A Survey of Graph-based Algorithms in Software Model Checking

Arie Gurfinkel
University of Toronto
arie@cs.utoronto.ca

Abstract

In this paper we survey different graph-based algorithms that can be used to solve the automata-theoretic model-checking problem. We concentrate on existing depth-first search algorithms to locate strongly-connected components, and describe different optimization and approximation techniques that can be utilized in conjunction with them. We also suggest how a breadth-first search can be used as a replacement to depth-first search in order to minimize the length of the error trace.

1 Introduction

Program verification has been an important topic in computer science research since the development of the first programming language. While programs are written and designed by humans, there will always be a possibility of an error. Testing and debugging are commonly used during the development cycle of a program to locate and fix these errors. However, these techniques do not perform well when applied to concurrent and distributed programs. A number of techniques have been developed under the name of *automated verification*, to allow to automatically analyze behaviors of concurrent and distributed programs. In this paper we survey algorithms present in one of such techniques, known as *automata-theoretic model-checking* [1].

In the *model-checking* approach to program verification, a finite-state transition system is first constructed from the program. This *model* is then verified against a given temporal formula. In *automata-theoretic* model-checking both the formula and the model of the program are combined together into a graph representing the program behavior. The model-checking procedure is then reduced to finding strongly-connected components of this graph.

In this paper we survey a progression of graph-theoretic algorithms that can be applied to find strongly-connected components of a given graph. We analyze how they satisfy the specific requirements posed by model-checking. We also discuss techniques that are used in model-checking to combat the state explosion problem, and how these techniques can be applied to the algorithms. Finally, we analyze how these algorithms can be further improved.

The rest of the paper is structured as follows. Section 2 provides details about automata-theoretic model-checking. Section 3 states the problem we are considering formally, and outlines the requirements that must be met by an algorithm to be used as a solution. Section 4 provides an analysis of the depth-first search algorithm. Section 5 describes a number of techniques that can be applied to depth-first search to combat the state explosion problem. Sections 6–8 describe in detail different depth-first search based algorithms that can be used to solve the model-checking problem. Section 9 describes our alternative algorithm based on the algorithm in Section 8. Finally, Section 10 concludes the paper.

2 An automata-theoretic approach to model-checking

The automata-theoretic approach to model-checking that we consider here can be summarized as follows. Given a program P, described as a concurrent execution of finite-state transition systems P_i, and a temporal logic formula φ, check that all infinite computations of P satisfy φ. Propositional temporal logic (LTL) [2] is used to describe the property φ. However, since the details of this logic are not important to the rest of the discussion they are omitted here. In order to solve the model-checking problem the following steps are used:

1. Build a Büchi-automaton corresponding to the negation of the formula φ, and call it $A_{\neg\varphi}$.

2. Build a Büchi-automaton corresponding to the program P by computing the product $\prod P_i$ of the transition systems P_i.

3. Finally, build a Büchi-automaton corresponding to the intersection of the automatons $A_{\neg\varphi}$ and P.

4. The language accepted by the Büchi-automaton constructed in the previous step corresponds to all infinite computations of P that violate the property φ. Thus if the automaton is empty, the property is satisfied; otherwise it is violated.

In order to describe the following algorithms we first need to provide some details about Büchi automata and how to decide their emptiness.

A *Büchi-automaton* is a tuple $\{\Sigma, S, s_0, F, T\}$, where

- Σ is an alphabet,
- S is a set of states,
- $s_0 \in S$ is an initial state,
- $F \subseteq S$ is a set of *accepting* states, and

- $T \subseteq S \times \Sigma \times S$ is a transition relation.

A Büchi-automaton can be represented by a digraph with vertices representing the states, and labeled edges representing the transitions. A formal definition of this graph is presented later in this paper. An infinite path through this graph is called a *run* of the automaton. More formally, a *run* of a Büchi-automaton over an infinite word $w = (\alpha_0\alpha_1 \ldots)$ is an infinite sequence of states $\sigma = (s_0, s_1, \ldots)$ such that s_0 is the initial state, and $\forall_{i \geq 0}(s_i, \alpha_i, s_{i+1}) \in T$. A run is called *accepting* if some *accepting* state occurs infinitely often in it. More formally, let $\sigma(i)$ denote the suffix of a run σ starting from position i; the run is accepting if and only if $\exists f \in F$ such that $\forall_{i \geq 0} f \in \sigma(i)$. An infinite word w is accepted by a Büchi-automaton A if there is an accepting run of A over w. The set of all infinite words accepted by a Büchi-automaton A is called a language of A and is denoted $\mathcal{L}(A)$.

