSearch(($\Sigma, Q, \Delta, q_I, \mathcal{F}$): Büchi automaton)
  begin
    $path$.push($q_I$);
    while ($path \neq \emptyset$) do begin
      $q$ := $path$.top();
      while ($\exists q' \in Q \setminus (path \cup processed) : q \rightarrow q'$) do begin
        $path$.push($q'$); $label[q']$ := $\emptyset$; $q$ := $q'$;
      end;
      if ($label[q] \neq \emptyset$) or ($\exists F \in \mathcal{F} : q \in F$) then begin
        propagate(($\Sigma, Q, \Delta, q_I, \mathcal{F}$), $\{q\}$, $label[q] \cup \{F \in \mathcal{F} \mid q \in F\}$);
        if ($label[q] = \mathcal{F}$) then exit "TRUE";
        $l$ := $\bigcup_{q'' \in \{q' \in Q \mid \text{hash}(q') = \text{hash}(q)\}} label[q'']$;
        for all $q' \in \{q\} \cup \{q'' \in Q \setminus path \mid \text{hash}(q'') = \text{hash}(q)\}$ do
          $label[q']$ := $l$;
      end;
      $processed$ := $processed \cup \{q' \in Q \mid \text{hash}(q') = \text{hash}(q)\}$;
      $path$.pop();
    end;
    exit "FALSE";
  end;

propagate(($\Sigma, Q, \Delta, q_I, \mathcal{F}$): Büchi automaton; $states \in 2^Q$;
          $labels\_to\_propagate \in 2^{\mathcal{F}}$)
  repeat
    remove any state $q$ from $states$;
    while ($\exists q' \in path \cup processed$ :
          $q \rightarrow q'$, $labels\_to\_propagate \not\subseteq label[q']$) do begin
      $states$ := $states \cup \{q'\}$;
      $label[q']$ := $label[q'] \cup labels\_to\_propagate$;
      for all $q'' \in \{\hat{q} \in Q \setminus path \mid \text{hash}(\hat{q}) = \text{hash}(q')\}$ do
        $label[q'']$ := $label[q''] \cup labels\_to\_propagate$;
    end
  until ($states = \emptyset$);
```

Fig. 2: Nested emptiness search algorithm for generalized Büchi automata

lower bound is less than or equal to the $|\mathcal{F}|n(s+2)$ bits of memory required by an efficient implementation of the basic nested depth-first search algorithm [11] for all $|\mathcal{F}| \geq 1$, when the worst case blow-up in the size of the automaton caused by the reduction in the number of acceptance conditions is taken into account. For example, with 40-bit state descriptors and three generalized acceptance conditions, there is approximately a 65% reduction in the minimum worst case memory requirements for the hash table.

Although similar reductions are obviously not to be expected with a more sophisticated hash table implementation based on various shared storage techniques [13, 24, 18], also these implementations may still benefit from the generalized algorithm because of the reduction in the total worst case number of states.

Clearly, the main depth-first search visits each state of the automaton at most once, and thus the nested search is started from each state at most once. Entering a state during the nested search always results in adding at least one new acceptance condition to the label of the state. It follows that each state of the automaton can be entered at most $|\mathcal{F}|$ times[2] over all nested searches, and thus the algorithm has $O(n|\mathcal{F}|)$ running time complexity in the number of visited states. Therefore, the generalized algorithm shares its worst case running time complexity with the basic algorithm (taking the worst case effects of acceptance condition reduction into account), assuming that all hash table and set manipulation operations can be implemented in constant time. (For the set manipulation operations, this constant depends on the number of acceptance conditions $|\mathcal{F}|$. However, its effect on the running time can be reduced by implementing the operations as bit vector primitives whenever possible, such as when $|\mathcal{F}|$ does not exceed the word length of the underlying implementation architecture.)

