Appears in: Formal Methods in System Design, vol. 1, 1992, pp. 275--288.

Memory-Efficient Algorithms for the Verification of Temporal Properties*

C. Courcoubetis† Inst. of Comp. Sci., FORTH, Crete, Greece‡

M. Vardi IBM Almaden§

P. Wolper¶ Un. de Liège||

M. Yannakakis AT&T Bell Labs**

Abstract

This paper addresses the problem of designing memory-efficient algorithms for the verification of temporal properties of finite-state programs. Both the programs and their desired temporal properties are modeled as automata on infinite words (Büchi automata). Verification is then reduced to checking the emptiness of the automaton resulting from the product of the program and the property. This problem is usually solved by computing the strongly connected components of the graph representing the product automaton. Here, we present algorithms which solve the emptiness problem without explicitly constructing the strongly connected components of the product graph. By allowing the algorithms to err with some probability, we can implement them with a randomly accessed memory of size $O(n)$ bits, where n is the number of states of the graph, instead of $O(n \log n)$ bits that the presently known algorithms require.

Keywords: Model-checking, verification, concurrent program, temporal logic, state-space exploration, hashing.

1 Introduction

Reachability analysis is one of the most successful strategies for analyzing and validating computer protocols. It was first proposed by West [24], and further studied by many researchers (cf. [15, 17]. Reachability analysis is applied to a protocol by systematically exercising all the protocol transitions. Such analysis can detect syntactical errors such as *static deadlock*, *unspecified reception*, or *unexercised code*. The simplicity of the strategy lends itself to easy implementation. Indeed, automated reachability analyses detected errors in published standards such as the X.21 (cf. [25]). The approach is less successful when it comes to *protocol verification*, i.e., verifying that the given protocol achieves its functional specification. This limitation is due to

*A preliminary version of this paper appeared in: Computer-Aided Verification '90, E.M. Clarke and R.P. Kurshan Editors, DIMACS series in Discrete Mathematics and Theoretical Computer Science, vol. 3, American Mathematical Society and Association for Computing Machinery, 1991.

†The work of this author is partially supported by ESPRIT-BRA project SPEC (3096)

‡Address: 36 Dedalou Street, P.O. Box 1385, 71110 Iraklio, Crete, Greece. Email: courcou@csi.forth.gr

§Address: Department K55/802, 650 Harry Road San Jose, California 95120-6099, U.S.A. Email: vardi@almaden.ibm.com

¶The work of this author is partially supported by ESPRIT-BRA project SPEC (3096)

||Address: Institut Montefiore, B28, B-4000 Liège Sart-Tilman, Belgium. Email: pw@montefiore.ulg.ac.be

**Address: 600 Mountain Avenue, Murray Hill, New Jersey 07974, U.S.A. Email: mihalis@research.att.com

the fact that a functional specification cannot be directly checked by reachability analysis. To apply reachability analysis to such a task, one first has to manually translate the functional specification to a property of the protocol state graph. While this can be done for some specific specifications (cf. [18]), it is not a general approach.

A general approach to protocol verification is to use a *theorem-prover* for an appropriate logic. Early systems used to focus on input/output behavior of protocols rather than on ongoing behavior (cf. [19]), but systems that are based on temporal logic overcame this shortcoming (cf. [8]). Unfortunately, theorem-proving systems are semi-automated at best, and their success at dealing with real-life protocols is not as impressive as that of reachability analysis (cf. [7]).

A new approach that emerged in the 1980's is the so-called *model-checking* approach [5, 4, 14, 16]. Model checking is based on the idea that verifying a propositional temporal logic property of a finite-state program amounts to evaluating that formula on the program viewed as a temporal interpretation. The algorithms for doing this are quite efficient, since their time complexity is a linear function of the size of the program[1]. As was shown later in the automata-theoretic approach of [21, 22, 26], model checking can be viewed as an augmented reachability analysis; the model-checking algorithm uses the temporal logic specification to guide the search of the protocol state space in order to verify that the protocol satisfies its functional specification. Model checking thus seems to solve one of the limits of reachability analysis: the inability to automatically verify functional specifications.

