On the Verification of Temporal Properties

Patrice Godefroid\textsuperscript{a} and Gerard J. Holzmann\textsuperscript{b}

\textsuperscript{a}Université de Liège, Institut Montefiore B28, 4000 Liège Sart-Tilman, Belgium

\textsuperscript{b}AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.

Abstract

We study and compare various algorithms that can be used for solving the model-checking problem for linear-time temporal logic. We then propose a new algorithm that 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 in 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.

Keyword Codes: D.1.3; D.2.4
Keywords: Concurrent Programming, Program Verification

1. Introduction

Techniques for verifying both safety and liveness properties 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 [1] 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 bit-state hashing techniques [2], 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 [3] an algorithm is 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 [4] this algo-
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 [4] 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 proved that in many cases it is simply not necessary to construct complete 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 that demonstrates its existence to a user. The cost of both algorithms in time and memory is still strictly 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 [3] can be simplified by being combined with the one from [4]. 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. We show that instead of a doubling of the memory requirements at worst, the algorithm requires no more than two extra bits of memory 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.

Despite its simplicity, the algorithm described here can completely 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 formulas, 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 first give a formal framework for the verification of temporal properties with reachability analyses. We discuss the algorithms from [3] and [4], 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

We assume we are given a program $P$ describing a system composed of several interacting concurrent processes $P_i$. Processes can communicate with each other in some way, for instance via shared variables or communication channels. The only assumption about the program $P$ is that it describes a finite-state system. In other words, we assume that it is possible to compute a finite-state automaton $A_P$ (also often called a “labeled transition system”) representing the joint behavior of all processes $P_i$. 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
Initially: $t = 1$, $y_1 = F$, $y_2 = F$

<table>
<thead>
<tr>
<th>$P_1$</th>
<th>$P_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_0$: execute (noncritical section)</td>
<td>$m_0$: execute (noncritical section)</td>
</tr>
<tr>
<td>$l_1$: $y_1 := T$</td>
<td>$m_1$: $y_2 := T$</td>
</tr>
<tr>
<td>$l_2$: if ($y_2 = F$) then go to $l_7$</td>
<td>$m_2$: if ($y_1 = F$) then go to $m_7$</td>
</tr>
<tr>
<td>$l_3$: if ($t = 1$) then go to $l_2$</td>
<td>$m_3$: if ($t = 2$) then go to $m_2$</td>
</tr>
<tr>
<td>$l_4$: $y_1 := F$</td>
<td>$m_4$: $y_2 := F$</td>
</tr>
<tr>
<td>$l_5$: loop until ($t = 1$)</td>
<td>$m_5$: loop until ($t = 2$)</td>
</tr>
<tr>
<td>$l_6$: go to $l_1$</td>
<td>$m_6$: go to $m_1$</td>
</tr>
<tr>
<td>$l_7$: $t := 2$ (critical section)</td>
<td>$m_7$: $t := 1$ (critical section)</td>
</tr>
<tr>
<td>$l_8$: $y_1 := F$</td>
<td>$m_8$: $y_2 := F$</td>
</tr>
<tr>
<td>$l_9$: go to $l_0$</td>
<td>$m_9$: go to $m_0$</td>
</tr>
</tbody>
</table>

Figure 1. Dekker’s algorithm

$s_{0P} \in S_P$ 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 $\Delta_P$ corresponds 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 $\sigma = s_0, s_1, \ldots$ 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 $a_1 a_2 \ldots$ by the system from its initial state.