3.3 Correctness

The basic correctness proof of the classic nested depth-first search algorithm (see, for example, [4]) does not directly generalize into the current algorithm, mainly due to the restriction concerning the set of states that may be entered during a nested search. We thus present a full correctness proof of the generalized algorithm here.

Notation. Let $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ be a Büchi automaton, and let $q \in Q$ be a state in the automaton. We use the shorthand $\texttt{propagate}(q)$ to denote the nested depth-first search rooted at the state q (i.e., the call at line 10 of the algorithm). Additionally, let $\mathit{processed}_q$ and path_q denote the contents of the $\mathit{processed}$ set and the path stack, respectively, at line 9 with $q \in Q$ on top of the path stack.

Lemma 1. *Let $q \in Q$ be the state on top of the path stack at line 9 of the algorithm. Then, $q' \rightarrow^* q$ holds for all states $q' \in \mathit{path}_q$.*

Proof. The algorithm uses path as the depth-first search stack; the states in the stack always form a path to the state currently on top of the stack. □

[2]Resetting a label at line 7 can occur only if the state has not been entered previously during a nested search. Clearly, this reset can occur only once per state.

Lemma 2. *Let $q \in Q$ be a state on top of the path stack at line 9 of the algorithm, and let $F \in \mathcal{F}$ be an acceptance condition. If $F \in label[q]$ already holds at this point, then the automaton contains a nontrivial path from q to itself such that the path fulfills the acceptance condition F.*

Proof. Because $label[q]$ was reset when q was first entered (line 7)[3], q was still in the *path* stack when $label[q]$ was updated. Due to the special treatment of the labels of the states in the *path* stack, there exists a state $q' \neq q$ such that F was added to $label[q]$ during a nested search rooted at q' from which the state q is reachable via a nontrivial path in the automaton.

Since q' was on top of the *path* stack when `propagate(`q'`)` was entered, it follows by Lemma 1 that q' is also reachable from q in the automaton. Thus the automaton contains a nontrivial path from q to itself through the state q'.

The result follows immediately if $q' \in F$. Otherwise, from the fact that the set *labels_to_propagate* remains unchanged during the nested search, we can conclude that $F \in label[q']$ was true at line 9 of the algorithm when the algorithm was about to enter `propagate(`q'`)`. (If this were not the case, F could not have been added to $label[q]$ during the nested search rooted at q', contrary to our assumption.)

By repeating the above reasoning for q', we find a nontrivial path from q' to itself through another state $q'' \neq q'$ (and $q'' \neq q$) such that F was added to $label[q']$ during a nested search rooted at q'' with q' in the *path* stack.

We can thus construct a sequence of unique states in which every two successive states are reachable from each other until we finally find a state q_F that belongs to the acceptance set F. The existence of this state follows from the finiteness of the automaton and from the fact that the algorithm will not include F into the label of any state if $F = \emptyset$.

Therefore, the automaton contains a nontrivial path $q \rightarrow^+ q' \rightarrow^* q'' \rightarrow^* \cdots \rightarrow^* q_F \rightarrow^* \cdots \rightarrow^* q'' \rightarrow^* q' \rightarrow^+ q$ that fulfills the acceptance condition F. □

The proof of Lemma 2 rests on the fact that the labels of the states in the *path* stack change only if these states are actually reached during a nested search. This is the reason why perfect hashing must be applied to the states in the *path* stack. The following theorem establishes the soundness of the algorithm.

Theorem 1. *Let $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ ($\mathcal{F} \neq \emptyset$) be a Büchi automaton given as input for the algorithm. If the algorithm exits with the value "`TRUE`", then the automaton contains a path from the initial state q_I to a state $q \in Q$ reachable from itself via a nontrivial path that fulfills all acceptance conditions $F \in \mathcal{F}$. Therefore, the infinite path $q_I \rightarrow^* q \rightarrow^+ q \rightarrow^+ q \rightarrow^+ \cdots$ is an accepting run of the automaton, and thus the automaton is nonempty.*

Proof. Assume that the algorithm exits with the value "`TRUE`". Clearly, this can occur only if $label[q] = \mathcal{F}$ holds at line 11 of the algorithm (with $q \in Q$ on top of the *path* stack). Since $q_I \in path_q$ is certainly true, there exists a path from q_I to q in the automaton by Lemma 1.