Model checking suffers, however, from the same fundamental problem plaguing the reachability-analysis approach: the ability to explore only limited-size state spaces. This problem, called the *state-explosion* problem, is the most basic limitation of both approaches. It has been the subject of extensive research both in the context of reachability analysis (cf. [15, 17]) and in the context of model checking (cf. [4]). A recent development [9] has substantially pushed back the state-explosion limit for reachability analysis. The main idea behind this development is that, at the price of possibly missing part of the state space, the amount of randomly accessed memory necessary for exploring a state space of a given size could be substantially reduced (essentially from $O(n \log(n))$ to $O(n)$ for a graph with n states). The essence of the method is the use of hashing without collision detection.

In this paper, we show that model checking can also benefit from a similar reduction in the required randomly accessed memory. This result is obtained by a combination of techniques. We approach model checking from the automata-theoretic perspective of [21, 22, 26]. This has the advantage of essentially reducing model checking to reachability analysis, though on a state space that is the cross product of the original state space with the state space of an automaton describing the functional specification. It is then possible to adapt techniques inspired by those of [9] to solve this problem. However, while Holtzmann's technique is suitable for searching for "bad" states in the state space, model checking involves searching for "bad" cycles. We thus had to develop some special purpose algorithms that are presented here.

This paper is organized as follows. We first review some background on model checking using the automata-theoretic approach and define the corresponding graph-theoretic problem. Then, we discuss the requirements that algorithms for solving this problem have to satisfy. Next we present our solutions. Finally, we present some extensions and some final remarks.

[1] actually, in the size of the global state-space of the program which can be exponential in the size of the program presented as a set of concurrent processes

2 Temporal Logic Verification using Büchi Automata

The model-checking problem we consider is the following[2]. Given a program P described as the concurrent execution of finite-state transition systems P_i and a temporal logic formula f, check that all infinite computations of P satisfy f. To solve this problem, we use the following steps:

1. Build the finite-automaton on infinite words for the *negation* of the formula f (one uses the negation of the formula as this yields a more efficient algorithm). The resulting automaton is $A_{\neg f}$. (Negating the formula f is simply done by prefixing it with the negation operator.)

2. Compute the global behavior of the program P. This is done by computing a form of product $\prod P_i$ of the transition systems P_i.

3. Take the product of the program P and of the automaton $A_{\neg f}$.

4. Check if this product automaton is non-empty.

The approach we have just outlined can be implemented in such a way that it has a major efficiency edge over other model-checking approaches. The idea is to compute the product P of the processes P_i and the product of P with $A_{\neg f}$ in the same step. This can lead to more efficiency for various reasons. In the first place, the product of P and $A_{\neg f}$ only accepts sequences that do not satisfy the requirement. One expects few of these (none if the program is correct). It is thus possible that the product of P and $A_{\neg f}$ will have fewer reachable states than P. Furthermore, when building $P \times A_{\neg f}$, it is not necessary to store the whole state-graph. It is sufficient to keep just enough information for checking that condition 4 above is satisfied. This is exactly what the algorithms we present in Section 3 will do. The advantages of reducing model checking to a reachability problem are also investigated in [12], but only for pure safety properties. In that case, it is sufficient to check that some states are simply reachable and the algorithms we develop in this paper are not needed.

To be able to describe our algorithms, we need more details about Büchi automata and how to check their emptiness. A *Büchi automaton* is a tuple $A = (\Sigma, S, \rho, s_0, F)$, where

- Σ is an alphabet,
- S is a set of states,
- $\rho : S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function,
- $s_0 \in S$ is an initial state, and
- $F \subseteq S$ is a set of designated states.

