# On the Verification of Temporal Properties

*Patrice Godefroid*
*University of Lieġe, Belgium*

*Gerard J. Holzmann*
*AT&T Bell Laboratories*
*Murray Hill, New Jersey 07974*

*ABSTRACT*

We present a new algorithm that can be used for solving the model−checking problem for linear−time temporal logic. This algorithm can be viewed as the combination of two existing algorithms plus a new state representation technique introduced in this paper. The new algorithm is simpler than the traditional algorithm of Tarjan to check for maximal strongly connected components in a directed graph which is the classical algorithm used for model−checking. It has the same time complexity as Tarjan's algorithm, but requires less memory. Our algorithm is also compatible with other important complexity management techniques, such as bit−state hashing and state space caching.

*Proc. IFIP, Symp. on Protocol Specification, Testing, and Verification. June 1993, Liege, Belgium.*

January 1, 1993

# On the Verification of Temporal Properties

*Patrice Godefroid*
*University of Lieġe, Belgium*

*Gerard J. Holzmann*
*AT&T Bell Laboratories*
*Murray Hill, New Jersey 07974*

## 1. INTRODUCTION

Techniques for verifying both safety and liveness properties of concurrent systems with on−the−fly search algorithms have been known for quite some time. The verification of safety properties relies only on a search for reachable ‘‘bad states.’’ The verification of liveness properties, however, requires also search for reachable ‘‘bad cycles.’’ Until recently, it was assumed that the validation of liveness properties required the implementation of an algorithm for the detection of strongly connected components in the reachability graph. Tarjan’s standard algorithm [AHU74] can perform this work at a cost that is linear in the size of the reachability graph, and is therefore implemented in almost all verification systems that support liveness properties.

The construction of strongly connected components, however, is not compatible with a range of validation techniques that can be used to reduce the memory requirements of on−the−fly verification algorithms. Examples of such techniques are state space caching techniques, as studied for instance in [GHP92], and the bit−state hashing technique from [Hol88], which can be found in most mainstream verification tools today.

In the last three years two new algorithms have been published that each solve one part of this problem. In [CVWY90] an algorithm was proposed that avoids the detection of strongly connected components by performing two nested depth−first searches. The algorithm can be used to check for cycles that pass through at least one state marked as ‘‘accepting state.’’ In [Hol91] this algorithm was implemented to facilitate the detection of so−called ‘‘acceptance cycles’’ in PROMELA validation models with the validator SPIN.

The second algorithm was first published in [Hol91] and similarly avoids the detection of strongly connected components by the definition of a two−state demon automaton, that controls two separate searches, the standard search, and a truncated search for cycles, as will be explained in more detail below. This second algorithm can be used to check for cycles that do not pass through any states marked as ‘‘progress states.’’ In the context of SPIN, this is used for a fast detection of so−called ‘‘non−progress cycles’’, which are the dual of the ‘‘acceptance cycles’’ from above.

Each of these two algorithms proves that it is often not necessary to construct maximal strongly connected components: it is almost always sufficient to show only that they exist, and to produce a single traversal of a strongly connected component to demonstrate its existence to a user. The cost of both algorithms in time and memory is still linear on the size of the reachability graph: it is never more than twice the cost of a standard reachability analysis for each algorithm.

In this paper we discuss two additional improvements. First, we show how the algorithm from [CVWY90] can be simplified by being combined with the one from [Hol91]. Next, we show that the resulting algorithm can be implemented with a much smaller space complexity than thought possible, thanks to a new state representation technique. Instead of a doubling of the memory requirements at worst, the new algorithm requires no more than two bits of overhead for each state stored. Since most protocols of practical significance require at least tens if not hundreds of bytes of memory per state stored, this does not alter the space requirements in a significant way.

We also show that, despite its simplicity, the algorithm described here can solve the model−checking problem, that is, it can be used to verify any temporal property, and, if required, it can do so under any fairness

assumption. This paper focuses on the verification of temporal logic formulae, specifically the detection of acceptance cycles in Büchi automata, though the algorithm we propose can be used also independently of model−checking, for the detection of acceptance cycles in general.

In Section 2 and 3 we give a formal framework for the verification of temporal properties with reachability analyses. We discuss the algorithms from [CVWY90] and [Hol91], and describe how they can be combined. Section 4 continues with a discussion of the new general storage technique we propose, which is named ''hybrid storage.'' Section 5 concludes the paper with a discussion of related work and the conclusions.

## 2. VERIFICATION OF TEMPORAL PROPERTIES

### 2.1. Representing Programs

Consider a program $P$ describing a system of several interacting concurrent processes $P_i$. Processes can communicate, for instance, via shared variables or via communication channels. We only assume, however, that program $P$ describes a *finite − state* system. In other words, we assume that it is possible to compute a *finite − state* automaton $A_P$, often called a ''labeled transition system,'' that represents the behavior of all processes $P_i$ combined. Formally, $A_P$ is a tuple $A_P = (\Sigma_P, S_P, \Delta_P, s_{0P})$, where $\Sigma_P$ is an alphabet, $S_P$ is a finite set of states, $\Delta_P \subseteq S_P \times \Sigma_P \times S_P$ is a transition relation, and $s_{0P} \in S_P$ is the initial state. $A_P$ can be computed by simulating all possible sequences of actions the system can perform from its initial state. $\Sigma_P$ is the set of actions that are present in the code of the program $P$, $S_P$ is the set of states that the system can reach from its initial state, and the transitions in $\Delta_P$ correspond to transitions between states that the system can perform while executing a single action.