[3]The explicit reset is required for soundness only when using imperfect state hashing. (With perfect hashing, the label is guaranteed to be empty even without the reset operation.)

Let $F \in \mathcal{F}$ be an acceptance condition. If $F \in \mathit{label}[q]$ was true already at line 9 of the algorithm (with q on top of the *path* stack), Lemma 2 proves the existence of a path from q to itself that fulfills the acceptance condition F. Otherwise the algorithm added F to $\mathit{label}[q]$ during a nested search rooted at q itself, which implies that $q \in F$ and that there exists a path from q to itself fulfilling F. By repeating the consideration for all acceptance conditions, we can construct a path from q to itself that fulfills all acceptance conditions. This proves the soundness of the algorithm. $\square$

We now turn to the completeness of the algorithm. From now on we therefore assume that the function `hash` used for state hashing is a perfect hash function. In this case lines 12–14 and 27–28 of the algorithm (required for simulating the effects of imperfect state hashing) become redundant and can be removed without affecting the behavior of the algorithm, and line 16 can be simplified.

We begin by listing several additional basic properties of the algorithm.

Lemma 3. *Let $q \in Q$ be a state in the automaton. If the algorithm proceeds to line 9 with q on top of the* path *stack, then*

(a) *the algorithm will never start a nested search from any state $q' \in \mathit{processed}_q$;*

(b) *if q' is an immediate successor of q, then $q' \in \mathit{path}_q \cup \mathit{processed}_q$; and*

(c) *if there exists a state $q' \in \mathit{path}_q \cup \mathit{processed}_q$, $q \rightarrow^* q'$, with an immediate successor $q'' \notin \mathit{path}_q \cup \mathit{processed}_q$, then $q' \in \mathit{path}_q$.*

Proof.

(a) The main depth-first search visits each state of the automaton at most once. Since $q' \in \mathit{processed}_q$, the search has already backtracked from q', and thus the algorithm cannot (re)start a nested search from q'.

(b) Immediate by the loop termination condition at line 6 of the algorithm.

(c) The case $q' = q$ is impossible by (b). Assume that $q' \in \mathit{processed}_q$, i.e., the algorithm had already backtracked from q'. However, also this is impossible by (b), since the algorithm would have had to proceed to line 9 with q' on top of the *path* stack at some previous point, which would imply that $q'' \in \mathit{path}_{q'} \cup \mathit{processed}_{q'} \subseteq \mathit{path}_q \cup \mathit{processed}_q$, a contradiction. Thus, $q' \in \mathit{path}_q$. $\square$

The completeness proof is based on an invariant of the nested search procedure. The invariant is stated using the following notion of $\mathtt{p}(q)$-*reachability*.

Definition. *Let $q \in Q$ be a state in the Büchi automaton. Assume that the algorithm calls* `propagate`(q) *at line 10. We say that the state $q' \in Q$ is* $\mathtt{p}(q)$-reachable (from q) *if and only if, at the beginning of* `propagate`(q),

- *there exists an integer $n \geq 2$ and states $\{q_1, q_2, \ldots, q_n\} \subseteq \mathit{path}_q \cup \mathit{processed}_q$ such that $q_1 = q$, $q_n = q'$, $q_i \rightarrow q_{i+1}$ for all $1 \leq i < n$, and*

- $\{q_2, q_3, \ldots, q_{n-1}\} \cap path_q = \emptyset$.

