On-the-fly Verification of Linear Temporal Logic

Jean-Michel Couvreur

LaBRI, Université de Bordeaux I, Talence, France
couvreur@labri.u-bordeaux.fr

Abstract. In this paper we present two new practical and pragmatic algorithms for solving the two key on-the-fly model-checking problems for linear temporal logic: on demand construction of an automaton for a temporal logic formula; and on-the-fly checking for whether the automata resulting from the product of the program and the property is empty.

1 Introduction

Automatically checking whether a finite state program satisfies its linear specification is a problem that has gained a lot of attention during the last 15 years. Linear temporal logic is a powerful specification language for expressing safety, liveness, and fairness properties. However, the model-checking problem is known do be PSPACE-complete [18]. In practice, model-checking methods face complexity related limits: the size of the state space program, the size of the underlying automaton that represents the formula, and the size of the product automaton on which model-checking algorithms are applied. Many techniques have been designed to avoid the state explosion problem. By representing programs and automata of formulas using binary decision diagram [1], [3], [4], it is possible to check very large concurrent systems [5]. Another optimizing approach is on-the-fly verification [7], [9], [11], [13], which consists in constructing the program state space, the negation of the property, and the product automaton while checking for the emptiness of the product automaton. An advantage of this approach is that the algorithm can give an answer before the full program state space and the property automaton have been constructed. On-the-fly verification can be combined with methods [10], [14], [17], [22], [23] that avoid the exploration of the complete state space by performing reductions using partial order semantics. Success stories for both methods clearly demonstrate the effectiveness of automatic verification, even for fairly large-scale industrial applications.

In this paper we present new practical and pragmatic algorithms designed for solving the two key on-the-fly model-checking problems for linear temporal logic:

- Constructing on demand an automaton for a temporal logic formula;
- Checking on-the-fly whether the automaton resulting from the product of the program and the property is empty.

Both of these problems have already been solved in an efficient way. The automaton construction algorithm in [9] appears to produce reasonable sized automata

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. I, LNCS 1708, pp. 253–271, 1999.

for temporal logic formulas and it operates on-the-fly. Algorithms in [7], [11], [12], [13] check on-the-fly for the emptiness of the product automaton.

Our new automata construction is always better than the one in [9]. It appears to produce smaller automata. From a pragmatic point of view, we build is a variant of Büchi automata, namely transition Büchi automata. As opposed to simple Büchi automata, which have only one set of accepting states [20], transition Büchi automata have multiple sets of accepting transitions. As was proven in [15], ω-automata with accepting conditions on transitions are simpler than ω-automata with accepting conditions on nodes. Automaton construction is very similar than the one proposed in [9]. It is also based on tableau procedures [25], [26]. The key point of our method is the use of symbolic computation, which allows us to simplify expressions in a natural way and then to reduce the number of nodes. Moreover, the implementation can be done efficiently and easily using Binary Decision Diagrams [1], [3], [4].

The new checking algorithm is a simple variation of the Tarjan algorithm. It has the following features:

- The algorithm is designed to run on-the-fly, that is, during the traversal of the product automaton, failure is detected as soon as a failure component is encountered.
- The algorithm works directly on transition Büchi automata with multiple accepting conditions, that is, no expansion of the transition Büchi automaton into a simple Büchi automaton is needed.
- The algorithm can be used for checking temporal properties under fairness assumptions of the form $\bigwedge_i GFp_i$, without needless overhead.

Previously existing algorithms [7], [11], [13] do not have any of the interesting properties mentioned above.

The rest of the paper starts with some preliminaries defining finite state programs, temporal logic, and its interpretations. Section 3 presents transition Büchi automata and the automata construction for a temporal logic formula. Section 4 gives the model-checking algorithm. The paper finishes with some concluding remarks.

2 Preliminaries

Let AP be a set of atomic propositions. We write as 2^{AP} the mapping set $AP \to \{False, True\}$ and $2^{2^{AP}}$ the mapping set $2^{AP} \to \{False, True\}$. We may note that 2^{AP} can also be defined just as easily as the set of subsets of AP and $2^{2^{AP}}$ as the set of propositional formulas induced by atomic propositions. Sometimes we will consider an element p of AP as a propositional formula: $\forall y \in 2^{AP} p(y) \equiv (p \in y)$.