A *computation* of the program $P$ is a sequence of states $\sigma = s_0, s_1, \cdots$ such that there exists transitions $(s_{i-1}, a_i, s_i) \in \Delta$, for all $i \geq 1$. Thus states in $\sigma$ are intermediate states reached during the execution of the sequence of actions $a_1 a_2 \cdots$ by the system from its initial state.

Initially: $t = 1, y_1 = F, y_2 = F$

$P_1$

$l_0$: execute (noncritical section)
$l_1$: $y_{1:=T}$
$l_2$: if $(y_{2=F})$ then go to $l_7$
$l_3$: if $(t = 1)$ then go to $l_2$
$l_4$: $y_{1:=F}$
$l_5$: loop until $(t = 1)$
$l_6$: go to $l_1$
$l_7$: $t := 2$ (critical section)
$l_8$: $y_{1:=F}$
$l_9$: go to $l_0$

$P_2$

$m_0$: execute (noncritical section)
$m_1$: $y_{2:=T}$
$m_2$: if $(y_{1=F})$ then go to $m_7$
$m_3$: if $(t = 2)$ then go to $m_2$
$m_4$: $y_{2:=F}$
$m_5$: loop until $(t = 2)$
$m_6$: go to $m_1$
$m_7$: $t := 1$ (critical section)
$m_8$: $y_{2:=F}$
$m_9$: go to $m_0$

Figure 1 – Dekker's algorithm

Consider the well−known Dekker algorithm [Dij68] for mutual exclusion, reproduced in Figure 1. There

are two parallel processes $P1$ and $P2$, a shared variable $t$, and two private boolean variables $y_1$ and $y_2$. Each private variable can be set only by the process owning it, but can be examined by both. The variable $y_1$ in $P_1$ ($y_2$ in $P_2$) is set to $T$ at $l_1$ to signal the intention of $P1$ to enter its critical section at $l_7$. Next $P_1$ tests at $l_2$ if $P_2$ is about to enter its critical section by checking if $y_2 = T$. If $y_2 = F$, $P_1$ proceeds immediately to its critical section. If $y_2 = T$, there is a conflict. The conflict is resolved by using the value of variable $t$. If $t = 2$, then $P1$ ''withdraws'' by setting $y_1$ to $F$, and waits until its turn comes ($t = 1$). If $t = 1$, it waits until $P_2$ ''withdraws'', and then enters its critical section at $l_7$. While in the critical section, it sets the variable $t$ to 2, to indicate that next time a potential conflict should be resolved in favor of $P_2$, and it sets $y_1$ to $F$ just before exiting the critical section. We assume that $P_1$ and $P_2$ are running asynchronously on different processors with different speeds, and that read and write instructions involving shared variables are executed as atomic operations. The automaton $A_P$ corresponding to this concurrent program has 101 reachable states and 202 transitions.

### 2.2. Specifying Temporal Properties

For representing temporal properties, we use linear−time propositional temporal logic [MP92]. Linear−time temporal logic can be used for specifying properties of infinite sequences of states. Propositions in the logic correspond to boolean conditions on variables and process states of the program. Formulas are constructed over propositions using the classical boolean connectives ($\neg$, $\vee$, ...) and the temporal operators $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next) and $\boldsymbol{U}$ (until), whose semantics is defined as usual [MP92]. Formulas are interpreted on *infinite* sequences $s_0 s_1 s_2 \cdots$ of states: given a particular infinite sequence of states, the formula is either satisfied or falsified by this sequence. Informally, one has:

- $\Box p$ holds in state $s_i$ if $p$ holds in $s_i$ and in all successor states of $s_i$ in the sequence on which the formula is interpreted;
- $\Diamond p$ holds in $s_i$ if $p$ holds in some successor state of $s_i$ or in $s_i$ itself;
- $\bigcirc p$ holds in $s_i$ if $p$ holds at the next state;
- $p \boldsymbol{U} q$ holds in $s_i$ if $p$ holds in $s_i$ and in all successor states of $s_i$ until the first state in which $q$ holds.

Consider the formula $\Box(p \supset \Diamond q)$. It expresses the property: ''every state where proposition $p$ holds coincides with or is followed by a state where proposition $q$ holds''. All infinite sequences of states that meet this requirement satisfy this formula.

### 2.3. Verification Problem

The verification problem we consider is the following. Given a concurrent program $P$ and a linear−time temporal logic formula $f$, check that all infinite computations of $P$ satisfy $f$. This is known as the *model−checking problem*.

To solve this problem, the only assumption we need to make is that, for each formula $f$ in linear temporal logic, it is possible to build a *Büchi automaton* $A_f$ that accepts exactly the infinite words satisfying the temporal formula $f$ [Buc62, WVS83]. Formally, a Büchi automaton is a tuple $A = (\Sigma, S, \Delta, s_0, F)$, where

- $\Sigma$ is an alphabet,
- $S$ is a set of states,
- $\Delta \subseteq S \times \Sigma \times S$ is a transition relation,
- $s_0 \in S$ is the starting state, and
- $F \subseteq S$ is a set of accepting states.

A Büchi automaton is thus a classical automaton as defined in Section 2.1, augmented with a set $F$ of accepting states. Büchi automata are used to define languages of ω−words, i.e., functions from the ordinal ω to the alphabet $\Sigma$. Intuitively, a word is *accepted* by a Büchi automaton if the automaton has an infinite execution that intersects set $F$ infinitely often.