(Thus, q' is $\mathsf{p}(q)$*-reachable if it is reachable from q via a nontrivial path contained in $path_q \cup processed_q$ such that the path does not intersect $path_q$ between its endpoints.)* ■

Lemma 4. *Let q be a state in the Büchi automaton, and let $F \in \mathcal{F}$ be an acceptance condition such that $q \in F$ or $F \in label[q]$ holds at line 9 of the algorithm with q on top of the path stack. Then, the algorithm calls* $\mathsf{propagate}(q)$*, and $F \in label[q']$ holds for all* $\mathsf{p}(q)$*-reachable states q' after the call returns.*

Proof. We proceed by induction on the length of paths starting from the state q in the automaton. The case for all immediate successors of q (which are $\mathsf{p}(q)$-reachable by Lemma 3 (b)) is clear from the operation of the nested search.

Assume that the result holds for all $\mathsf{p}(q)$-reachable states reachable from q via a shortest path with exactly n ($n \geq 2$) states. Let q' be a $\mathsf{p}(q)$-reachable state reachable from q via a shortest path $q_1 (= q) \to q_2 \to \cdots \to q_n \to q'$ with $n + 1$ states. Clearly, q_n is $\mathsf{p}(q)$-reachable via a path with n states. There are two cases:

1. The algorithm visits q_n during the nested search rooted at q. Because q' is an immediate successor of q_n and $\mathsf{p}(q)$-reachable, the nested search guarantees that $F \in label[q']$ will hold upon returning from $\mathsf{propagate}(q)$.

2. q_n is not visited during the nested search. Since $F \in label[q_n]$ nevertheless holds after the call to $\mathsf{propagate}(q)$ (by the induction hypothesis), F must have been added to $label[q_n]$ during a previous nested search rooted at some state $\hat{q} \neq q$. Assume that F was not included in $label[q']$ when entering $\mathsf{propagate}(\hat{q})$ and that this search did not visit q' (otherwise there is nothing to show). It follows that $\hat{q} \neq q_n$ and $q' \notin path_{\hat{q}} \cup processed_{\hat{q}}$, and thus $q_n \in path_{\hat{q}}$ by Lemma 3 (c). Therefore, the algorithm starts a nested search from q_n after backtracking from $\hat{q}$. On the other hand, because q' is $\mathsf{p}(q)$-reachable via the path $q_1 (= q) \to q_2 \to \cdots \to q_n \to q'$, it follows that $q_n \in processed_q$. By Lemma 3 (a), it now follows that the nested search from q_n occurs strictly between the calls to $\mathsf{propagate}(\hat{q})$ and $\mathsf{propagate}(q)$. Since $F \in label[q_n]$ holds at the beginning of this search and because q' is an immediate successor of q_n, F will be added into $label[q']$ during $\mathsf{propagate}(q_n)$ if it is not there already. Thus $F \in label[q']$ will hold also upon returning from $\mathsf{propagate}(q)$. This completes the induction. □

We can now prove the completeness of the algorithm.

Theorem 2. *Let $\mathcal{A} = (\Sigma, Q, \Delta, q_I, \mathcal{F})$ ($\mathcal{F} \neq \emptyset$) be a Büchi automaton given as input for the algorithm. If $\mathcal{A}$ has an accepting run, the algorithm exits with the value* "TRUE"*.*

Proof. Because the automaton has an accepting run, it contains a maximal nontrivial strongly connected component (i.e., a maximal subset of states, all of which are reachable from each other in the automaton via a nontrivial path) reachable from the initial state q_I such that the component intersects all acceptance conditions.

Assume that the algorithm exits with the value "`FALSE`". In this case the main depth-first search will visit all states reachable from q_I. Let $C \subseteq Q$ be the first component satisfying the above condition entered in the main search with $q \in C$ as the first state of C pushed on the *path* stack. By the choice of q, the main search will not backtrack from q until all states in C have been visited. Let $F \in \mathcal{F}$ be an acceptance condition, and let $q_F \in C \cap F$. The algorithm will start a nested search from q_F before backtracking from q.