Example 2.1 Consider the well-known Dekker algorithm [5] for mutual exclusion, reproduced in Figure 1. There are two parallel processes $P_1$ and $P_2$, 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 may 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 $P_1$ to enter its critical section at $l_7$. Next $P_1$ tests at $l_2$ if $P_2$ has any interest in entering its own critical section by checking if $y_2 = T$. If $y_2 = T$, $P_1$ proceeds immediately to its critical section. If $y_2 = T$, there is a conflict. This conflict is resolved by using the value of variable $t$. If $t = 2$, then $P_1$ “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 [6]. 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
Formulas are constructed over propositions using the classical boolean connectives (¬, ∨, ...) and the temporal operators □ (always), ◯ (eventually), ○ (next) and U (until), whose semantics is defined as usual [6]. Formulas are interpreted on infinite sequences s_0s_1s_2... of states: given a particular infinite sequence of states, the formula is either satisfied or falsified by this sequence. Informally, one has:

- □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;
- ◯p holds in s_i if p holds in some successor state of s_i or in s_i itself;
- ○p holds in s_i if p holds at the next state;
- p U q holds in s_i if ◯q holds in s_i and if p holds in s_i and in all successor states of s_i until the first state in which q holds.

Example 2.2 Consider the formula □(p □ ◯ 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 are said to “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 fact we need about linear-time temporal logic is that, for each formula f, it is possible to build a Büchi automaton A_f that accepts exactly the infinite words satisfying the temporal formula f [7]. Formally, a Büchi automaton [8] is a tuple A = (Σ, S, Δ, s_0, F), where

- Σ is an alphabet,
- S is a set of states,
- Δ ⊆ S × Σ × S is a transition relation,
- s_0 ∈ S is the starting state, and
- F ⊆ S is a set of accepting states.

A Büchi automaton is thus an 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 Σ. 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 computation σ of A over an ω-word w = a_1a_2... as an ω-sequence σ = s_0s_1s_2... (i.e., a function from ω to S) where (s_{i-1}, a_i, s_i) ∈ Δ, for all i ≥ 1. A computation σ = s_0s_1s_2... is accepting if there is some state in F that repeats infinitely often, i.e., for some state x ∈ F there are infinitely many i ∈ ω such that s_i = x. The ω-word w is accepted by A if there is an accepting computation of A over w.

A construction of a Büchi automaton A_f from a formula f can be found in [9] and in chapter 4 of [10]. This construction is exponential in the length of the formula, but this is usually not a problem since the formulas to be checked are quite short and since the algorithm often behaves much better than its upper bound.
Figure 2. Büchi automaton corresponding to \( \neg(\Box(p \supset \Diamond q)) \)

**Example 2.3** Figure 2 shows a Büchi automaton that accepts exactly all the infinite words satisfying the formula \( \neg f \) with \( f \equiv \Box(p \supset \Diamond q) \), i.e., all sequences of states that contain a state where \( p \) holds and from which \( q \) never holds in the remainder of the sequence. Its initial state is designated by the symbol \( > \). It has one accepting state designated by a double circle.

The verification procedure is the following [7, 11]. We first build a finite-automaton on infinite words for the negation of the formula \( f \). The resulting automaton \( A_{\neg f} = (\Sigma_{\neg f}, S_{\neg f}, \Delta_{\neg f}, s_{0\neg f}, F_{\neg f}) \) accepts all sequences of states that violate \( f \). (Of course, if \( A_{\neg f} \) is provided by the user, the above construction can be skipped.) Then we compute the product automaton \( A_G = A_P \times A_{\neg f} \) which is the Büchi automaton \( A_G = (\Sigma, S, \Delta, s_0, F) \) defined by

- \( \Sigma = \Sigma_{\neg f} \),
- \( S = S_P \times S_{\neg f}, s_0 = (s_{0P}, s_{0\neg f}) \),
- \( ((s, w), a, (u, v)) \in \Delta \text{ when } (s, t, u) \in \Delta_P, (w, a, v) \in \Delta_{\neg f} \text{ and } a \text{ evaluates to true in state } s \text{ of } A_P \),
- \( F = S_P \times F_{\neg f} \).

This product automaton accepts all infinite computations of \( P \) that are accepted by the automaton \( A_{\neg f} \), i.e., all computations that violate the formula \( f \). Finally, we check if the automaton \( A_G \) is empty, i.e., if it accepts any sequences. If \( A_G \) is empty, we have proven that all infinite computations of \( P \) satisfy the formula \( f \).