Formally, we define a *run* $\sigma$ of $A$ over an ω−word $w = a_1 a_2 \cdots$ as an ω−sequence $\sigma = s_0, s_1, \cdots$, that is, a function from ω to $S$, where $(s_{i-1}, a_i, s_{i)} \in \Delta$, for all $i \geq 1$. A run $\sigma = s_0, s_1, \cdots$ is *accepting* if there is some state in $F$ that repeats infinitely often, i.e., for some state $x \in F$ there are infinitely many $i \in \omega$ such that $s_i = x$. The ω−word $w$ is *accepted* by $A$ if there is an accepting run of $A$ over the ω−word $w$.

The construction of the Büchi automaton $A_f$ from the formula $f$ is, in the worst case, exponential in the length of the formula [Wol89, Tha89]. In practice, however, most formulas are short, and the worst case behavior is rarely seen.

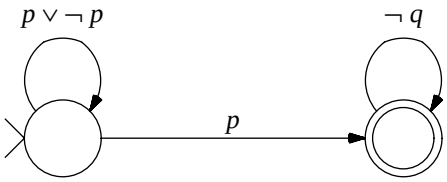


*Figure 2 – Büchi automaton corresponding to* $\neg(\Box(p \supset \Diamond q))$

Figure 2 shows a Büchi automaton that accepts all the infinite words satisfying the formula $\neg f$ with $f \equiv \Box(p \supset \Diamond q)$, i.e., all sequences of states which contain a state where $p$ holds and from which $q$ never holds in the remainder of the sequence.

The verification procedure is the following [WVS83,VW86b]. We first build the finite−automaton on infinite words for the *negation* of the formula *f*. The resulting automaton $A_{\neg f}$ accepts all sequences of states that violate *f*. (Of course, if $A_{\neg f}$ is provided by the user, the negation procedure can be skipped.) Then we compute the product automaton $A_G = A_P \times A_{\neg f}$ which is the Büchi automaton $A_G = (\Sigma, S, \Delta, s_0, F)$ with

- $\Sigma = \Sigma_{\neg f}$,
- $S = S_P \times S_{\neg f}$, $s_0 = (s_{0P}, s_{0\neg f})$,
- $((s,w),a,(u,v)) \in \Delta$ when $(s,t,u) \in \Delta_P$ and $(w,a,v) \in \Delta_{\neg f}$ (i.e., each transition $t$ of $A_P$ is synchronized with a transition $a$ of $A_{\neg f}$),
- $F = S_P \times F_{\neg f}$.

This product automaton accepts all infinite computations of $P$ that are accepted by $A_{\neg f}$, i.e., all computations that violate the formula *f*. The verification is completed by checking if automaton $A_G$ accepts any sequences. If $A_G$ is empty, we have proven that all infinite computations of $P$ satisfy formula *f*.

Consider Dekker's algorithm again. Let us verify that, if one of the two processes (say $P_1$) wants to enter its critical section, it eventually enters it. This property can be formalized with the formula $f \equiv \Box((at\ l_1) \supset \Diamond(at\ l_7))$. The Büchi automaton $A_{\neg f}$ corresponding to $\neg f$ is presented in Figure 2. Its initial state is designated by the symbol >. It has one accepting state designated by a double circle. Maybe surprisingly, the automaton $A_G$ corresponding to the product of Dekker's algorithm and $A_{\neg f}$ is nonempty, which means that there exists at least one infinite computation of the program violating the property. One such computation is the following: $P_1$ moves from $l_0$ to $l_1$ and next to $l_2$; then $P_2$ moves from $m_0$ up to $m_5$ and then loops for ever in $m_5$. This infinite computation violates the formula *f* given above since $l_1$ has been reached but $l_7$ is never reached.

Note that we verify properties of the infinite computations of $P$. These are defined by viewing $A_P$ as a restricted type of Büchi automaton in which the set of accepting states is the whole set of states in $A_P$. Thus this verification procedure does not consider *finite* computations of the program $P$. However, if required, it is always possible to transform finite computations into infinite ones by letting the terminating state repeat forever [LP85].

### 2.4. Specifying Fairness Assumptions

It is sometimes useful in verifications to formalize specific assumptions about the context in which a concurrent program is executed. If, for instance, concurrent processes are executed on different processors, it is customary to assume that each such processor will always make finite progress: if it has an enabled operation, it will eventually execute it. This ''finite progress'' assumption was already expressed in, for instance, Dijkstra's work in the late sixties [Dij68]. More recently, the classic finite progress assumption is usually defined as a special case of a larger class of ''fairness assumptions.'' In this context, finite progress is often defined as ''weak fairness'' [MP92,Fra86]. Other notions of fairness are used to formalize specific properties of, for instance, process schedulers for concurrent systems. The main purpose of these assumptions is to exclude computations that would not be allowed by the specific type of process scheduler that is

assumed. The fairness assumptions then act as filters, removing classes of infinite behaviors that conflict with the assumptions made about scheduler behavior.

Consider the previous example. We showed that the computation ''$P_1$ moves from $l_0$ to $l_1$ and next to $l_2$; then $P_2$ moves from $m_0$ up to $m_5$ and then loops for ever in $m_5$'' violates the property $\Box((at\ l_1) \supset \Diamond(at\ l_7))$. But, since we assumed that $P_1$ and $P_2$ are running asynchronously on different processors with different speeds, the above computation violates the finite progress (or weak fairness) assumption. Since $P_1$ and $P_2$ can always execute a transition (including looping), assuming weak fairness is in this case equivalent to assuming that every process always eventually executes a transition.