When considering a Büchi-automaton corresponding to a program P, the alphabet Σ is interpreted as all possible actions of P. In this case, each word $w \in \mathcal{L}(P)$ corresponds to an infinite computation of P. Similarly, for the Büchi-automaton A_φ corresponding to an LTL formula φ, each word $w \in \mathcal{L}(A_\varphi)$ is interpreted as an infinite computation of P that satisfies φ.

Let two Büchi automata A and B be given. The language $\mathcal{L}(A \cap B)$ corresponding to the intersection of the two automata consists of infinite words common to both $\mathcal{L}(A)$ and $\mathcal{L}(B)$. Therefore, the language $\mathcal{L}(A_{\neg\varphi} \cap P)$ corresponds to all infinite computations of P that satisfy the property $\neg\varphi$ (and therefore violate φ). Therefore, solving the model-checking problem is equivalent to deciding the emptiness of the language $\mathcal{L}(A_{\neg\varphi} \cap P)$.

3 Problem statement and algorithm requirements

In this paper we describe graph based algorithms to decide emptiness of Büchi automata, but first we need to state the problem formally using graph-theoretic terms. A Büchi-automaton is represented by a digraph $G(V, E)$, where V are the vertices of G, and E are the edges. Given a Büchi-automaton $A = \{\Sigma, S, s_0, F, T\}$ a graph $G(V, E)$ is defined as follows. Let $V = S$, $F \subseteq V$, and an edge $uv \in E$ if and only if $\exists \alpha \in \Sigma$ such that $(u, \alpha, v) \in T$. From the definition of a Büchi-automaton it follows that a Büchi-automaton is not empty if and only if there exists a strongly-connected component (SCC) in G, containing at least one vertex f from F, and reachable from s_0. Existence of such a strongly-connected component clearly implies the existence of an infinite accepting run through f. The problem stated in graph-theoretic terms becomes

Problem 3.1 (Emptiness of a Büchi-automaton) *Given a digraph $G(V, E)$, a subset of vertices F, and an initial vertex $s_0 \in V$, find a strongly-connected component (SCC) C of G, such that C is reachable from s_0, and C contains at least one vertex from F.*

Notice that if the strongly-connected component is found, the Büchi-automaton corresponding to G is not empty and the property φ is violated, otherwise, the

Büchi-automaton is empty and the property is satisfied.

In this paper we will present a number of algorithms that can be used to solve this problem. We evaluate them based on their memory requirement, ability to provide error traces, and ability to be adapted to produce approximate solutions when an exact solution cannot be obtained due to hardware limitations.

The memory requirement is the most important one. It is the limiting factor, dictating the size of the problem that can be solved by the algorithm. Therefore, we pay special attention to the data structures used by the algorithm. The total amount of memory required by these data structures is the total amount of memory required by the algorithm. We classify these data structures by their access pattern as *randomly-accessed*, and *sequentially-accessed*. It was observed in [3] that performance of the algorithm can be dramatically improved if the amount of randomly-accessed memory required by the algorithm does not exceed the amount of physical memory available. This follows from the fact that if the algorithm requires much more randomly-accessed memory than is physically available, the amount of paging involved will dramatically reduce its performance (thrashing). On the other hand, sequentially-accessed data structures can be efficiently maintained in secondary memory, such as a hard disk. Therefore, when assessing the memory requirement of a given algorithm, we will only concentrate on the amount of randomly-accessed memory required by it.

In general, any algorithm that decides whether a given graph contains a strongly-connected component containing a vertex marked as accepting and produces 'yes', and 'no' answers, is an acceptable solution to the problem as stated so far. However, such an algorithm would not be practical. The goal of any model-checking tool is to help the user identify and correct errors in the program, and thus, in a case of a "no" answer, the algorithm must provide an execution trace to exemplify an erroneous behavior of the program. Such an execution, known as the *error trace*, can be presented to them in terms of a simulation, as in model checker SPIN [4], and is used for understanding why the property was violated. From the algorithm point of view, this implies that it is not enough to decide whether a strongly-connected component exists, but in addition to this, a path through the graph corresponding to an infinite accepting run must be explicitly computed. Moreover, since this path will be examined by a human user, it is important for this path to be as short as possible.