A finite state program P consists of the following components:

- S is a finite set of states;
- $\to \subseteq S \times 2^{AP} \times S$ is a transition relation;
- s_0 is an initial state ($s_0 \in S$).

Intuitively, the set S represents the set of states the system may enter. The relation $\rightarrow$ describes the actions available to states and the state transitions that may result upon execution of the actions. AP is used to associate atomic properties with transition relations. Without loss of generality (as in [16]), one can add loop transition relations, labeled $\{Dead\}$, to every terminal state. In this case every state has some enabled action. A run of P is an infinite sequence of states

$$\rho = s_0 \xrightarrow{x_0} s_1 \xrightarrow{x_1} s_2 \xrightarrow{x_2} \dots$$

such that for every $j \geq 0 : (s_j, x_j, s_{j+1}) \in \rightarrow$. We call a trace of ρ the infinite word over the alphabet 2^{AP} such that $trace(\rho) = x_0 \cdot x_1 \cdot x_2 \dots$

We use linear temporal logic (LTL) for our specification language. It defines a logic for the trace set of a program. We will say that program P fulfils a linear temporal property iff every trace of P fulfils f. The formal syntax for LTL is given below.

1. Every atomic proposition $p \in AP$ is a LTL formula.
2. If f and g are LTL formulas, then so are $\neg f$, $f \wedge g$, Xf , fUg.

An interpretation for a LTL formula is an infinite word $\sigma = x_0.x_1.x_2 \dots$ over the alphabet 2^{AP}. We use the standard notation $\sigma \models f$ to indicate the truth of a LTL formula f for an infinite word σ. We write σ_i, for the suffix of σ starting at x_i. The relation $\models$ is defined inductively as follows:

1. $\sigma \models p$ if $x_0(p)$ for $p \in AP$;
2. $\sigma \models \neg f$ if $\neg(\sigma \models f)$;
3. $\sigma \models f \wedge g$ if $\sigma \models f$ and $\sigma \models g$;
4. $\sigma \models Xf$ if $\sigma_1 \models f$;
5. $\sigma \models fUg$ if $\exists i \geq 0 : \sigma_i \models g \wedge (\forall j < i : \sigma_j \models f)$.

The standard boolean operators, $true$ and $false$ can also be used to construct LTL formulas. We also use the following abbreviations: $F\varphi = trueU\varphi$, $G\varphi = \neg F \neg \varphi$ and $fVg = \neg(\neg fU \neg g)$.

Remark 1. Every LTL formula can be rewritten as an equivalent LTL formula where the $\neg$ unary operator is applied only to atomic propositions. In the following, we will consider only such formulas.

Remark 2. Programs can also be constructed using parameterized transition systems [2]. In such programs, atomic propositions can also be associated with states. This model is useful but does not give any extension when using LTL formulas for the specification language: the atomic propositions of a state can be moved to all its output transitions.

3 A Tableau Construction

Our goal is to build an automaton that generates all infinite words satisfying a given formula f. The automaton we build is a transition Büchi automaton. As opposed to simple Büchi automata that have only one set of accepting states [20], transition Büchi automata have multiple sets of accepting transitions.

Formally a transition Büchi automaton $< Q, Acc, \rightarrow, q_0 >$has the following components:

- Q is a finite set of states;
- Acc is a finite set of accepting conditions;
- $\rightarrow \subseteq S \times 2^{2^{AP}} \times 2^{Acc} \times S$ is a transition relation;
- q_0 is an initial state ($q_0 \in Q$).

An infinite word $\sigma = x_0.x_1.x_2 \ldots$ over the alphabet 2^{AP} is accepted by a transition Büchi automaton iff there exists an infinite path

$$\rho = q_0 \overset{(X_0,A_0)}{\longrightarrow} q_1 \overset{(X_1,A_1)}{\longrightarrow} q_2 \overset{(X_2,A_2)}{\longrightarrow} \ldots$$

such that $\forall i \geq 0 : ((q_i, X_i, A_i, q_{i+1}) \in \rightarrow) \wedge (\forall a \in Acc, \forall i \geq 0, \exists j \geq i : a \in A_j)$.