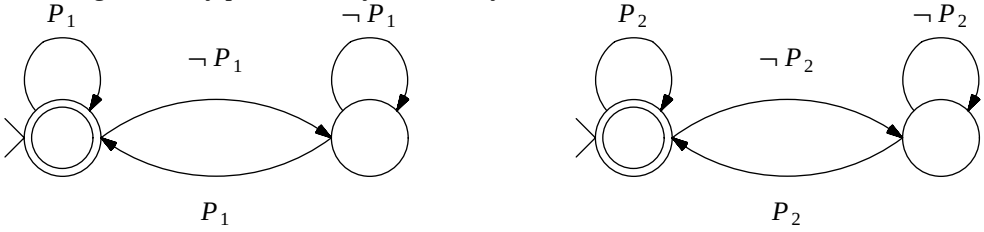


*Figure 3 – Büchi automata corresponding to $\Box \Diamond P_1$ and $\Box \Diamond P_2$ respectively*

It is beyond the scope of this paper to discuss the various notions of fairness that have been studied (see for instance [MP92,Fra86]). It can shown that fairness assumptions can be modeled by temporal logic formulas [LP85], or by Büchi automata [ACW90].

For the previous example, the assumption ''every process always eventually executes a transition'' can be formalized by the formula ''$\Box \Diamond P_1 \wedge \Box \Diamond P_2$'', where $P_i$ denotes the fact that the current transition of the program is performed by the process *i*. Another possibility is to add in the program two more processes corresponding to the two Büchi automata presented in Figure 3.

The verification procedure of a formula *f* remains very similar in the presence of fairness assumptions. If fairness assumptions are modeled by a formula *f′*, the verification problem amounts to checking that all infinite computations of the program *P* satisfy the formula $f' \supset f$, which can be done with the procedure discussed in the previous section. If fairness assumptions are modeled by several Büchi automata $A_{fair}$, each synchronized with the *program*,[1] the product $A_G = A_P \times A_{fair} \times A_{\neg f}$ is computed in a different way than defined in the previous section, since this time several automata have nontrivial acceptance conditions (e.g., see Chapter 4 of [Tha89]), but also here the verification problem reduces to checking the emptiness of $A_G$.

Consider Dekker's algorithm and formula $(\Box \Diamond P_1 \wedge \Box \Diamond P_2) \supset \Box((at\ l_1) \supset \Diamond(at\ l_7))$. The Büchi automaton $A_{\neg f}$ corresponding to this formula has 16 states and 92 transitions. The product automaton $A_G$ is now empty, which means that this formula is satisfied by the program.

Note that assuming fairness is often ill−advised in formal verifications. If the fairness ''filter'' is too restrictive, erroneous computations might be eliminated and thus missed during a verification. The result of the verification becomes conditional on the validity of the fairness assumptions. If, for instance, the scheduler on a system is changed, the proof of correctness of an application protocol that relied on the properties of that scheduler immediately becomes invalid. A verification that does not rely on fairness assumptions is therefore always stronger than one that does. In principle, furthermore, it is the obligation of the user to also prove formally that the fairness assumptions made are indeed valid for a given scheduler. In practice, this can be very hard, and it is often impossible.

We conclude, though, that the problem of proving that the program *P* satisfies the formula *f*, with or without assuming some notion of fairness, can be reduced to the problem of checking the emptiness of the Büchi automaton $A_G$. Note that computing $A_G$ and checking its emptiness can be done at the same time.

1. Another possibility is to specify acceptance sets for each process directly in the program, thus to define the program itself as being the product of Büchi automata [ACW90,GW91a].

## 3. CHECKING EMPTINESS OF BUCHI AUTOMATA

### 3.1. Previous Work

To check if the Büchi automaton $A_G$ is nonempty, one has to check if there exists a cycle in $A_G$ (viewed as a graph) that contains an accepting state and that is reachable from the initial state $s_0$. Note that it is not necessary to consider all possible cycles in $A_G$; it is sufficient to check if $A_G$ contains at lest one maximal (non−trivial) strongly connected component that is reachable from the initial state and that includes a state from set $F$.

Searching for maximal strongly connected components in $A_G$ can be done with the Tarjan algorithm [AHU74, Tar72]. This algorithm is based on a depth−first search in $A_G$ with additional computations at each state of $A_G$ that is encountered during the search. (See [AHU74] for a complete presentation of this algorithm.) The algorithm visits all $n$ reachable states of $A_G$ no more than once. Its time complexity is therefore still linear in the size of $A_G$. It requires the storage of all reachable states in a randomly accessed memory. Moreover, for each state the value of a variable **dfnumber**, which labels the reachable states in the order they are visited, must be stored as well. The Tarjan algorithm also requires the use of an additional stack.

Since this algorithm requires access to explicit state information, such as the value of **dfnumber**, to ensure its correctness, it is not compatible with techniques that do not guarantee the preservation of this information. The bit−state hashing technique, for instance, collapses the representation of states and their associated information into one or two bits of memory [Hol88]. State space caching techniques may even remove previously visited states completely from memory [GHP92]. Hence, given a fixed amount of memory, the size of the problems that can be analyzed with Tarjan's algorithm is unavoidably smaller than those that can be handled with these two techniques.