Another important criterion in evaluating an algorithm suitable for model-checking is its ability to tackle even problems that cannot be solved explicitly given current hardware constraints. In general, the algorithms considered here proceed by explicit state exploration. This implies that potentially the whole graph G will have to be stored in memory. Unfortunately, in practice, this makes model-checking applicable only to small, "toy", examples. It is, therefore, important to develop algorithms that can be used to provide good approximations if the exact solution is not possible to achieve. One way to achieve this is to develop generic techniques that reduce the amount of memory required to store the graph, and adapt exact algorithms to be compatible with them. Therefore, it is important that algorithms considered for model-checking be compatible

```
proc DFS(v)
  add v to Visited
  d[v] := time
  time := time + 1
  for all w ∈ succ(v) do
    if w ∉ Visited then
      DFS(w)
    end if
  end for
  f[v] := time
  time := time + 1
end proc
```

Figure 1: The depth-first search.

with such techniques.

4 The depth-first search

Before describing the actual algorithms that can be used to solve the emptiness problem of a Büchi-automaton, let us first describe a traditional depth-first algorithm on which these algorithms are based. A depth-first search algorithm is a technique to construct and examine a graph given its root vertex s_0. It proceeds by recursively exploring each path through a successor of s_0. An implementation of this algorithm is given in figure 1.

To avoid redundant exploration, the algorithm maintains the list of all visited vertices in a global table *Visited*. In addition to this, for each vertex two numbers representing the discovery and finishing times are kept in arrays d and f, respectively. The discovery time corresponds to the time when the vertex was first discovered, and the finishing time corresponds to the time when all of its successors have been explored. These times are often used by different DFS-based algorithms, and are also useful when describing different properties of the DFS algorithm itself.

A subgraph $G_\pi(V, E_\pi)$ of G, where an edge $uv \in E_\pi$ if the vertex v was first discovered as a successor of u, is called the *depth-first forest*. It is used to describe the following two properties of the depth-first search.

Theorem 4.1 (Parenthesis theorem) *[5]In any depth-first search of a graph $G = (V, E)$, for any two vertices u and v, exactly one of the following three conditions holds:*

- *the intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint,*
- *the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$, or*

- *the interval $[d[u], f[u]]$ is contained entirely within the interval $[d[v], f[v]]$.*

Corollary 4.1 (Nesting of descendants' intervals) *[5]Vertex v is a proper descendant of vertex u in the depth-first forest for a graph G if and only if $d[u] < d[v] < f[v] < f[u]$.*

Corollary 4.1 is immediate from Theorem 4.1, and the proof of Theorem 4.1 is available in [5].

Depth-first search is linear in the size of the input graph since it examines every vertex exactly once. However, this is true only if the lookups of the *Visited* table take constant time. Therefore, this table is usually implemented as a hashtable. In the following discussions we will assume that this is the case in the algorithms we are considering.

We now examine the memory requirements of the depth-first search algorithm. As was stated before, we make a distinction between random-access and sequential-access memory required. A basic implementation that does not maintain the discovery and finishing time will make use of two data structures: a stack to maintain the vertices that are currently being visited, and the *Visited* table. In the recursive implementation the stack is stored in main memory. However, in an iterative implementation of the algorithm the stack is local to the algorithm, and being sequentially-accessed, the data structure can be stored in secondary memory. The *Visited* table, however, is accessed randomly, and therefore must be kept in main memory. It follows that the size of this table determines the size of the graph that can be explored by the algorithm, given the physical constraints of the machine on which it is executing. The size of this table depends on two factors: the number of vertices of the input graph, n, and the number of bits S needed to uniquely identify each vertex. The actual table size is then given by $n \times S$, plus an additional overhead required to maintain the hashtable. However, if the algorithm is also required to store the discovery and/or finishing time of each vertex, an additional overhead of order $O(\log n)$ bits per vertex is incurred. Therefore, the memory required by any DFS based algorithm, that guarantees that all of the vertices of the input graph are explored, is of order $O(n \times S)$, if discovery and finishing times are not maintained, and is of order $O(n \times S \times \log n)$ otherwise. This distinction may not seem significant unless n is large. This is the case with graphs that are considered during the model-checking procedure. It is suggested in [4] that the value of $\log_2 n$ must be assumed to be at least 32.