A *run* of A over an infinite word $w = a_1 a_2 \ldots$, is an infinite sequence $s_0, s_1, \ldots$, where s_0 is the initial state and $s_i \in \rho(s_{i-1}, a_i)$, for all $i \geq 1$. A run $s_0, s_1, \ldots$ is *accepting* if there is some designated state that repeats infinitely often, i.e., for some $s \in F$ there are infinitely many i's such that $s_i = s$. The infinite word w is *accepted* by A if there is an accepting run of A over w. The set of denumerable words accepted by A is denoted $L(A)$.

From the definition of Büchi automata, it is relatively easy to see that a Büchi automaton is nonempty iff it has some state $f \in F$ that is reachable from the initial state and reachable

[2] [10] is a detailed introduction to the verification of finite-state systems. It also describes how the algorithms we introduce in this paper can be used in an actual verification system.

from itself (in one or more steps) [23]. In graph-theoretic terms, this means that the graph representing the automaton has a reachable cycle that contains at least one state in F. In what follows, we will give a memory-efficient algorithm to solve this problem.

To formalize our verification approach, we consider processes P_i, $1 \leq i \leq n$, which are finite-state transition system consisting of[3]

- a state space V_i,
- a set of actions Σ_i
- a nondeterministic transition function $\sigma_i : V_i \times \Sigma_i \to 2^{V_i}$
- a starting state $v_{0i} \in V_i$.

The program P corresponding to the concurrent execution of the processes P_i is obtained by taking the product of the transition systems P_i. More specifically, the transition system $P = \prod P_i$ is $P = (V, \Sigma, \sigma, V_0)$ where

- $V = \prod V_i$, the state space is the cartesian product of the state spaces of the P_i's;
- $\Sigma = \bigcup \Sigma_i$, the set of actions is the union of the sets of actions of the components;
- σ is defined by $(v_1', \ldots, v_n') \in \sigma((v_1, \ldots v_n), a)$ iff
  - $v_i' \in \sigma(v_i, a)$ for each i such that $a \in \Sigma_i$ and
  - $v_i' = v_i$ for each i such that $a \notin \Sigma_i$, i.e. actions common to several processes are synchronized whereas other actions are interleaved;
- $v_0 = (v_{01}, \ldots, v_{0n})$.

Our goal is to verify properties of the infinite behaviors of P. These are defined by viewing P as a restricted type of Büchi automaton in which the set of designated states is the whole set of states V. Note that the verification procedure we are describing does not consider the finite behaviors of P. Properties of finite behaviors, like for instance absence of deadlock, have to be checked separately.

To check that P satisfies a temporal logic property f, we first need to build a Büchi automaton over the program alphabet Σ that accepts all infinite words that do not satisfy f. A description of temporal logic and of the construction of automata from temporal logic formulas can be found in [26] and in Chapter 4 of [20]. Let thus $A_{\neg f} = (\Sigma, S, \rho, s_0, F)$ be the Büchi automaton constructed from the negation of the formula f. The next step of the verification procedure is to compute $P \times A_{\neg f}$ which is the Büchi automaton with

- state set $V \times S$,
- transition function $\tau : V \times S \times \Sigma \to 2^{V \times S}$ defined by $(v_2, s_2) \in \tau((v_1, s_1), a)$ iff $v_2 \in \sigma(v_1, a)$ and $s_2 \in \rho(s_1, a)$,
- initial state (v_0, s_0),
- and set of designated states $V \times F$.

[3] What follows just specifies one possible description of the program P. Other definitions of processes and of the concurrent execution of processes are possible. The only essential property is that it be possible to compute a finite transition system describing P.

This product automaton accepts all runs which are possible behaviors of P (accepted by the automaton P) and violate the formula f (are accepted by the automaton $A_{\neg f}$). Hence we have reduced the problem of proving that the program P satisfies the formula f to the problem of checking the emptiness of the Büchi automaton $P \times A_{\neg f}$.