This observation triggered the development of an algorithm for checking Büchi automata emptiness [CVWY90] which is compatible with the bit−state hashing method and the state space caching discipline. In [CVWY90], checking emptiness of Büchi automata is reduced to a set of reachability problems. This is justified by the fact that a Büchi automaton is nonempty if and only if it has some state $x \in F$ that is both reachable from the initial state and reachable from itself.

The algorithm in [CVWY90] consists of two successive depth−first searches (DFS's). The purpose of the first DFS is to determine the accepting states of $F$ that are reachable from $s_0$. It orders them in postorder as $x_1, \cdots, x_k$, where $x_1$ is the reachable accepting state that was first backtracked during the search, and $x_k$ is the last such state. These accepting states are entered into a FIFO queue. The aim of the second DFS is to check if any of the accepting states in the queue is accessible from itself. The second DFS starts on $x_1$. If $x_1$ is reached during the search, a cycle that passes through $x_1$ has been detected and an error (i.e., a violation of the property being checked) is reported. A new DFS is then initiated from $x_2$, and so on until all $k$ accepting states have been checked. Due to the postorder ordering, it is possible to show that the states visited during the $i$th search cannot be revisited during the following $j$th searches, $i < j$. Consequently, the $k$ searches can be performed by using only one single hash table to store the states that have been visited. In other words, all searches from the $x_i \in F$ together correspond at most to one unique second DFS in $A_G$.

In the worst case, this algorithm visits all reachable states of $A_G$ twice: once in each phase of the search. Its time complexity is still linear in the size of $A_G$. It requires the storage of all $n$ reachable states in a randomly accessed memory. In case of error, the states in the stack of the second search correspond to a ''bad'' cycle through an accepting state $x_i$. However, a counter−example, with the complete error path starting from the initial state, can not be produced. It is then necessary to perform a third search to find a path starting from the initial state, leading to the state $x_i$.

In [CVWY90], a second version of this algorithm is also presented. This algorithm does away with the additional queue by using a second stack and a second hash table. The basic idea behind it is to perform the above two DFS's in an interleaved way, rather than sequentially. Each time an accepting state is ''backtracked'' in the first search, that search phase is suspended and a second search explores whether the accepting state is reachable from itself. If this is not the case, the algorithm resumes the first search to look for other accepting states. This version of the algorithm requires twice as much space than the first one. Its advantage is, however, that if an error is detected, a complete counter−example can immediately be

extracted from the two stacks.

### 3.2. Algorithm

In this section, we build upon the work of [CVWY90] and present an improved version of their second algorithm. The improved version does not require a second stack or a second hash table. To accomplish this, we use an algorithm presented in [Hol91] which solves a related problem: the detection of non−progress cycles. A non−progress cycle is a cycle that does not contain any states marked as ‘‘progress−states.’’ The algorithm from [Hol91] inspects two distinct state spaces, the regular one and a second one where transitions from progress states are disabled. It switches from one state space to the other by means of a two−state ‘‘demon’’ which is added to the system. The state of the demon process always determines in which state space the search currently operates. Below, we combine the ideas from [CVWY90] and [Hol91] to obtain a new algorithm for checking emptiness of Büchi automata.

Let us add a two−state *demon* process to the system being verified, as in [Hol91]:

Initially: $magic = 0$

**Demon**
$d_0$: $magic = 1$
$d_1$:

The state of this demon process defines in which ‘‘mode’’ the search operates. The initial state of the demon process is $d_0$, with variable *magic* equal to 0. The second, and final, state of the demon is $d_1$, immediately after the assignment, with *magic* equal to 1. We assume that the demon process can switch from its initial state to its final state only when the system is in an accepting state and only after all other enabled transitions have been explored. Once it has switched, the demon process can not go back.

The effect is that when *magic* is zero, a normal depth−first search is performed, corresponding to the first DFS of above. When *magic* is one, the second phase of the search is entered, with a check if an accepting state *x* is reachable from itself.

```
1   Initialize: Stack is empty; H is empty;
2   Search()
3   {   enter s0 in H;
4       push (s0) onto Stack;
5       DFS();
6   }
7   DFS()
8   {   s = top(Stack);
9       for all t enabled in s
10      {   s' = succ(s) after t; /* execution of t */
11          if s'.magic = 1 ∧ x = s' { halt and return ''Error'' }
12          if  s' is NOT in H
13          {   enter s' in H;
14              push (s') onto Stack;
15              DFS();
16          }
17      }
18      if  s.magic = 0 ∧ s ∈ F
19      {   s' = s with magic = 1; /* execution of Demon */
20          if  s' is NOT in H
21          {   x = s;
22              enter s' in H;
23              push (s') onto Stack;
24              DFS();
25          }
26      }
27      pop s from Stack
28  }
```

Figure 4 – ''Magic'' Search

A description of the new algorithm is given in Figure 4. It consists of a simple modification of a classical depth−first search. If the lines number 11 and from 18 to 26 are removed, the code of a classical DFS remains. One bit *magic* is added to the representation of each state of $A_G$ to store the current state of the demon process. $s.magic$ denotes the value of *magic* in state $s$. If $magic = 0$, the search is performed as usual. When an accepting state $s \in F$ is backtracked (line 18), then *magic* is set to 1 (line 19) and a second search is initiated (line 20) to determine if this accepting state, whose description is stored in an additional variable $x$ (line 21), is reachable from itself. If this is the case, this is detected in line 11 and an error is reported.