5 Techniques to combat sate explosion problems

The depth-first search provides full coverage of a graph when enough memory is available to store the complete graph, or more precisely, when enough memory is available to store all of the vertices of the graph in the *Visited* table. However, the coverage degrades dramatically when not enough memory is available. This is a major problem, since it has been shown in [6] that for typical problem sizes, and typical amounts of memory available the coverage is just 1% of the full graph. Clearly a technique is required to extend the coverage of the search for these cases. Moreover, since the size of the graph is not known beforehand, such a technique must also provide good coverage when enough memory is available to perform a complete search.

As explained in Section 4, the size of the *Visited* table depends on the size of the graph, and the size (S) of each table entry. Since the size of the graph is fixed, the only way to reduce the size of the table is to reduce S. In this case it is possible that the search will not be able to distinguish between two different vertices, and will conclude that a certain vertex was visited, when it was not. The expected coverage of such an algorithm will depend on the probability of confusing two vertices, and the structure of the graph.

The first technique to reduce the size of S was proposed in [3], under a name *bitstate hashing*. It is further analyzed and compared to alternative techniques in [7]. This technique is based on the following argument. The visited vertices have to be stored in a hashtable *Visited*. That is, a hash value must be computed for each vertex, and is then used to store the vertex. If the size of the *Visited* table is large compared to the number of vertices, the possibility of collision is very low, and the hash value itself is enough to uniquely identify a vertex. In this case, the *Visited* table can be implemented as a bit vector, where a bit is turned on at position h when a vertex with a hash value of h is added to the table. This technique can be further improved by using multiple independent hash functions. It is suggested in [7] that for typical problems seen in practice using two hash functions gives the most benefit. In this case, two hash values are calculated for each visited vertex, and two bits are turned on in the *Visited* table. A vertex is considered to be visited if and only if both bits corresponding to its two different hash values are turned on. Again, in [7] it is shown that if the ratio between the amount of memory and number of vertices in a graph is around a 100, the expected coverage provided with bitstate hashing technique is roughly 99%. An additional benefit of this approach is that the expected coverage decreases gradually when this ratio is less than optimal.

An alternative technique, based on the same principle, is proposed in [8] under the name *hashcompact*. There, it is suggested that instead of using multiple hash functions to enhance bitstate hashing, it is possible to increase the number of bits stored per each vertex. This is achieved by first computing for each vertex a hash value for a virtual hashtable, much larger than the available physical memory. It is suggested that for typical problems the hashtable should contain at least 2^{64} entries. The computed hash value is then stored in a traditional hashtable with collision resolution. The advantage of this technique is that the probability of not visiting a given vertex is less than 10^{-3}. Notice that this a much stronger estimate than the expected coverage. The only downside of this technique is that even though it performs better than the two-bit bitstate hashing technique when the ratio between the size of the physical memory and the size of the graph is more than 64, the coverage degrades quickly when the ratio is smaller.

The expected coverage can be further increased for both of these techniques at the expense of running time. This can be achieved by executing the search

```
proc SCC − SEARCH(v)
  add v to Visited
  d[v] := time
  time := time + 1
  lowlink[v] := d[v]
  push v on STACK
  for all w ∈ succ(v) do
    if w ∉ Visited then
      SCC − SEARCH(w)
      lowlink[v] := min(lowlink[v], lowlink[w])
    else if d[w] < d[v] and w is on STACK then
      lowlink[v] := min(d[w], lowlink[v])
    end if
  end for
  if lowlink[v] = d[v] then
    repeat
      pop x from top of STACK
      if x ∈ F then
        terminate with "Yes"
      end if
    until x = v
  end if
  terminate with "No"
end proc
```

Figure 2: Tarjan's algorithm for finding maximal SCCs

algorithm multiple times, using different independent hash functions in each run.

The biggest advantage of these techniques is that they can be used to convert any depth-first search based algorithm from an exact algorithm, into an approximation algorithm. However, they can only be applied if no additional information, such as the discovery time, is stored for each vertex.

6 Tarjan's algorithm to find maximal strongly-connected components

The first algorithm we present here is a well known algorithm for finding all maximal strongly-connected components of a graph G. This algorithm was presented by Tarjan in [9]. In here we will present the algorithm as it was presented in [10]. Given an initial vertex s_0 the algorithm explores the graph G in a depth-first manner, sequentially outputting each maximal strongly-connected component. Clearly, all strongly-connected components found by this algorithm are reachable from s_0. An implementation of this algorithm is given in figure 2.

The algorithm can be easily adapted to solve the emptiness problem of a Büchi-automaton as follows. As soon as a strongly-connected component was found, a check is made to see if it contains a vertex from F. If this is the case, the algorithm terminates claiming that the Büchi-automaton is not empty. If no such strongly-connected component was found, the algorithm terminates claiming that the Büchi-automaton is empty.