The automaton construction is very similar to the one proposed in [9]. It is also based on tableau procedures [25], [26]. The nodes of the graph are labeled by a set of formulas and the transitions are obtained by expanding the temporal operators in order to distinguish what has to be true immediately from what has to be true from the next state on. The fundamental assertions, which are used for this expansion, are:

$$fUg = g + f \cdot X(fUg)$$
$$fVg = f \cdot g + g \cdot X(fVg)$$

where + is the boolean "or" operator and . is the boolean "and" operator.

Our automata construction is based on symbolic computation over a set of boolean variables constructed as follows:

- Every atomic proposition is a boolean variable,
- If f is a LTL formula then r_f is a boolean variable,
- If fUg is a LTL formula then a_{fUg} is a boolean variable.

Intuitively, for an infinite word $\sigma = x_0 \cdot x_1 \cdot x_2 \ldots$ over the alphabet 2^{AP}, r_f corresponds to $f \models \sigma$ and a_{fUg} corresponds to $(\sigma \models fUg) \wedge \neg(\sigma \models g)$. The fundamental identities of Boolean variables r_f and a_{fUg} are:

$$\begin{aligned}
r_{fUg} &= r_g + a_{fUg} \cdot r_f \cdot r_{X(fUg)} \\
r_{fVg} &= r_f \cdot r_g + r_g \cdot r_{X(fVg)} \\
r_{f \wedge g} &= r_f \cdot r_g \\
r_{f \vee g} &= r_f + r_g \\
r_p &= p \\
r_{\neg p} &= \neg p
\end{aligned}$$

Using the fundamental identities, and given a LTL formula f, variable r_f can be expressed in an expression which only uses variables of the form p, $\neg p$, a_g and r_{Xg}, where variables p are atomic propositions and g are subformulas of f. Proposition 1 is the application of this property to a set of formulas F.

Proposition 1. *Let F be a set of formulas. $\Delta(F) = \prod_{f \in F} r_f$ can be expanded to the form:*

$$\prod_{f \in F} r_f = \sum_{(X, Nacc, Next) \in L_F} \left(X \cdot \prod_{g \in Nacc} a_g \cdot \prod_{h \in Next} r_{Xh} \right)$$

with

$$L_F \subseteq 2^{2^{AP}} \times \{gUh \in Sub(f)\} \times \{\{gUh \in Sub(f)\} \cup \{gVh \in Sub(f)\} \cup \{g \in Sub(f) : Xg \in Sub(f)\}\}.$$

Proof. Obvious.

A transition Büchi automaton that accepts exactly the infinite words satisfying formula f, is obtained by expanding formulas of the form $\prod_g r_g$. The set of accepting condition for the automaton is composed of the subformulas gUh in f. The automata construction starts by each expanding variable r_f. Each implicant of this expansion defines a transition $(f, X, Acc \setminus NAcc, Next)$. We then expand $\prod_{g \in Next} r_g$ to produce new nodes and new transitions in the same manner. Theorem 1 formalizes the resulting automaton.

Theorem 1. *Let f be a LTL formula. Let $Bu(f)$ be the transition Büchi automaton defined by:*

- $Q = \{F \subseteq \{gUh \in Sub(f)\} \cup \{gVh \in Sub(f)\} \cup \{g \in Sub(f) : Xg \in Sub(f)\} \cup \{f\}\}$;
- $Acc = \{gUh \in Sub(f)\}$;
- $\rightarrow = \{(F, X, Acc \setminus NAcc, Next) : (X, NAcc, Next) \in L_F\} \cup \{(\emptyset, true, Acc, \emptyset)\}$
- $\{f\}$ *is an initial state.*

Then $Bu(f)$ accepts exactly the infinite words over the alphabet 2^{AP} that satisfy f.

Proof. ($\Rightarrow$) Let $\sigma = x_0 \cdot x_1 \cdot x_2 \ldots$ be an infinite word accepted by $Bu(f)$. Let us prove that σ satisfies formula f.

By definition, there exists an infinite path in $Bu(f)$

$$\rho = q_0 \overset{(X_0, A_0)}{\longrightarrow} q_1 \overset{(X_1, A_1)}{\longrightarrow} q_2 \overset{(X_2, A_2)}{\longrightarrow} \ldots$$

such that $\forall i \geq 0 : ((q_i, X_i, A_i, q_{i+1}) \in \rightarrow) \wedge (\forall a \in Acc, \forall i \geq 0, \exists j \geq i : a \in A_j)$.