The correctness proof of the algorithm is straightforward.

**THEOREM** – If there exists a strongly connected component with at least one acceptance state, at least one cycle through this component will always be reported.

*Proof* – Consider a strongly connected component (SCC) with at least one accepting state. The first accepting state $S$ from this SCC entered into the $magic = 1$ part of the state space becomes the root of a new search subtree. If $S$ is reachable from itself, it is part of its own reachable subtree and will be detected on line 11. There is only one case where the path leading back to $S$ in this subtree can be truncated: when the subtree contains states previously visited, and present in $H$. Such intermediate states in $H$ must have been reached from another accepting state, and since they are both reachable from $S$ and $S$ is assumed to be reachable from them, they must be part of the same SCC as $S$. Therefore, $S$ could not have been the first accepting state of the SCC considered, which contradicts the assumption. □

The two successive or nested searches of the previous section are combined in this new algorithm, and they are performed using one single stack and one single hash table thanks to the demon process. The algorithm is quite straightforward to implement. Moreover, if an error is detected, the states in the current stack correspond to a complete infinite computation violating the property being checked and can be exhibited to the

user immediately as a counter−example. The time complexity remains unchanged: it is linear in the size of $A_G$. One additional bit corresponding to the value of *magic* has to be stored in the hash table attached to each reachable state, which slightly increases the memory requirements. In practice, this overhead is negligable due to the fact that the number of bits necessary to store one state is usually much larger than one bit (often hundreds or thousands of bits are necessary to represent each state).

In the worst case, however, the algorithm will still store all *n* reachable states of $A_G$ with the two different possible values of *magic*. This is twice as much as the number of states that needs to be stored with Tarjan's algorithm, or with the first version of the algorithm from [CVWY90] (it is the same as for the second version). It is possible to overcome also this problem by using a new *hybrid storage technique*, which will be introduced in the next section.

## 4. HYBRID STORAGE

### 4.1. Storage Techniques

During the search in $A_G$ performed by the new algorithm, all states visited are stored in memory. There are various ways in which these *n* reachable states of $A_G$ could be stored.

Assume the states of $A_G$ have names from a name space *U*. Its cardinality $|U|$ corresponds to the product of the number of all possible values for all individual process states, all local and global variables, and all message channel contents. Since $|U|$ is the number of possible names for a state, at least $\log|U|$ bits are necessary to represent each of these states unambiguously. Hence storing *n* reachable states requires at least $MEM_{classic} = n\log|U|$ bits of memory. This is the classical storage technique used in conventional reachability analysis algorithms, which we may call a *logarithmic storage* technique.

We now consider an alternative storage technique, which we will call a *linear storage* technique.

Define a one−to−one correspondence between the elements of the name space *U* and the elements of an array *A* of bits, whose size must therefore be at least $|U|$. Initially, all bits of *A* are set to 0. If the *i*th state in *U* is encountered during the exploration of $A_G$, the *i*th bit in *A* is set to 1. With this storage technique, the memory $MEM_{linear}$ required by the state−space exploration algorithm is $MEM_{linear} = |U|$ and is independent of the number *n* of reachable states.

For a particular *U*, one can determine the critical ''density'' $d_{crit}$ for which both storage methods require the same amount of memory by stating $MEM_{classic} = MEM_{linear}$ or $d_{crit}: \frac{n}{|U|} = \frac{1}{\log|U|}$ Consequently, if $n > |U|/\log|U|$ the linear storage discipline is preferable, else the logarithmic storage requires less memory. In other words, the linear storage technique is only suitable for ''high density'' state spaces. Most protocol state spaces are usually far below the critical density for which linear storage pays off. $|U|$ is typically many orders of magnitude larger than the number of reachable states *n* in $A_G$. Moreover, the name space $|U|$ is usually so large that the linear storage technique would require an astronomic amount of memory. Typically a few hundred bytes are required to store one state; thus $|U|$ can be much greater than $2^{1000}$ bits, much more than available on today's computers. The applicability of the linear storage discipline is therefore limited to special cases. The algorithm from Figure 4 offers just such a special case.

### 4.2. Hybrid Storage

We first discuss a storage technique, that is a combination of logarithmic and linear storage, and which is therefore called *hybrid storage*.

We assume each state of $A_G$ can be unambiguously identified by a pattern of precisely $\log U$ bits. We divide the representation of each state *s* into two parts $s_1$ and $s_2$ of length, respectively, $\log U_1$ and $\log U_2$. One has $\log U = log U_1 + \log U_2$. Each state *s* of $A_G$ corresponds thus to one unique pair $(s_1, s_2)$. Call $s_1$ the head of *s* and $s_2$ the tail of *s*. Next let us collect states that have the same head into *groups*. States in a same group have the same head and only differ by their tail. During the exploration of $A_G$, we now store groups of states in memory, rather than individual states as with the logarithmic storage technique. Each group consists of a head $s_1$ of length $\log U_1$ plus a bit−array of length $U_2$ to store the tails of the states of that group.

With this hybrid storage technique, groups of $(log U_1) + U_2$ bits are stored to memorize the states of $A_G$

that have already been visited, instead of states of $\log U$ bits (i.e. $\log U_1 + \log U_2$) with the logarithmic storage method. The overhead is $U_2 - \log U_2$ bits per group stored, but clearly the number of groups stored can be smaller then the number of individual states.