This algorithm is based on the following idea. Each strongly-connected component of G forms a tree in the depth-first forest of G. A root r of this tree is called the root of a strongly-connected component. Discovering the strongly-connected components of G is equivalent to discovering these roots in the order they are finished by the depth-first search, i.e. by their finishing time. In order to decide whether a vertex is a root of a strongly-connected component, a value called the *lowlink number* is computed as follows:

$$lowlink[v] = \min(d[v], d[w] | \quad \begin{array}{l} \exists u \text{ a descendant of } v \\ \text{such that } uw \in E, \text{ and} \\ w \text{ belongs to the same SCC as } v) \end{array}$$

where $d[v]$ is the discovery time of the vertex v. It can be shown that the roots of strongly-connected components are exactly those vertices for which the lowlink number equals their discovery time [10].

The algorithm consists of a single depth-first sweep of the input graph. During this sweep the lowlink number for each vertex is incrementally computed (lines 5, 10, and 12), and the visited vertices are placed on the stack $STACK$ based on their discovery time. Once all of the successors of a vertex have been visited, its lowlink number is known. At this time (lines 15-22), a check is made to see whether this vertex is a root of a strongly-connected component. If this is the case, the vertices of that strongly-connected component are removed from the stack. More information about this algorithm, as well as the proof of correctness, can be found in [10].

Let us analyze the advantages and disadvantages of this algorithm. The running time of the algorithm is linear in the size of the graph. Moreover, the algorithm can terminate as soon as the first strongly-connected component containing a vertex from F is found. This means that in practice, only a part of the graph may need to be constructed. Moreover, the information required to generate an error trace can be extracted from the $STACK$. The error trace obtained this way will not be the shortest error trace possible, but this is the case with any depth-first search based algorithm. The biggest disadvantage of this algorithm is the amount of additional memory required by it. Since it makes use of the discovery time, and has to maintain the lowlink number for each vertex, an addition memory requirement of size $2n \log n$ bits is incurred. Furthermore, this algorithm is not compatible with either bitstate-hashing, or the hashcompact techniques described above. That means that it cannot be easily adapted to provide good approximate results when there is not enough memory to store the entire graph.

7 The double depth-first search algorithm

The first version of a memory-efficient algorithm to decide emptiness of a Büchi-automaton was proposed in [11]. It is based on the fact that to decide the emptiness of a Büchi-automaton it is sufficient to find a cycle containing a vertex from F in the reachability graph of s_0. Unlike Tarjan's, algorithm the algorithm suggested there is compatible with both bitstate-hashing and hashcompact technique, and can be implemented with just $O(n)$ bits of randomly-accessed memory when combined with them. An implementation of this algorithm is provided in figure 3.

The algorithm consists of two sweeps of the depth-first search. During the first sweep the vertices from F are ordered in decreasing order based on their finishing time. The second sweep starts from each of these vertices and tries to locate a cycle containing them. What makes this algorithm linear is the fact that if f_i and f_j are members of F such that $f[f_i] < f[f_j]$, and there does not exist a cycle containing f_i, then none of the vertices reachable from f_i can belong to a cycle containing f_j. This fact follows from the following lemma,

```
proc DFS1(v)
  add v to Visited
  for all w ∈ succ(v) do
    if w ∉ Visited then
      DFS1(w)
    end if
  end for
  if v ∈ F then
    add v to Q
  end if
end proc
```

(a)

```
proc SWEEP2(Q)
  while Q ≠ [] do
    f := deque(Q)
    DFS2(f, f)
  end while
  terminate with "No"
end proc
```

(b)

```
proc DFS2(v, f)
  add v to Visited
  for all w ∈ succ(v) do
    if v = f then
      terminate with "Yes"
    else if w ∉ Visited then
      DFS2(w, f)
    end if
  end for
end proc
```

(c)

Figure 3: Double DFS algorithm. (a) The first DFS sweep to order vertices of F. (b) Helper procedure that starts the second DFS sweep from each vertex in F. (c) The second DFS sweep to locate cycles.

presented in [11] for the proof of correctness of the algorithm.