It is interesting to note that the product automaton $P \times A_{\neg f}$ has the Büchi type of acceptance condition because the acceptance condition for P is the trivial one. In the case in which the program P is modeled as an arbitrary Büchi automaton, the problem of checking the emptiness of $P \times A_{\neg f}$ is somewhat different and will be examined in Section 4.

3 Verification Algorithms

3.1 Requirements on the Algorithms

We characterize the memory requirements of any verification algorithm as follows. We consider the data structures used by the algorithm. The total amount of space used by these data structures corresponds to the total space requirements of the algorithm. The above space can be divided into memory that is *randomly accessed* and into memory that is *sequentially accessed*. For example, for implementing a hash table we need randomly accessed memory, while a stack can be implemented with sequentially accessed memory.

As correctly pointed out in [9], the bottleneck in the performance of most verification algorithms is directly related to the amount of the randomly accessed memory these algorithms require, and is due to the significant amount of paging involved during the execution of the algorithm. Holzmann observed that there is a tremendous speed-up for an algorithm implemented so that its randomly accessed memory requirements do not exceed the main memory available in the system (since sequentially accessed memory can be implemented in secondary storage).

The basic problem that Holzmann considered is how to perform reachability analysis by using the least amount of randomly accessed memory. For a graph with n states, his scheme involves a depth-first search in the graph, where the information about the states visited is stored in a bit-array of size m as follows. When a new state is generated, its name is hashed into an address in the array; if the bit of the corresponding location is on, then the algorithm considers that the above state has already been visited; if the bit is off then it sets the bit and adds the state on the stack used by the depth-first search. Since there is no collision detection, it follows that the above search is partial; there is always a possibility that a state will be missed.

The key assumption behind this method, see [9], is that in general one can choose the value of m large enough and construct a hash function so that the number of collisions becomes arbitrarily small. Furthermore, since the limiting factor in reachability analysis is usually the space required by the computation rather than the time required to do the computation, one could significantly reduce the probability of error by running the algorithm a few times with different hash functions. Indeed, Holzmann claims that, for most practical applications, choosing a hash table of size $m = O(n)$ (in fact $m \geq 100n$) together with appropriate hash functions is sufficient for the effect of collisions to become insignificant. Is this really so?

To answer this question, let us consider the memory requirements of the general reachability problem defined as follows. We assume that the states of the graph G have names from a name space U. In many applications (for example protocols), $|U|$ is many orders of magnitude larger than the number n of reachable states of G. In this case, complete reachability analysis (no missed states whatever the input graph) appears to require $O(n \log |U|)$ bits of randomly accessed memory, and probably cannot be done with less memory (unless the names of the reachable states of G are not randomly selected from U). Indeed, representing each state with less than $\log|U|$ bits amounts to mapping the state space U to a smaller state space. Now, for any such mapping there will always be subsets of U on which it is not one-to-one and hence on which complete reachability will not be guaranteed.

The situation is different is one analyses the problem from a probabilistic point of view. Consider all possible mappings from the set $S = \{1, \ldots, n\}$ into the set $\{1, \ldots, m\}$. There are m^n such mappings of which $m!/(m-n)!$ are one-to-one. Thus, if one assumes that the mapping implemented by a hash function is randomly selected, the probability that it is one-to-one (no collisions) is $m!/((m-n)!m^n)$, which for $n << m$ can be approximated by $e^{-n^2/m}$. This implies that in the case of a name space U and a graph with n reachable states, we can do partial reachability (with arbitrarily small probability of missing reachable states) by using $O(n \log n)$ bits of randomly accessed memory (instead of $O(n \log|U|)$ bits for complete reachability) as follows. First hash the n reachable states into a set $1, \ldots, m$ with an arbitrarily small probability of collision. As we have just seen, this is possible if we take $m = O(n^2)$. Then, do complete reachability using the set $1, \ldots, m$ as the name space for the states.