**Example 2.4** 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 \( f \equiv \Box((at \ l_1) \supset \Diamond(at \ l_7)) \). A Büchi automaton \( A_{\neg f} \) corresponding to \( \neg f \) is presented in Figure 2, where \( p \) represents at \( l_1 \) and \( q \) represents at \( l_7 \). 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 forever 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
Figure 3. Büchi automata corresponding to $\square \Diamond P_1$ and $\square \Diamond P_2$ respectively

set of states in $A_P$. Thus the verification procedure we are describing does not consider finite computations of the program $P$. However, if one wants to take stopping computations (i.e., computations that leads to a terminating state where all processes of $P$ are blocked) into account during the verification, it is possible to transform these finite computations into infinite ones by letting the terminating state repeat forever [12].

2.4. Specifying Fairness Assumptions

It is sometimes useful in verification to take into account 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 process 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 [5]. More recently, the classic finite progress assumption is usually defined as a special case of a larger class of so-called “fairness assumptions”. In this context finite progress is similar to “weak fairness” [6, 13]. 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 certain classes of infinite behaviors that conflict with the assumptions made about scheduler behavior.

**Example 2.5** 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 $m_5$” was violating the property $\square((at \ l_1) \sqcup (at \ l_7))$. But, since we assumed that $P_1$ and $P_2$ are running asynchronously on different processors with different speeds, the above computation, where $P_2$ loops for ever in $m_5$ while $P_1$ never executes its next transition from $l_2$, does not correspond to an “actual” one. Indeed, in practice, $P_1$ would eventually execute its next transition since it is running on another processor than $P_2$. In order to only consider fair computations, we can assume that a process which is not blocked will eventually execute a transition. Since $P_1$ and $P_2$ can always execute a transition (including looping), this is equivalent to assuming that every process always eventually executes a transition.

It is beyond the scope of this paper to discuss the various notions of fairness that have been studied (see for instance [6, 13]). It has been shown that fairness assumptions can be modeled
by temporal logic formulas [12], or by Büchi automata [14].

**Example 2.6** For the previous example, the assumption “every process always eventually executes a transition” can be formalized by the formula $\Box \Diamond P_1 \land \Box \Diamond P_2$, where $P_i$ denotes the fact that the next transition of $A_P$ that is taken in the computation is performed by process $i$ ($P_i$ is a special kind of “state formula”; see [6]). Another possibility is to add in the program two more processes corresponding to the two Büchi automata presented in Figure 3.

The verification procedure for a formula $f$ remains very similar in 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 as presented in the previous section. If fairness assumptions are modeled by Büchi automata $A_{fair}$ that are synchronized with the program, the product $A_G = A_P \times A_{fair} \times A_{-f}$ is computed in a different way as defined in the previous section, since this time several automata have nontrivial acceptance conditions (e.g., see chapter 4 of [10]), but the verification problem is reduced again to checking the emptiness of $A_G$.

**Example 2.7** Consider Dekker's algorithm again and the formula $\forall(\Box \Diamond (P_1 \land \Box \Diamond P_2) \supset \Box((at l_1) \lor (at l_7)))$. The Büchi automaton $A_{-f}$ corresponding to this formula obtained from the construction given in [9, 10] has 16 states and 92 transitions. The product automaton $A_G$ is empty, which means that the formula is satisfied by the program under the considered fairness assumptions.

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 the 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.

### 3. Checking Emptiness of Büchi 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 least one maximal (nontrivial) strongly connected component that is reachable from the initial state and that includes a state from set $F$.