For each transition (q_i, X_i, A_i, q_{i+1}), consider the boolean variables $r[i]_g$ for each LTL formula g, and boolean variable $a[i]_{gUh}$ for each accepting condition gUh.

Let the sets $r[i]_{Xg} = [g \in q_{i+1}]$, $r[i]_p = [p \in x_i]$ for $p \in AP$, $a[i]_{gUh} = [gUh \in A_i]$. Using fundamental identities, we deduce the value of each variable $r[i]_g$ for any LTL formula g. By the construction of $Bu(f)$, $r[i]_g$ is true for any formula in q_i. Indeed, the initial value of the implicant associated with transition was *true*, so $\prod_{g \in q_i} r[i]_g$ must be *true*.

We can establish by induction on the size of the formula that, if $r[i]_g$ is *true*, then the suffix word σ_i satisfies g. This will conclude the first part of the proof.

- If p is an atomic proposition and $r[i]_p$ is true then $p \in x_i$ and then σ_i satisfies p;
- If $r[i]_{g \wedge h}$ is *true* then $r[i]_{g \wedge h} = r[i]_g \cdot r[i]_h = true$ and by induction σ_i satisfies g and h;
- If $r[i]_{g \vee h}$ is *true* then $r[i]_{g \vee h} = r[i]_g + r[i]_h = true$ and by induction σ_i satisfies g or h;
- If $r[i]_{gUh}$ is *true* then $r[i]_{gUh} = r[i]_h + a[i]_{gUh} \cdot r[i]_g \cdot r[i]_{X(gUh)} = true$; if $r[i]_h = true$ then by induction σ_i satisfies h and thus gUh; otherwise $r[i]_{X(gUh)} = true$ and $r[i]_g = true$, by definition $gUh \in q_{i+1}$ and then $r[i+1]_{gUh} = true$ and by induction σ_i satisfies g. In the latter case one can apply the same deduction for $j > i$ until $r[j]_h = true$. This procedure will eventually stop at least when $a[j]_{gUh}$ is false ($gUh \in A_j$).
- If $r[i]_{gVh}$ is *true* then $r[i]_{gVh} = r[i]_g \cdot r[i]_h + r[i]_h \cdot r[i]_{X(gVh)} = true$; if $r[i]_g = r[i]_h = true$ then by induction σ_i satisfies g and h,and thus gVh; otherwise $r[i]_{X(gVh)} = true$ and $r[i]_h = true$, by definition $gVh \in q_{i+1}$ and then $r[i+1]_{gVh} = true$ and by induction σ_i satisfies h. In the latter case one can apply the same deduction for $j > i$ until $r[j]_g = true$. This procedure can stop or proceed infinitely; in both case, we deduce that σ_i satisfies gVh.

($\Leftarrow$) Let $\sigma = x_0 \cdot x_1 \cdot x_2 \ldots$ be an infinite word satisfying formula f. Let us prove that σ is accepted by $Bu(f)$.

One can construct an infinite path in $Bu(f)$

$$\rho = q_0 \overset{(X_0,A_0)}{\longrightarrow} q_1 \overset{(X_1,A_1)}{\longrightarrow} q_2 \overset{(X_2,A_2)}{\longrightarrow} \ldots$$

such that $\forall i \geq 0 : X_i(x_i); \forall g \in q_i, \sigma_i$ satisfies g, and if σ_i satisfies $h \vee \neg(gUh)$ then $gUh \in A_i$.

Each transition (q_i, X_i, A_i, q_{i+1}) corresponds to a *true* implicant in formula $\prod_{g \in q_i} r_g$ when setting variables: $r_{Xg} = [\sigma_{i+1} \models g]$, $r[i]_p = [p \in x_i]$ for $p \in AP$, $a_{gUh} = (\sigma_i \models gUh) \wedge \neg(\sigma_i \models h)$. Any accepting condition gUh appears infinitely often in the path. Otherwise there exists an $i > 0$ and a formula gUh such that $\forall j > i, \sigma_j$ satisfies $\neg h \wedge (gUh)$.