If q is $\mathtt{p}(q_F)$-reachable, $F \in \mathit{label}[q]$ holds after the nested search by Lemma 4. This applies especially to the case $q_F = q$, since q is certainly $\mathtt{p}(q)$-reachable from itself. Otherwise $q_F \neq q$, and q is not $\mathtt{p}(q_F)$-reachable. Let x be any nontrivial path from q_F to q in which no state occurs twice. Clearly, any such path is also contained in C. There are two cases:

- x is entirely contained in $\mathit{path}_{q_F} \cup \mathit{processed}_{q_F}$. Because q is not $\mathtt{p}(q_F)$-reachable, however, x must intersect path_{q_F} strictly between its endpoints. The first such state occurring along x is $\mathtt{p}(q_F)$-reachable.

- x contains a state q' with an immediate successor q'' (in x) such that all states occurring along x up to (and including) the state q' belong to $\mathit{path}_{q_F} \cup \mathit{processed}_{q_F}$, but $q'' \notin \mathit{path}_{q_F} \cup \mathit{processed}_{q_F}$. By Lemma 3 (c), it follows that $q' \in \mathit{path}_{q_F}$. In addition, $q' \neq q_F$ by Lemma 3 (b), and $q' \neq q$, since q has no successor in x. Thus, x intersects path_{q_F} strictly between its endpoints, and there exists a $\mathtt{p}(q_F)$-reachable state $\hat{q} \in \mathit{path}_{q_F}$, $\hat{q} \neq q_F$, $\hat{q} \neq q$.

Thus, if $q_F \neq q$ and q is not $\mathtt{p}(q_F)$-reachable, we can find a $\mathtt{p}(q_F)$-reachable state $\hat{q} \in \mathit{path}_{q_F} \cap C$ ($\hat{q} \neq q_F$, $\hat{q} \neq q$). By Lemma 4 we see that $F \in \mathit{label}[\hat{q}]$ will hold after returning from $\mathtt{propagate}(q_F)$. Because $\hat{q} \neq q_F$ and $\hat{q} \in \mathit{path}_{q_F}$, it follows that the algorithm will start another nested search from $\hat{q}$ after returning from $\mathtt{propagate}(q_F)$. On the other hand, by the choice of q, this search will start before the main search backtracks from q.

When the algorithm starts a nested search from $\hat{q}$, there are again two possibilities: either q is $\mathtt{p}(\hat{q})$-reachable, in which case $F \in \mathit{label}[q]$ will hold after the search (by Lemma 4), or there exists a $\mathtt{p}(\hat{q})$-reachable state $\hat{q}' \in \mathit{path}_{\hat{q}} \cap C$ ($\hat{q}' \neq \hat{q}$, $\hat{q}' \neq q$) such that $F \in \mathit{label}[\hat{q}']$ holds after the nested search from $\hat{q}$, and the algorithm will start a nested search from $\hat{q}'$ before backtracking from q.

By repeating the above reasoning if necessary, we are bound to find a state $\bar{q}$ from which q is $\mathtt{p}(\bar{q})$-reachable. Therefore, F will be added to $\mathit{label}[q]$ (at the latest) during the nested search rooted at $\bar{q}$.

Since F is arbitrary, we can conclude that $\mathit{label}[q] = \mathcal{F}$ holds at line 11 when q is on top of the *path* stack. Hence, the algorithm exits with the value "`TRUE`", contrary to assumption. This proves the completeness of the algorithm. □

4 DISCUSSION

4.1 Disadvantages

Although generalization improves the memory requirements of the nested depth-first search algorithm, it results also in some easily seen disadvantages. For example, the generalized algorithm does not support extracting full counterexamples for the given specifications directly from the internal data structures. Although we can still extract a path to a state belonging to an accepting cycle when the existence of an accepting run can be confirmed, constructing the cycle itself requires an additional search in the automaton in the general case. (This search can be implemented, for example, as a simple variant of the presented algorithm.) The main benefits of the reduced memory usage are achieved whenever an exhaustive search of the product space is needed: the generalized algorithm may therefore be able to complete the successful verification of some properties with fewer resources than an implementation based on the classic algorithm. Whether the theoretical memory savings of the generalized algorithm are in practice significant enough to justify an additional search in the automaton in the case of a failed verification run can be evaluated only with careful testing against previous algorithms.