Lemma 7.1 *Given a digraph $G(V, E)$, and two vertices $u, v \in V$. If in any depth-first search $f[u] < f[v]$, and there exists a u, v path p in G, then there exists a cycle containing u in G.*

Proof: Suppose there exists a u, v path p. If $\forall w \in p\ d[w] < d[u]$ then at the time vertex u was discovered by the depth-first search, none of the vertices on the path p where visited. Therefore, the depth-first search can follow this path making u an ancestor of v in the depth-first forest. By corollary 4.1 it then follows that $f[v] < f[u]$ which contradicts our choice of u. Therefore, there exists $w \in p$ such that $d[w] < d[u]$. If $f[w] < f[u]$, then, since v is reachable from u, by theorem 4.1 it follows that $f[v] < f[u]$, which again contradicts our choice of u. Therefore, there exists $w \in p$ such that $d[w] < d[u]$ and $f[u] < f[w]$, which by corollary 4.1 implies that w is a proper ancestor of u in the depth-first forest. Thus u is contained in a cycle consisting of a path from w to u and a path from u to w. □

The proof of correctness of this *double DFS* algorithm proceeds as follows.

Theorem 7.1 *The second DFS halts and returns "Yes" if and only if there exists a reachable vertex in F that belongs to a strongly-connected component.*

Proof: The first part of the proof is simple. If the second DFS halts and returns "Yes", then it has found a path that starts from a reachable vertex in F and ends at the same vertex. Therefore, there exists a reachable vertex of F that belongs to a strongly-connected component. The second part of the proof is slightly more difficult. Let f_j be the first vertex in F, as ordered by the first DFS, contained in the strongly-connected component. That is, $f_j \in F$, and there exists a strongly-connected component containing f_j, and $\forall f_i \in F$ $f[f_i] < f[f_j]$ implies that f_i is not contained in any strongly-connected component. Since f_j belongs to a strongly-connected component, there exists a path p from f_j to itself. The second DFS will find this path provided that no vertices on it are reachable from any $f_i \in F$ for which $f[f_i] < f[f_j]$. Say there exists a vertex $w \in p$ that is reachable from some such f_i. Then there exists a path from f_i to f_j, and by lemma 7.1 it follows that there exists a strongly-connected component containing f_i. This contradicts our choice of f_j; therefore, the algorithm will find a strongly-connected component if one exists. $\Box$

The major advantage of this algorithm is its randomly-accessed memory requirement. Since the *Visited* table is only used to decide whether or not a vertex was visited and no other per vertex information is stored, the algorithm can be implemented in kn bits of randomly-accessed memory. Moreover, when combined with two bit bitstate-hashing technique, the value of k can be made as small as 2. The major disadvantage of this algorithm is that it is not clear how the information required to generate an error trace can be obtained. Since the two passes are completely separated, by the time when it is discovered that a certain vertex $f \in F$ is contained in a cycle, a path from s_0 to f is lost. Moreover, even though this is still a linear algorithm, it takes twice as much time as Tarjan's approach. This is due to the fact that in the worst case every vertex of G is visited twice.

8 The nested depth-first search algorithm

The first version of a nested depth-first search algorithm to decide an emptiness of a Büchi-automaton was presented in [11] as a modification of the double DFS algorithm. It is based on the following observation. Since the second DFS starts at each vertex in F based on its finishing time in the first DFS, the second DFS can be started as soon as the first such vertex is discovered. The algorithm then combines two depth-first search sweeps into one nested depth-first search. As soon as the first depth-first search finishes exploring a vertex $f \in F$, it is suspended and the second depth-first sweep is initiated. If the second sweep finds a cycle containing f, the algorithm terminates; otherwise, the first sweep is restarted until the next vertex in F is finished. The problem with this algorithm is that the amount of memory required for it is double the amount required for

proc $DFS1(v)$
  add $(v, 0)$ to $Visited$
  push v on $STACK$
  for all $w \in succ(v)$ **do**
    if $(w, 0) \notin Visited$ **then**
      $DFS(w)$
    end if
  end for
  if $w \in F$ **then**
    $DFS2(w)$
  end if
  pop $STACK$
end proc

(a)

proc $DFS2(v)$
  add $(v, 1)$ to $Visited$
  for all $w \in succ(v)$ **do**
    if $w \in STACK$ **then**
      terminate with "Yes"
    else if $(w, 1) \notin Visited$ **then**
      $DFS(w)$
    end if
  end for
end proc

(b)

Figure 4: The nested DFS algorithm. (a) The main depth-first search. (b) The nested depth-first search.

the double DFS algorithm. This is because two separate copies of the *Visited* table have to be kept in memory: one for each sweep.