Holtzmann's technique goes one step further and only uses one bit of randomly accessed memory per reachable state[4]. This is equivalent to assuming that there exists a hash function mapping U into $1, \ldots, m$, $m = O(n)$, with a small probability of collisions. As the analysis above shows, this is not possible if we just assume that the hash function is random. It can however be possible if the state space U is only of size $O(n)$ or if the set of reachable states has a particular structure that can be used by the hash function. In these cases, the gain in randomly accessed memory, size $O(n)$ instead of size $O(n \log|U|)$ is quite significant for large state spaces.

However, this gain in memory use is only obtained for straightforward reachability analysis. To verify general temporal properties we have to check nonemptiness of the product automaton. One way to accomplish this is to construct the strongly connected component of the product automaton state graph and then to check whether one of the strongly connected component contains an accepting state. Unfortunately, we cannot apply Holtzmann's method to the standard algorithm for constructing the strongly connected components of the graph [2]. Indeed, although in that algorithm the states of the components are stored in a stack, it requires access to information (depth-first and low-link number) about states randomly placed in the stack, which implies the need of at least $O(n \log n)$ bits of randomly accessed memory. Hence, given a fixed amount of memory, the size of the problems we could efficiently analyze with the above algorithm is substantially smaller than the size of the problem that can be analyzed with the technique of [9].

From the previous discussion the following problem emerges. Assuming that reachability analysis in graphs of size n can be efficiently done with randomly accessed memory of size $O(n)$, can we solve the emptiness problem for Büchi automata using only randomly accessed memory of size $O(n)$? The answer to this problem is positive and the corresponding algorithms are described in the following section.

3.2 The Algorithms

In this section we provide algorithms for the following problem.

[4]Actually, [10] recommends 2 bits per state. This means using two hash tables, each containing one bit per state and addressed with its own hash function. A collision then only occurs if there is a collision in both hash tables. In this setting, the probability of no collision is approximately e^{-n^3/m^2}.

Problem 1 (nonemptiness of Büchi automata) *Given directed graph G, start node s_0, distinguished set of accepting nodes F, determine whether there is a member of F which is reachable from s_0 and belongs to a cycle, or equivalently, to a nontrivial strongly connected component.*

We make the following representation assumptions. The graph G is given by a *successor* function: a function that takes a node as argument and returns an ordered list of its successors. The set F is specified by a membership routine. We assume that we have a one-to-one function h mapping every node to an integer in the range $1, \ldots, m$.

Algorithm A:
The algorithm consists of two depth-first-searches (DFS's). The two searches can be performed one after the other, or can be done together in an interleaved fashion. It is simpler to describe first the noninterleaved execution. The purpose of the first DFS is to (1) determine the members of F that are reachable from s_0, and (2) order them according to last visit (i.e., in postorder) as $f_1, \ldots, f_k$.[5] The second DFS explores the graph using this ordering; it does not perform k searches but only one. In more detail, the main data structures are as follows: a stack S (to hold the path of DFS from root to current node), a (FIFO) queue Q to hold the reachable members of F in postorder and a bit-array M indexed by the hash values $1, \ldots, m$ for the "marked" bit (whether the node has been visited). The two passes share the same structures S and M.

The first DFS is as follows:

```
1. Initialize: S := [s0], M := 0, Q := ∅.
2. Loop: while S ≠ ∅ do
             begin
                  v := top(S);
                  if M[h(w)] = 1 for all w ∈ succ(v)
                    then begin
                          pop v from S;
                          if v ∈ F insert v into Q
                          end
                    else begin
                          let w be the first member of succ(v) with M[h(w)] = 0;
                          M[h(w)] := 1;
                          push w into S
                    end
             end
```

The second DFS is as follows:

```
1. Initialize: S := ∅, M := 0.
2. Loop: while Q ≠ ∅ do
         begin
              f := head(Q);
              remove f from Q;
              push f into S;
              while S ≠ ∅ do
                    begin
```

[5] f_1 is the first postorder reachable accepting state and f_k is the last

```
            v := top(S);
            if f ∈ succ(v) then halt and return "YES";
            if M[h(w)] = 1 for all w ∈ succ(v)
              then pop v from S
              else begin
                   let w be the first member of succ(v) with M[h(w)] = 0;
                   M[h(w)] := 1;
                   push w into S
              end
        end
end
```

The correctness of the algorithm is based on the following claims.