The overall amount of memory $MEM_{hybrid}$ required with this storage technique is $MEM_{hybrid} = (n-m)(logU_1 + U_2)$ where $m$ is the number of head matchings during the search, i.e., the number of states which have the same head as another state previously encountered during the search.

For a particular $U$, and a given partition $U_1 -- U_2$, one can determine the critical proportion $m_{crit}/n$ of head matchings from which the hybrid storage technique pays off by setting $MEM_{classic} = MEM_{hybrid}$. One obtains: $\frac{m_{crit}}{n} = \frac{(U_2 - \log U_2)}{(log|U| - \log U_2 + U_2)}$ If $m > m_{crit}$, hybrid storage requires less memory than logarithmic storage and is thus preferable.

Assume $\log|U| = 1000$ and $U_2 = 2$. Then $m_{crit}/n = 1/1001$. Therefore, if more than one state among 1001 states has the same head of length 999 as another state previously encountered during the search, then hybrid storage is preferable to the logarithmic storage.

### 4.3. Application to the Verification of Temporal Properties

Clearly, the hybrid storage method pays off when we can identify a part of the state description with an high density, and place it in linear storage. Consider the algorithm from Figure 4. Since $A_G$ is a product of automata, we can divide them into two groups $A_{G1}$ and $A_{G2}$ such that $A_G = A_{G1} \times A_{G2}$, and use hybrid storage with this partition. Automata that likely have the highest density of coverage can be grouped in $A_{G2}$, i.e., the ''tail'' using linear storage.

A good choice is then to take $A_{G1} = A_P \times A_{\neg f}$ and use the two−states demon process $A_{demon}$ as $A_{G2}$. Instead of representing the current state of the demon process by 1 bit *magic* with logarithmic storage, we now store $n$ groups of $(logU) + 2$ bits, with the last two extra bits representing the linear storage area, indicating if the state of $A_{G1}$ has been visited with either $magic = 0$, or with $magic = 1$, or with both values of *magic*.

Consider the example from the end of Section 4.2: assume that the system being checked by the algorithm of Figure 4 is such that a state of $A_P \times A_{\neg f}$ requires 125 bytes, 1000 bits, to be represented. The memory overhead due to the storage of the state of the demon process is limited to 2 bits per reachable state in $A_P \times A_{\neg f}$. For this example, if more than 1 in 1001 states is visited in both search modes (the expected case), the hybrid storage requires less memory than the logarithmic one. If $A_G$ itself is a strongle connected graph and contains at least one accepting state, all $n$ reachable states are guaranteed to be visited in both search modes. The overall memory requirement when using hybrid storage will then be $n \times (1000 + 2)$ bits, compared to $2 \times n(1000 + 1)$ bits for the classical, logarithmic storage technique.

Another possibility would be to include $A_{\neg f}$ in $A_{G2}$, using $A_{G1} = A_P$ and $A_{G2} = A_{\neg f} \times A_{demon}$. If additional automata $A_{fair}$ are used, they can also be included in $A_{G2}$.

Experimental results with this storage technique are presented in the Table below. $A_P$ is the state−graph corresponding to Dekker's algorithm, $A_{\neg f}$ is the Büchi automaton shown in Figure 2, and $A_{\neg(f' \supset f)}$ is the Büchi automaton that corresponds to the negation of the formula $(\Box \Diamond P_1 \wedge \Box \Diamond P_2) \supset \Box((\text{at } l_1) \supset \Diamond(at \;\; l_7))$. The Table compares logarithmic storage with hybrid storage applied to the algorithm from Figure 4. The number of stored states (resp. groups) is given by #states (#groups). The size of each state (resp. group) is given by $|state|(|group|)$, with *B* representing bytes, and *b* bits. The column titled *Memory* gives the total memory required by the algorithm to store all states reached during the verification of each property. To compute this total, all sizes were rounded up to integer numbers of bytes (20B+2b becomes 21B), before being multiplied by the number of states. The first row in the Table gives the memory requirements of a simple exploration of $A_P$ alone, i.e., without being combined with a property. The second row corresponds to the verification of the formula *f*. With logarithmic storage, two bits are added to the representation of each state of $A_P$: one bit to represent the state of $A_{\neg f}$ (which has two states; see Figure 2) and one (magic) bit to represent the state of the demon. The third row corresponds to the verification of the formula $\neg(f' \supset f)$. Since the automaton $A_{\neg(f' \supset f)}$ contains 16 states, 5 bits are added with logarithmic storage: 4 bits for $A_{\neg(f' \supset f)}$, plus one bit for the demon process.

| Partition | Logarithmic Storage | Hybrid Storage |
|---|---|---|

| $A_{G1}$ | $A_{G2}$ | #states | \|state\| | Memory | #groups | \|group\| | Memory |
|---|---|---|---|---|---|---|---|
| $A_P$ | | 101 | 20B | 2020B | 101 | 20 | 2020B |
| $A_P$ | $A_{\neg f} \times A_{demon}$ | 243 | 20B+2b | 5103B | 101 | 20B+4b | 2121B |
| $A_P$ | $A_{\neg(f' \supset f)} \times A_{demon}$ | 670 | 20B+5b | 14070B | 101 | 20B+32b | 2424B |

TABLE – Comparison between logarithmic and hybrid storage

The hybrid storage technique requires less memory for both formula.

Note that the hybrid storage method is quite different from, and orthogonal to, other techniques, such as the state compression scheme from [HGP92]. The techniques can easily be combined to increase the reduction still further.