Additionally, although the worst case running time of the generalized algorithm remains the same as in the basic case, the algorithm may repeat a nested search several times (with different labels) in the same subgraph of the product even if it does not contain any accepting runs. Apart from limiting the nested search only to states visited also during the main search, there are no obvious heuristics that would remedy this problem without any extra memory overhead while still retaining the completeness of the algorithm. (It is possible, however, to modify the algorithm to detect the existence of an accepting run already during the nested search: the search can be aborted with the answer "`TRUE`" if the nested search enters a state $q \in path$ such that $label[q] \cup labels\_to\_propagate = \mathcal{F}$.)

An unfortunate consequence of restricting the nested search to states visited only in the main search is that the algorithm becomes incompatible with *state space caching* techniques [12], which reduce the memory requirements of the search at the risk of repeating the search multiple times in parts of the automaton; the completeness of the algorithm is not preserved (for example, in the extreme case when only the states in the *path* stack are kept in the state space cache). This also makes it impossible to apply any heuristics for choosing which states to keep in the set of visited states (see, for example, [2]).

4.2 Simple Extensions

The algorithm extends directly to generalized Büchi automata with multiple initial states. It is straightforward to check that the algorithm still remains sound and complete (with perfect state hashing) if the search is repeatedly restarted (without data structure reinitialization) from a previously unvisited initial state until the algorithm either exits with the value "`TRUE`" or all initial states have been explored.

Obviously, the algorithm is not restricted to generalized Büchi automata;

it can be applied (with the above extension) to any directed graph with "multicolored" vertices (where the colors correspond to generalized Büchi acceptance conditions) to decide the existence of a cyclic path covering at least some fixed subset (or number) of colors.

4.3 Compatibility with Related Work

Correctness of a system is often checked with respect to various *fairness assumptions* (see, for example, [9]). While it is often possible to embed these assumptions into the correctness specification, they can alternatively be encoded directly into the system model instead (see, for example, [19, 20]) to reduce the complexity of the specification automaton.

In particular, state-based *unconditional* or *weak fairness* assumptions reduce to conditions on the infinite occurrence of states satisfying some state predicate in any computation path of the system. Such conditions correspond directly to generalized Büchi acceptance conditions imposed on the system components [1]. The generalized algorithm handles these assumptions directly due to its ability to decide the emptiness of the product of any finite number of generalized Büchi automata. *Strong fairness*, however, cannot be easily expressed as (generalized) Büchi acceptance and cannot thus be handled by the algorithm presented here.

Explicit state model checking tools often employ probabilistic model checking techniques such as *bitstate hashing* [15] or *hash compaction* [25, 21] to reduce the state storage memory requirements at the risk of missing some errors in the system. Apart from the requirement that the states in the depth-first search stack need to be hashed perfectly, the algorithm is otherwise compatible with all of these techniques by the soundness proof of Sect. 3.3 (with bitstate hashing, the label bits of each state can be stored into the hash table using multiple hash functions, which corresponds to the *multihash* technique analyzed in [25] and [16]). The number of states in the depth-first search stack that need to be stored simultaneously in the perfect hash table depends on the maximum length of any distinct-state path starting from the initial state of the automaton; the costs of perfect hashing may still be tolerable if this length is small in comparison with the size of the automaton.

In [6], the maximal strongly connected components of the specification automaton (with one acceptance condition) are classified according to the existence of cycles fulfilling the condition in the component. This information is then used to detect accepting runs early during the main depth-first search. In addition, an equivalence relation between states is used to restrict the set of states to visit during the nested search. These ideas generalize to the algorithm presented in this paper; however, there may be fewer opportunities to apply the early cycle detection heuristic successfully in the presence of multiple acceptance conditions.