Lemma 1 *Let $f_1, \ldots, f_k$ be the members of Q after the first DFS, i.e., the members of F that are reachable from s_0 in postorder (f_1 is the first member of F to be reached in postorder, f_k the last). If for some pair f_i, f_j with $i < j$ there is a path from f_i to f_j, then node f_i belongs to a nontrivial strongly connected component.*

Proof: Suppose that there is a path from f_i to f_j. If no node on this path was marked before f_i, then the DFS would have reached f_j from f_i, so f_j would have come before f_i in the postorder. Thus, some node p on the path was marked before f_i. If p comes before f_i in the postorder, then f_j also should come before f_i in the postorder. Since p was marked before f_i, but comes after f_i in the postorder, it must be an ancestor of f_i. Thus, f_i can reach an ancestor and therefore belongs to a nontrivial strongly connected component. $\Box$

Theorem 1 *If the second DFS halts and returns "YES", then some reachable node of F belongs to a nontrivial strongly connected component. Conversely, suppose that some reachable node of F belongs to a nontrivial strongly connected component. Then the second DFS will return "YES".*

Proof: The first part is clear: suppose the second DFS returns "YES" while processing node f_j of Q. Then, it is building a tree with root f_j and discovers a back edge to the root f_j, and therefore f_j is obviously in a cycle. For the converse, let f_j be a reachable member of F that belongs to a nontrivial strongly connected component and has the smallest index j among all such members. Consider a path p from f_j to itself. We claim that no node of p is reachable from a f_i with a smaller i. For, if some node was reachable, then f_i would also reach f_j, which by Lemma 1 contradicts the choice of f_j. Therefore, no node of the path p is marked when we push f_j into S in the second DFS, and thus we will find a back edge to the root f_j. $\Box$

Note that the creation of both S and Q and access to them in both searches are sequential. Hence, both can be stored in secondary memory as needed.

So far we analyzed the algorithm under the assumption that the hash function f is perfect. One of the main features of our algorithm is its behavior in the presence of hash collisions. In that case, although the algorithm might erroneously conclude (due to collisions) that the Büchi automaton does not accept any word, it will never mistakenly conclude that the automaton accepts some word. In terms of the underlying verification problem, this means that our algorithm might miss some errors, but will never falsely claim that the protocol is incorrect.

Thus, the algorithm should be viewed more as a systematic debugging tool rather than as a verification tool.

The following alternative does away with the queue Q by using instead a second stack S_2 and bit array M_2. We do the second DFS on top of the first in an interleaved way. The details are as follows.

Algorithm B:

```
1. Initialize: S1 := [s0], S2 := ∅, M1 := M2 := 0.
2. while S1 ≠ ∅ do
        begin
            x := top(S1);
            if there is a y in succ(x) with M1[h(y)] = 0
            then begin
                    let y be the first such member of succ(x);
                    M1[h(y)] := 1;
                    push y into S1
                    end
            else begin
                    pop x from S1;
                    if x ∈ F then begin
                       push x into S2;
                       while S2 ≠ ∅ do begin
                               v := top(S2);
                               if x ∈ succ(v) then halt and return "YES";
                               if M2[h(w)] = 1 for all w ∈ succ(v)
                                  then pop v from S2
                                  else begin
                                         let w be the first member of succ(v) with M2[h(w)] = 0;
                                         M2[h(w)] := 1;
                                         push w into S2
                                  end
                       end
                    end
            end
        end
```

The above algorithm requires twice as much space as that required by the first algorithm. Its advantage is that if the automaton is found to be nonempty, an accepted word can be extracted from the stacks S_1 and S_2. In verification terms, this means that, if the protocol is found to be incorrect by the algorithm, a sample incorrect path can be produced. This is essential for debugging to be possible.