Another similar possibility is to directly specify acceptance sets for the processes in the program, thus to define the program as being a product of Büchi automata [14, 15].
Searching for maximal strongly connected components in $A_G$ can be done with the Tarjan algorithm [16, 1]. 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 [1] for a complete presentation of this algorithm.) The algorithm visits all $n$ reachable states of $A_G$ once, its time complexity is linear in the size of $A_G$. It requires the storage of all reachable states in a randomly accessed memory. Moreover, with each state, the value of a variable “DFNUM- BER”, 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 “DFNUM- BER”, to ensure its correctness, it is not compatible with techniques that do not guarantee the preservation of this information, as the bit-state hashing techniques, for instance, which collapse the representation of states into a single bit of memory [2]. Hence, given a fixed amount of memory, the size of the problems that can be analyzed with Tarjan’s algorithm is unavoidably smaller than the size of the problems that can be handled with bit-state hashing techniques.

This observation triggered the development of an algorithm for checking Büchi automata emptiness [3] which is compatible with the bit-state hashing method. In [3], 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 it has some state $x \in F$ that is reachable from the initial state and reachable from itself.

The algorithm in [3] 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 $s_0$, and to order them according to last visit (i.e., in postorder) as $x_1, \ldots, x_k$ ($x_1$ is the first backtracked reachable accepting state 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. Else another search 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 need not to 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 DFS. 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 corresponds 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 and leading to the state $x_i$.

In [3], a second version of this algorithm is also presented. This algorithm does away with the additional queue by using instead 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, the first DFS is suspended and the second DFS 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, etc. This second version requires twice as much space as 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 [3] and present another version of this algorithm. Our version does not require a second stack and a second hash table. The basic idea behind the simplification is to use an algorithm presented in [4] which solves a different though 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 [4] 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 [3] and [4] 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 [4]:

\[
\begin{align*}
\text{magic} = 0 & \quad \text{magic} = 1 \\
\rightarrow & \quad \rightarrow \\
\text{d}_0 & \quad \text{d}_1
\end{align*}
\]

Figure 4. Demon machine

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 \(\text{magic}\) equal to 0. The second, and final, state of the demon is \(d_1\) with \(\text{magic}\) 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 \(\text{magic}\) is zero, a normal depth-first search is performed, corresponding to the first DFS of above. When \(\text{magic}\) is one, the second phase of the search is entered, with a check if an accepting state \(x\) is reachable from itself.

A description of the new algorithm is given in Figure 5. It consists of a simple modification of a classical depth-first search. If the lines number 11 and from 18 to 25 are removed, the code of a classical DFS remains. One bit \(\text{magic}\) is added to the representation of each state of \(A_G\) to store the current state of the demon process. \(s.\text{magic}\) denotes the value of \(\text{magic}\) in state \(s\). Initially, \(s_0.\text{magic} = 0\) and the search is performed as usual (the value of \(\text{magic}\) remains 0). When an accepting state \(s \in F\) is backtracked (line 18), then the transition of the demon process becomes enabled and is executed: \(\text{magic}\) is set to 1 (line 19) and a second search is initiated (line 20) to determine if the 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 similar to the ones presented in [3] and [4].
Initialize: Stack is empty; $H$ is empty; $s_0.magic = 0$; $x = none$;

Search() {
  enter $s_0$ in $H$;
  push $(s_0)$ onto Stack;
  DFS();
}

DFS() {
  $s = \text{top}(\text{Stack})$;
  for all $t$ enabled in $s$ do {
    $s' = \text{succ}(s)$ after $t$; /* execution of $t$ */
    if $s'.magic = 1 \land x = s'$ then halt and return “Error”;
    if $s'$ is NOT already in $H$ then {
      enter $s'$ in $H$;
      push $(s')$ onto Stack;
      DFS();
    }
  }
  if $s.magic = 0 \land s \in F$ then {
    $s' = s$ with magic = 1; /* execution of Demon */
    if $s'$ is NOT already in $H$ then {
      $x = s$;
      enter $s'$ in $H$;
      push $(s')$ onto Stack;
      DFS();
    }
  }
  pop $s$ from Stack
}

Figure 5. “Magic” Search

**Theorem 3.1** If there exists in $A_G$ at least one strongly connected component containing at least one accepting state, the above algorithm will report an error.