A trivial solution was proposed to fix this problem in [12]. Instead of maintaining two separate copies of the *Visited* table, it is suggested to keep just one, where each entry will contain additional two bits: one bit to indicate that a vertex was visited by the first DFS, and one to indicate that it was visited by the second DFS. In this case the amount of additional memory required is virtually insignificant. An implementation of this algorithm is provided in figure 4.

The algorithm in figure 4 also utilizes an optimization suggested in [13]. If s_0 is the initial vertex on which the first DFS was started, and $f \in F$ is an accepting vertex that was just finished, then the stack of the first DFS contains a s_0,f-path. Therefore, the second, nested, DFS can terminate as soon as it has found a vertex w that is on the stack of the first DFS. This works, since the existence of such a vertex would imply that there is a cycle containing f, namely the cycle composed of a w, f and f, w paths.

As was noted before, the overhead in the memory requirement of the nested depth-first algorithm is insignificant compared to the memory requirements of the double DFS approach. In addition, enough information to generate an error trace is available on the stacks of two depth-first searches, when the second depth-first search terminates. The error trace can be generated as follows. When the second depth-first search is started, the first DFS contains a path from the initial vertex s_0 to some $f \in F$ on its stack. When the second DFS terminates, it contains a f,w-path, where w is some vertex in the stack of the first DFS, possibly f itself. Therefore, the error trace would consist of a s_0, w path, followed by a cycle consisting of all of the vertices on the stack of the first DFS after w, and the vertices from the stack of the second DFS.

Moreover, the nested DFS algorithm is compatible with bitstate hashing

```
proc BFS(v)
  add v to Visited
  Q := [v]
  while Q ≠ [] do
    u = head(Q)
    for all w ∈ succ(u) do
      if w ∉ Visited then
        enqueue(Q, w)
      end if
    end for
    dequeue(Q)
  end while
end proc
```

Figure 5: The breadth-first search.

and hashcompact techniques, and therefore, can be easily made into a good approximation algorithm, when required.

The proof of correctness of this algorithm is similar to the one for the double DFS algorithm, and can be found in [14].

9 Minimizing error traces through the use of breadth-first search

The nested depth-first search algorithm described above is a memory-efficient algorithm that can be applied to solve the emptiness problem of a Büchi-automaton. In addition it provides a simple method to generate an error trace, and is compatible with both bitsate hashing, and hashcompact techniques. The downside of this algorithm is that the error trace obtained from it is usually much larger than the smallest possible error trace. This presents a problem since the error trace is something that is examined by a human user. The larger such trace, the harder it is to analyze why the model-checking procedure failed. The reason for this is directly related to the use of depth-first search. Depth-first search proceeds by exploring the graph one path at a time. Therefore, the path currently explored is easy to extract from the algorithm, yet it is not guaranteed to be the shortest. An alternative technique, known as the breadth-first search (BFS), is usually employed when a shortest path is required. The breadth-first search explores all of the paths in the graph at once. This is accomplished by examining all of the immediate successors of a given vertex, before its descendants. The breadth-first search guarantees to find the shortest possible path; however, since all paths are explored at once, an individual path is hard to obtain from the algorithm.

An implementation of breadth-first search is presented in figure 5. Like the depth-first search examined before, it uses the *Visited* table to store all of the vertices that have been visited. The vertices whose children are yet to be explored are stored in the queue Q. Once again, only the *Visited* table must be kept in main memory, and Q can be stored on disk.

In order to consider breadth-first search as an alternative to the depth-first search for a solution of the model-checking problem, we first need to verify that it satisfies the following two requirements. First, that breadth-first search is compatible with bitstate hashing and hashcompact techniques, and provides results comparable to the depth-first search. Second, we need a method to generate the path to the solution once it was found by the breadth-first search. An implementation of breadth-first search that is compatible with hashcompact technique is presented in [15]. To recover the path, all that is required is that for each visited vertex a reference to its predecessor, as discovered by the breadth-first search, is stored. In this case, once the algorithm terminates at a vertex, the path to this vertex can be reconstructed by following its parent reference. Moreover, since this information is not used during the exploration of the graph, it can be stored in secondary memory. A technique to do that compactly is presented in [16, 15]. It is also mentioned that empirical evidence suggests that no performance penalty is incurred when this technique is applied.