4 Extensions, Concluding Remarks and Related Work

An extension of the verification problem described in Section 2 is the verification of programs with liveness conditions, see [1]. In this case the program is given in terms of components, each having it own liveness conditions. Each such component is modeled as a Büchi automaton.

Hence, the product $P \times A_{\neg f}$ corresponds to an automaton whose transition table G is the product of the corresponding transition tables and its acceptance condition is given in terms of a set of sets of designated states $\{F_1, \ldots, F_k\}$. A run is accepting if it repeats some state from each of these sets infinitely often. Clearly, checking the emptiness of $P \times A_{\neg f}$ is equivalent with checking for the existence of a strongly connected component in the product transition table which is reachable from the initial state and intersects all these sets. Let S be the state space of the product transition table. We can construct a Büchi automaton B with $k|S|$ states, such that the emptiness of B is equivalent with the emptiness of $P \times A_{\neg f}$ (see for instance [22]).

- The graph of B consists of k copies of G with the transitions modified as follows. Consider the k copies $G_1, \ldots, G_k$ of G. For $i = 1, \ldots, k$, replace the transitions from every state $f \in F_i$ of G_i by similar transitions to the states in $G_{(i \bmod k)+1}$.
- The initial states of B are those of one copy of G, say G_1.
- The accepting states of B are the states F_i of the copy G_i of G, for some arbitrary i. For instance we can take $F_1 \subset G_1$.

Hence, if we apply the algorithms of the previous section to B, we can do verification with $O(k|S|)$ bits of randomly accessed memory.

Another remark is the following. In many applications it is reasonable to assume that the predecessor function of the graph is given as well. In this case one can use the algorithm in Section 6.7 in [3] for constructing the strongly connected components of the graph G by using randomly accessed memory of size $O(n)$. Let G_r be the directed graph corresponding to G by reversing its edges. This algorithm performs first a DFS on G and numbers the states in order of completion of the recursive calls (in postorder). This can be implemented by pushing the states in a stack according to their postorder visit by the DFS; this stack can use sequentially accessed memory. Then the algorithm performs a DFS on G_r (by using the predecessor function of G) starting with the state with the highest postorder sequence number (top of stack). This DFS on G_r must be restricted to the states reached during the first DFS, and uses a hashing mechanism for marking the states already visited. If the search does not reach all states, the algorithm starts the next DFS on G_r from the highest-numbered state which has not been already visited by the previous DFS. This can be easily done by poping the postorder stack until a state which has not been visited (the corresponding bit in the hash table is zero) is found. Since each tree in the resulting spanning forest is a strongly connected component, one can easily check for the properties of each such component while it is being generated.

Hashing without collision detection is but one technique that reduces the amount of randomly accessed memory needed for a state-space search. Another technique that has been investigated recently is "state-space caching" [11, 13, 6]. State-space caching is based on the observation that, when doing a depth-first search of a graph, storing only the states that are on the search stack is sufficient to guarantee that the search terminates. While this can produce a very substantial saving in the use of randomly accessed memory, it usually has a disastrous impact on the run time of the search. Indeed, each state will be visited as many times as there are simple paths reaching it. An improvement on this idea, is to store, not only the states that are on the search stack, but also a bounded number of other states (as many as will fit in the chosen "state-space cache"). If the state-space cache is full when a new state needs to be stored, random replacement of a state that is not currently on the search stack is used.

The advantage of state-space caching is that the amount of memory that is used can be reduced with a limited impact on the time required for the search. Indeed, if the cache is large enough to contain the whole state space, there is no change in the required time. If the size of the cache is reduced below this limit, the time required for the search will only increase gradually. Experimental results, however, show that, below a threshold that is usually between 1/2 and 1/3 of the size of the state space, the run time explodes, unless additional techniques are used to restrict the number of distinct paths that can reach a given state [6].