**Proof:** Consider the first accepting state $x$ of a strongly connected component (SCC) that is entered into the $\text{magic} = 1$ part of the state space. This state $x$ becomes the root of a new search subtree. Since $x$ is reachable from itself, it is part of its own reachable subtree and will be detected on line 11. Indeed, paths leading from $x$ back to $x$ in this subtree can not be truncated because all intermediate states along these paths have not been visited yet, and are thus not present in $H$. Let us prove this by contradiction. Assume there is such an intermediate state $s$ that has been visited with $\text{magic} = 1$ before $x$. $s$ must have been reached from another accepting state $x'$, that has been backtracke before $x$ in the first DFS. Since $x$ and $s$ belong to the same SCC and since $s$ is reachable from $x'$, $x$ is also reachable from $x'$. Since $x$ is backtracke after $x'$ in the first DFS though $x$ is reachable from $x'$, this implies that all paths from $x'$ to $x$ (we know there exists at least one) intersect the stack with which state $x'$ is reached during the first DFS. Since there is a path from $x'$ to a state of the stack, i.e., closing a cycle on the stack, there is a path from $x'$ to itself, and thus $x'$ is an accepting state of a SCC. This contradicts the assumption that
$x$ is the very first such state. ■

The two successive or nested searches of the previous section are now combined and 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 with each reachable state, which slightly increases the memory requirements. In practice, this overhead is negligible 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 uniquely 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 [3] (it is the same as for the second version). In the next section, we show how it is possible to overcome this problem by using a new technique for representing states, called the hybrid storage technique.

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. The exploration is performed “on-the-fly”, i.e., without storing the transitions that are taken. In this section, we study various techniques for storing the $n$ reachable states of $A_G$.

Assume the states of $A_G$ have names from a name space $U$. $|U|$ corresponds to the product of the number of all possible values for all individual process states, all local and global variables, and all message channels 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

$$MEM_{\text{classic}} = n \log |U|$$

bits of memory. This is the classical storage technique used in conventional reachability analysis algorithms.

We now consider an alternative to this 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 $MEM_{\text{linear}}$ required by the state space exploration algorithm is

$$MEM_{\text{linear}} = |U|$$

and is independent of the number $n$ of reachable states.

For a particular $U$, one can determine the critical “density” $d_{\text{crit}}$ for which both storage methods require the same amount of memory by stating $MEM_{\text{classic}} = MEM_{\text{linear}}$:

$$d_{\text{crit}} = \frac{n}{|U|} = \frac{1}{\log |U|}$$
Consequently, if $n > |U|/\log |U|$, the linear storage discipline is preferable, else a classical 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. Indeed, $|U|$ is typically many order of magnitude larger than the number $n$ of reachable states 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 necessary 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 remains therefore very limited.

4.2. Hybrid Storage

We now discuss a storage technique that is a combination of classical 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 packets. States in a same packet have the same head and only differ by their tail. During the exploration of $A_G$, let us store packets of states in memory, rather than states as with the classical storage technique. Each packet consists of the head $s_1$ of length $\log |U_1|$, which characterizes it, plus a bit-array of length $|U_2|$ to store the tails of the states of that packet that have already been visited during the search.

With this hybrid storage technique, packets 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 classical storage method. The overhead is $|U_2| - \log |U_2|$ bits per packet stored. On the other hand, the number of packets that have to be stored during the search can be smaller than the number of states. Indeed, two different states of $A_G$ that have the same head $s_1$, and thus that only differ by their tail $s_2$ and $s'_2$ respectively, are represented in only one single packet: the first $\log |U_1|$ bits of the packet contain the name of $s_1$ while the bit corresponding to $s_2$ and the bit corresponding to $s'_2$ amongst the $|U_2|$ last bits will be set to 1.