ACKNOWLEDGMENTS

This work was supported by Helsinki Graduate School in Computer Science and Engineering (HeCSE), the Academy of Finland (Project 53695) and the Nokia Foundation.

The author is also grateful to Keijo Heljanko, Tommi A. Junttila, and the anonymous reviewers of CONCUR 2002 and SPIN 2003 for many helpful comments and valid criticisms.

References

[1] S. Aggarwal, C. Courcoubetis, and P. Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, 1990.

[2] L. Brim, I. Černá, and M. Nečesal. Randomization helps in LTL model checking. In *Proc. Joint Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM-PROBMIV 2001)*, volume 2165 of *Lecture Notes in Computer Science*, pages 105–119. Springer-Verlag, 2001.

[3] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *Journal of Computer and System Sciences*, 8:117–141, 1974.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

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

[6] S. Edelkamp, S. Leue, and A. Lluch Lafuente. Direct explicit-state model checking in the validation of communication protocols. Technical Report 161, Computer Science Department, University of Freiburg, 2001.

[7] K. Etessami and G. J. Holzmann. Optimizing Büchi automata. In *Proc. 11th Int. Conf. on Concurrency Theory (CONCUR'2000)*, volume 1877 of *Lecture Notes in Computer Science*, pages 153–167. Springer-Verlag, 2000.

[8] K. Etessami, Th. Wilke, and R. Schuller. Fair simulation relations, parity games, and state space reduction for Büchi automata. In *Proc. 28th Int. Colloquium on Automata, Languages and Programming (ICALP'2001)*, volume 2076 of *Lecture Notes in Computer Science*, pages 694–707. Springer-Verlag, 2001.

[9] N. Francez. *Fairness*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[10] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. IFIP WG6.1 15th Int. Symp. on Protocol Specification, Testing, and Verification (PSTV'95)*, pages 3–18. Chapman & Hall, 1995.

[11] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proc. IFIP TC6/WG6.1 13th Int. Symp. on Protocol Specification, Testing, and Verification (PSTV'93)*, pages 109–124. North-Holland, 1993.

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

[13] J.-Ch. Grégoire. State space compression in SPIN with GETSs. In *Proc. 2nd* SPIN *Workshop*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.

[14] S. Gurumurthy, R. Bloem, and F. Somenzi. Fair simulation minimization. In *Proc. 14th Int. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 610–624. Springer-Verlag, 2002.

[15] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[16] G. J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, 1998.

[17] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth-first search. In *Proc. 2nd* SPIN *Workshop*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.

[18] G. J. Holzmann and A. Puri. A minimized automaton representation of reachable states. *International Journal on Software Tools for Technology Transfer*, 2:270–278, 1999.

[19] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In *Proc. 25th Int. Colloquium on Automata, Languages, and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1998.

[20] T. Latvala and K. Heljanko. Coping with strong fairness. *Fundamenta Informaticae*, 43(1–4):175–193, 2000.

[21] U. Stern and D. Dill. A new scheme for memory-efficient probabilistic verification. In *Proc. IFIP TC6 WG6.1 Joint Int. Conf. on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV'95)*, pages 333–348. Kluwer, 1996.

[22] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[23] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. Symp. on Logic in Computer Science (LICS'86)*, pages 332–344. IEEE Computer Society Press, 1986.

[24] W. Visser and H. Barringer. Memory efficient state storage in SPIN. In *Proc. 2nd* SPIN *Workshop*, volume 32 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997.

[25] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. 5th Int. Conf. on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer-Verlag, 1993.

HELSINKI UNIVERSITY OF TECHNOLOGY LABORATORY FOR THEORETICAL COMPUTER SCIENCE
RESEARCH REPORTS

ISBN 951-22-6667-9
ISSN 1457-7615