The behavior of state-space caching is quite the opposite of that of the technique we discuss in this paper. Indeed, state-space caching guarantees a correct result, but at the cost of a potentially large increase in the time needed for the state-space search. On the other hand, hashing without collision detection never increases the required run time, but can fail to explore the whole state space. Moreover, state-space caching cannot easily be used with the algorithms we have described for checking the emptiness of Büchi automata. Indeed, the effectiveness of these algorithms crucially depends on visiting states only once during the second search.

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] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading, 1974.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison Wesley, Reading, 1982.

[4] E. M. Clarke and O. Grümberg. Avoiding the state explosion problem in temporal logic model-checking algorithms. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pages 294–303, Vancouver, British Columbia, August 1987.

[5] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[6] P. Godefroid, G. J. Holzmann, and D. Pirottin. State space caching revisited. In *Proc. 4th Workshop on Computer Aided Verification*, Montreal, June 1992.

[7] R. Grotz, C. Jard, and C. Lassudrie. Attacking a complex distributed systems from different sides: an experience with complementary validation tools. In *Proc. 4th Work. Protocol Specification, Testing, and Verification*, pages 3–17. North-Holland, 1984.

[8] B.T. Hailpern. Tools for verifying network protocols. In K. Apt, editor, *Logic and Models of Concurrent Systems, NATO ISI Series*, pages 57–76. Springer-Verlag, 1985.

[9] G. Holzmann. An improved protocol reachability analysis technique. *Software Practice and Experience*, pages 137–161, February 1988.

[10] G. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall International Editions, 1991.

[11] G. J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(12):2413–2434, 1985.

[12] C. Jard and T. Jeron. On-line model-checking for finite linear temporal logic specifications. In *Automatic Verification Methods for Finite State Systems, Proc. Int. Workshop, Grenoble*, volume 407, pages 189–196, Grenoble, June 1989. Lecture Notes in Computer Science, Springer-Verlag.

[13] C. Jard and T. Jeron. Bounded-memory algorithms for verification on the fly. In *Proc. Workshop on Computer Aided Verification*, Aalborg, July 1991.

[14] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[15] M.T. Liu. Protocol engineering. *Advances in Computing*, 29:79–195, 1989.

[16] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. 5th Int'l Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.

[17] H. Rudin. Network protocols and tools to help produce them. *Annual Review of Computer Science*, 2:291–316, 1987.

[18] H. Rudin and C.H. West. A validation technique for tightly-coupled protocols. *IEEE Transactions on Computers*, C-312:630–636, 1982.

[19] C.A. Sunshine. Experience with automated protocol verification. In *Proceedings of the International Conference on Communication*, pages 1306–1310, 1983.

[20] André Thayse and et al. *From Modal Logic to Deductive Databases: Introducing a Logic Based Approach to Artificial Intelligence.* Wiley, 1989.

[21] M. Vardi. Unified verification theory. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Temporal Logic in Specification*, volume 398, pages 202–212. Lecture Notes in Computer Science, Springer-Verlag, 1989.

[22] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. Symp. on Logic in Computer Science*, pages 322–331, Cambridge, june 1986.

[23] M.Y. Vardi and P. Wolper. Reasoning about infinite computation paths. IBM Research Report RJ6209, 1988.

[24] C.H. West. Generalized technique for communication protocol validation. *IBM J. of Res. and Devel.*, 22:393–404, 1978.

[25] C.H. West and P. Zafiropulo. Automated validation of a communication protocol: the ccitt x.21 recommendation. *IBM Journal of Research and Development*, 22:60–71, 1978.

[26] P. Wolper. On the relation of programs and computations to models of temporal logic. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Proc. Temporal Logic in Specification*, volume 398, pages 75–123. Lecture Notes in Computer Science, Springer-Verlag, 1989.