The overall amount of memory $MEM_{\text{hybrid}}$ required with this storage technique is

$$MEM_{\text{hybrid}} = (n - m)(\log |U_1| + |U_2|)$$

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 $m_{\text{crit}}/n$ of head matchings from which the hybrid storage technique pays off by stating $MEM_{\text{classic}} = MEM_{\text{hybrid}}$. One obtains:

$$\frac{m_{\text{crit}}}{n} = \frac{(U_2 - \log U_2)}{(\log |U| - \log U_2 + U_2)}$$

If $m > m_{\text{crit}}$, then hybrid storage requires less memory than the classical storage and is thus preferable. Clearly, the hybrid storage method is useful when the number $m$ of head matchings is important.

Example 4.1 Assume $\log |U| = 1000$ and $|U_2| = 2$. Then $m_{\text{crit}}/n = 1/1001$. 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 classical storage.

4.3. Application to the Verification of Temporal Properties

The application of the hybrid storage method we have in mind is the storage of states of \( A_G \) explored by the algorithm that has been presented in Section 3.2. 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 first possibility is to take \( A_{G1} = A_P \times A_{\neg f} \) and use the two-states demon process \( A_{\text{demon}} \) as \( A_{G2} \). Then, instead of representing the current state of the demon process by 1 bit \( \text{magic} \) with the classical storage and taking the risk of storing \( 2n \) states of \( (\log |U|) + 1 \) bits, it is possible to store only \( n \) packets of \( (\log U) + 2 \) bits, the two last bits indicating if the state of \( A_{G1} \) has been visited with \( \text{magic} = 0 \), with \( \text{magic} = 1 \), or with both values of \( \text{magic} \).

Example 4.2 Consider the previous example again: assume that the system being checked by the algorithm of Section 3.2 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 classical one. If all \( n \) reachable states are visited in both search modes (which will happen, for instance, if \( A_G \) corresponds to a strongly connected graph and contains at least one accepting state), then the overall memory requirement when using hybrid storage is \( n(1000 + 2) \) bits instead of \( 2n(1000 + 1) \) bits when using classical storage.

Another possibility is to include \( A_{\neg f} \) in \( A_{G2} \) with \( A_{\text{demon}} \). Thus one has \( A_{G1} = A_P \) and \( A_{G2} = A_{\neg f} \times A_{\text{demon}} \). If some additional automata \( A_{\text{fair}} \) are used, they can also be included in \( A_{G2} \).

We have experimented these possibilities with the examples presented in Section 2. Let \( A_P \) be the state-graph corresponding to Dekker’s algorithm, let \( A_{\neg f} \) be the Büchi automaton shown in Figure 2 and \( A_{\neg(f \rightarrow f)} \) be the Büchi automaton corresponding to the negation of the formula “\( (\Box R_1 \land \Box R_2) \supset (\Box (at h_1) \land \Box (at h_2)) \)”. Table 1 compares the memory requirements of the algorithm presented in Section 3.2 when it is used with a classical storage and when it is used with the hybrid storage technique. \# states (\# packets) is the number of stored states (resp. packets) during the search, while \(|\text{state}||\text{packet}||\) is the size of one state (resp. packet) given in bytes (B) and bits (b). Memory is the memory required by the algorithm to store the states reached during the verification of the corresponding property. Sizes \(|\ldots||\) have been rounded to integer numbers of bytes (20B+2b becomes 21B), as an actual implementation requires, before being multiplied by \#... to obtain the value of Memory. The first row 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 classical storage, two bits have to be 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. In the third case, since \( A_{\neg(f \rightarrow f)} \) contains 16 states, 4 bits plus one for the demon are added with classical storage. One clearly sees in Table 1 that the hybrid storage technique requires less memory than the classical storage technique.
<table>
<thead>
<tr>
<th>Partition</th>
<th>Classical Storage</th>
<th>Hybrid Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_{G1} \times A_{G2} )</td>
<td># states [state] Memory</td>
<td># packets [packet] Memory</td>
</tr>
<tr>
<td>( A_P )</td>
<td>101 20B 2020B</td>
<td>101 20B 2020B</td>
</tr>
<tr>
<td>( A_P \times A_{dem} )</td>
<td>243 20B+2b 5103B</td>
<td>101 20B+4b 2121B</td>
</tr>
<tr>
<td>( A_P \times (\phi \land \neg A_{dem} \land A_{dem}) )</td>
<td>670 20B+5b 14070B</td>
<td>101 20B+32b 2424B</td>
</tr>
</tbody>
</table>