Before describing the algorithm, let us first examine a typical error trace. From the graph-theoretic perspective, an error trace is just a path in the input graph. This path can be divided into two parts as follows. Given an initial vertex s_0, and a vertex f from F, an error trace is the union of a s_0, f-path, and an f-cycle. The s_0, f-path corresponds to a finite execution of a program that leads to unwanted behavior, and the f-cycle corresponds to an illegal infinite loop in the program. It is reasonable to assume that the user will spend most of his/her time examining this infinite loop since this is where the unwanted behavior is observed. Therefore, if it is not possible to efficiently minimize the complete error trace, it is beneficial to at least minimize the f-cycle.

Let us now examine if we can replace depth-first search with breadth-first search in the double DFS algorithm. As can be seen from the proof of correctness of the double DFS algorithm, it only depends on the properties of the first DFS. In fact, the second DFS can be replaced with any other search technique, provided it can be used to detect cycles. This immediately implies that we can replace the second depth-first search with a breadth-first search. The result is a DFS-BFS algorithm. Moreover, the nested DFS algorithm can be augmented to use breadth-first search as its nested algorithm in a similar fashion.

The proof of correctness of the DFS-BFS algorithm is the same as for the double DFS algorithm. The major advantage of the algorithm is that the error trace that is obtained from it is in general shorter than the one obtained with the double, or nested DFS algorithms. Moreover, it is guaranteed to minimize the f-cycle part of the error trace. The amount of randomly-accessed memory required by a nested version of this algorithm is the same as by the nested DFS. And as was mentioned before, it is compatible with the hashcompact technique, and therefore can be used as an approximation algorithm when required.

The next logical step would be to replace the first depth-first search with the breadth-first search as well. This, however, does not seem to be possible.

The ordering of F that is produced by the first DFS is what makes nested DFS into a linear algorithm. However, we cannot achieve the same ordering using the breadth-first search.

10 Conclusion

Model checking is an important part of automatic verification. Unlike other techniques, it completely automates the task of verification, and requires almost no user intervention. In this paper we have shown that the model-checking problem in the automata-theoretic approach reduces to a relatively simple graph-theoretic problem of finding strongly-connected components of a directed graph. We have surveyed the progression of algorithms that are used to solve this problem, starting from the classical algorithm by Tarjan, and concluding with a newer nested depth-first search algorithm.

The nested depth-first search algorithm is a linear time, memory-efficient algorithm to solve the model-checking problem. It is used in many model-checking tools available today. Empirical evidence from such tools shows that it can be successfully applied to many real life problems. Its only downside is that the error trace, that is generated by it, is usually much longer than otherwise possible. This in turn complicates analyzing large programs with it.

In [15] a memory-efficient version of breadth-first search was proposed to be used in the context of a model checker Murφ. It is also suggested that applying a breadth-first search instead of depth-first search to solve the model-checking problem may be an interesting topic for further research. This paper partially accomplishes this task. It was shown that the second, nested, depth-first search can be replaced with a breadth-first search in the nested depth-first search algorithm. Even though this new algorithm still does not provide the shortest possible error trace, the f-cycle part of the error trace is minimized.

References

[1] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings, Symposium on Logic In Computer Science*, pages 332–344. IEEE Computer Society, June 1986.

[2] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.

[3] G. Holzmann. An improved protocol reachability analysis technique. *Software, Practice and Experience*, 18(2):137–161, February 1988.

[4] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[5] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1997.

[6] G. Holzmann. Working material for the lectures of Gerard J. Holzmann on software model checking. In *Marktoberdorf Summer School 2000 on Engineering Theories of Software Construction*, 2000.

[7] G. Holzmann. An analysis of bitstate hashing. In *Proceedings of 15th International Conference on Protocol Specification, Testing and Verification*, 1995.

[8] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proceedings of 5th International Conference on Computer Aided Verification*, 1993.

[9] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1:146–160, 2 1972.

[10] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[11] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Proceedings of 2nd International Conference on Computer-Aided Verification*. Springer Verlag, 1990.

[12] P. Godefroid and G. Holzmann. On the verification of temporal properties. In *Proceedings of 13th IFIP Symposium on Protocol Specification, Testing, and Verification*, 1993.

[13] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proceedings of 2nd Workshop in the SPIN Verification System*, DIMACS. American Mathematical Society, August 1996.

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

[15] Ulrich Stern and David Dill. Using magnetic disk instead of main memory in the Murφ verifier. In *Proceedings of 10th International Conference on Computer-Aided Verification*, 1998.

[16] Ulrich Stern and David Dill. A new scheme for memory-efficient probabilistic verification. In *Proceedings of Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocal Specification, Testing, and Verification*, 1996.