## 5. COMPARISON WITH OTHER WORK AND CONCLUSIONS

We have presented a new algorithm for checking the emptiness of Büchi automata with the following features:

- The algorithm can solve the model−checking problem for linear−time temporal logic, i.e., it can be used for the verification of any temporal property under any fairness assumption.
- Using a hybrid storage technique, its memory requirements are close to that of a conventional state space exploration of the program alone, almost eliminating the memory overhead required to verify a temporal property.
- The algorithm is compatible with techniques that can be used to increase the scope of automated validations, such as bit−state hashing and state space caching techniques.
- The algorithm can generate complete counter−examples, in each case where the program can violate a temporal property.
- The algorithm has a simple proof of correctness, and can be implemented as a relatively minor modification of a standard depth−first search.

On each of these points, the new algorithm compares favorably with all previously known algorithms, including [Tar72], [LP85], [CVWY90], and [Hol91].

The algorithm from Figure 4 has been added to the SPIN protocol validation tool in mid 1992 (though as yet without the hybrid storage method), replacing an earlier algorithm based on [CVWY90]. Non−commercial users can obtain the SPIN system via anonymous ftp from research.att.com from the /netlib/spin directory.

The importance of the devlopment of efficient algorithms for the verification of temporal logic formulae will need no justification. As one example of the growing importance of this field, we can mention the recently completed pilot verification project at AT&T, which was named NewCoRe. One of the main goals of the NewCoRe project was to demonstrate the feasibility of formal verification based on temporal logic in an industrial environment. Over a period of two years (April 1990 to April 1992) a team of 4 people worked on the formal verification of the ISDN/ISUP code for the 5ESS ® Switch, in parallel with a ‘‘mainstream’’ team of 20 to 25 people that was developing a conventional design. The verification team modeled high level requirements into hundreds of temporal logic formulae, and performed a total of 10,000 formal validations (at a sustained rate of over 400 automated validations per month). The main tool used in these validations was a new version of the validation tool SDLVALID [HP89], extended with algorithms for proving liveness properties that are similar to the ones discussed in this paper. As a result of this effort, the NewCoRe team was able to trap and prevent hundreds of sometimes quite subtle high level design errors, clearly demonstrating that temporal logic verification is today not only feasible, even for fairly large−scale industrial applications, but also extremely effective.

**Acknowledgements**

The work of the first author was done in part while visiting AT&T Bell Laboratories.

## 6. REFERENCES

[ACW90] 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.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, 1974.

[Bü62] J.R. Büchi, 'On a decision method in restricted second order arithmetic,' In: *Proc. Intern. Congr. Logic, Method and Philos. Sci.*, 1960, pp. 1−12, Stanford, Stanford University Press, 1962.

[CVWY90] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, 'Memory efficient algorithms for the verification of temporal properties,' In: *Proc. 2nd Workshop on Computer Aided Verification*, LNCS 531, pp. 233−242, June 1990.

[Dij68] E.W. Dijkstra, 'Cooperating Sequential Processes,' In: *Programming Languages*, F. Genuys (Ed.), Academic Press, New York.

[Fra86] N. Francez, *Fairness*, Springer Verlag, 1986.

[GHP92] P. Godefroid, G.J. Holzmann, and D. Pirottin, 'State space caching revisited,' In: *Proc. 4th Workshop on Computer Aided Verification*, Montreal, June 1992.

[GW91] P. Godefroid, and P. Wolper, 'A partial approach to model checking,' In: *Proc. 6th IEEE Symp. on Logic in Computer Science*, pp. 406−415, Amsterdam, July 1991.

[HGP92] G.J. Holzmann, P. Godefroid, and D. Pirottin, 'Coverage preserving reduction strategies for reachability analysis,' In: *Proc. 12th Int. Symp. on Protocol Specification, Testing, and Verification*, Florida, June 1992.

[Hol88] G.J. Holzmann, 'An improved protocol reachability analysis technique,' *Software, Practice and Experience*, 18(2):137−161, 1988.

[Hol91] G.J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1992.

[HP89] G.J. Holzmann, and J. Patti, 'Validating SDL specifications: an experiment,' In: *Proc. 9th Int. Symp. on Protocol Specification, Testing, and Verification*, North−Holland Publ, June 1989.

[LP85] O. Lichtenstein, and A. Pnueli, 'Checking that finite state concurrent programs satisfy their linear specification,' In: *Proc. 12th ACM Symp. on Principles of Programming Languages*, pp. 97−107, New Orleans, Jan. 1985.

[MP92] Z. Manna, and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer Verlag, Berlin, 1992.

[Tar72] R. E. Tarjan, 'Depth first search and linear graph algorithms,' *SIAM J. Computing*, 1:2, pp. 146−160, 1972.

[Tha89] A. Thayse, et al., *From Modal Logic to Deductive Databases: Introducing a Logic Based Approach to Artificial Intelligence*, Wiley, 1989.

[VW86] M.Y. Vardi, and P. Wolper, 'An automata−theoretic approach to automatic program verification,' In: *Proc. Symp. on Logic in Computer Science*, pp. 322−331., Cambridge, June 1986.

[Wol89] P. Wolper, 'On the relation of programs and computations to models of temporal logic.' In: *Proc. Temporal Logic in Specification*, LNCS 398, pp. 75−123, 1989.

[WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla, 'Reasoning about infinite computation paths,' In: *Proc. 24th IEEE Symp. on Foundations of Computer Science*, pp. 185−194, Tucson, 1983.