Table 1
Comparison between classical and hybrid storage

In summary, the hybrid storage method can be of great practical interest when some of the components in the product of automata are susceptible to be reached with several different states for a single state of the rest of the components, as it is often the case for the demon process and as it might also be the case for the automaton \( A_{\neg f} \) corresponding to the negation of the property being verified (especially when this property does not really constraint the behaviors of the program \( A_P \)).

Note that the hybrid storage method is quite different from, and orthogonal to, other techniques like the state compression schemes used in [17]. These techniques can easily be combined to increase the reduction still further.

5. Comparison With Other Work and Conclusions

We have presented an 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 the hybrid storage technique, its memory requirements are close to that of a conventional state space exploration of the program alone, i.e., without being combined with a property.
- The algorithm is compatible with techniques that can be used to increase the scope of automated validation, as bit-state hashing and state space caching techniques (e.g., see [18]).
- If the program does not satisfy the property being checked, a complete counter-example can be produced to the user, which explains why the program violates the property. This is required to understand what the error in the program (or in the property) is and to correct it.
- Last but not least, the algorithm is straightforward to implement: it is a minor modification of a standard depth-first search.

Previously existing algorithms do not have all the interesting properties mentioned above. As we recalled in Section 3, Tarjan’s algorithm requires the storage of the value of the variable DFNUMBER with each reached states. This requires more than two bits per state, the precise overhead of our algorithm which therefore requires less memory. Moreover, Tarjan’s algorithm is not compatible with bit-state hashing techniques, and is more complicated. Although “simplicity” is a subjective notion, it should be easy to convince oneself of the difference in complexity of the standard Tarjan algorithm (see [1]) and the algorithm presented here.
The algorithms given in [3] requires either several successive searches (first version; cf. Section 3.1), or a doubled amount of memory (second version), while our algorithm solves both problems by using simultaneously a demon process and a hybrid storage technique. Moreover, the hybrid storage technique makes the so-called “automata-theoretic approach to model-checking” [11] yet more attractive by tackling the problem that $P \times A_{-f}$ can be much bigger than $P$ alone, since it can reduce its memory requirements to approximately the ones of simple reachability analysis.

In [19], an algorithm is given for verifying a property expressed by a deterministic Büchi automaton without computing strongly connected components. Our algorithm is more general since it can deal with nondeterministic Büchi automata as well as with deterministic ones. (For instance, the algorithm of [19] could not be used to verify the property modeled by the Büchi automaton of Figure 2.)

Finally, several algorithms (based on Tarjan’s algorithm) have been proposed to solve the model-checking problem with some precise pre-defined fairness assumptions [12]. Instead of requiring a pre-defined notion of fairness, our algorithm can adapt to arbitrary fairness assumptions and is much simpler.

The algorithm presented in Section 3.2 has been implemented in the SPIN protocol validation tool in mid 1992 (though as yet without the hybrid storage method), replacing an earlier algorithm based on [3]. Noncommercial users can obtain the SPIN system via anonymous ftp from research.att.com from the /netlib/spin directory.

The importance of the development of efficient algorithms for the verification of temporal logic formulas will need little justification. As an 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 generic 5ESS\textsuperscript{3} telephone switches, 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 formulas, 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 SDL\textsuperscript{VALID} [20], 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 not only feasible, even for fairly large-scale industrial applications, but also extremely effective.

Acknowledgements

We wish to thank Pascal Gribomont, Doron Peled, Didier Pirottin, Pierre Wolper and anonymous referees for helpful comments on this paper.

REFERENCES

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer

\textsuperscript{3}5ESS is a registered trademark of AT&T.