Example 1. Construction of an automaton for formula $f = pU(qUs)$

Let $g = qUs$. We deduce $Acc = \{f, g\}$. The fundamental identities used in the construction are:

$$r_f = r_g + a_f \cdot p \cdot r_{Xf}$$

$$r_g = s + a_g \cdot q \cdot r_{Xg}$$

Firstly, one can expand variable r_f:

$$r_f = r_g + a_f \cdot p \cdot r_{Xf} = s + a_g \cdot q \cdot r_{Xg} + a_f.p.r_{Xf}$$

This expansion produces 3 transitions:

$$(\{f\}, s, \{f, g\}, \emptyset)$$

$$(\{f\}, q, \{f\}, \{g\})$$

$$(\{f\}, p, \{g\}, \{f\})$$

Secondly, one has to produce the successors of states $\{g\}$ and $\emptyset$. The expansion of r_g is immediate: $r_g = s + a_g \cdot q \cdot r_{Xg}$ and produces transitions:

$$(\{g\}, s, \{f, g\}, \emptyset)$$

$$(\{g\}, q, \{f\}, \{g\}).$$

The transition for state $\emptyset$ is always $(\emptyset, true, \{f, g\}, \emptyset)$. Figure 1 gives the resulting automaton.

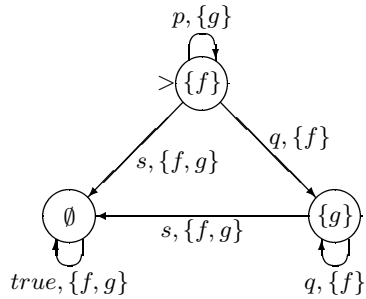

Fig. 1. A transition Büchi automaton for the formula $pU(qUs)$

Example 2. Construction of an automaton for Formula $f = GXFp$

Let $g = Fp$. We deduce $Acc = \{g\}$.

The fundamental identities used in the construction are:

$$r_f = r_{Xg} \cdot r_{Xf}$$

$$r_g = p + a_g \cdot r_{Xg}$$

First, we expand the variable r_f:

$$r_f = r_{Xg} \cdot r_{Xf}$$

This expansion produces the transition $(\{f, true, \{g\}, \{f, g\})$.

Second, we produce the successors of states $\{f, g\}$. The expansion of $r_f \cdot r_g$ is:

$$r_f \cdot r_g = p \cdot r_{Xg} \cdot r_{Xf} + a_g \cdot r_{Xg} \cdot r_{Xf}$$

and produces two transitions:

$$(\{f, g\}, p, \{g\}, \{f, g\})$$

$$(\{f, g\}, True, \emptyset, \{f, g\})$$

Figure 2 gives the resulting automaton.

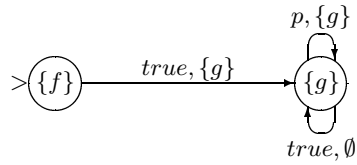

Fig. 2. A transition Büchi automaton for the formula $GXFp$

The implementation of this automaton construction can be done efficiently and easily using Binary Decision Diagrams [1], [3], [4]: one only needs classical BDD operations (Boolean operators, variable substitution) and a way to extract the prime implicants of Boolean functions [6], [8]. The following table compares the (previous) construction described in [9] and the one based on Theorem 1. Formulas and results come for previous construction from [9].

Table 1. Comparing the construction described in [9] and the new construction

Formula	| Acc |	Previous Construction		New Construction	
		Nodes	Transitions	Nodes	Transitions
pUq	1	3	4	2	3
$pU(qUs)$	2	4	6	3	6
$\neg((pU(qUs))$	0	7	15	3	6
$GFp \Rightarrow GFq$	2	9	15	5	11
$FpUGq$	2	8	15	4	10
$GpUq$	1	5	6	4	6
$\neg(FFp \Leftrightarrow Fp)$	2	22	41	3	5

The new construction is always better. This is not a surprise. One can remark that the number of nodes for new construction is in $O(2^{temporal(f)})$, where

$temporal(f)$ is number of temporal operators in formula f, while for the previous one, it is in $O(2^{|f|})$. Moreover the use of symbolic computation makes it possible to simplify expressions in a natural way and then reduces the number nodes.

The new construction can be improved in some ways. For instance:

- When two nodes F, G represent equivalent formulas $\left(\prod_{f\in F} f \Leftrightarrow \prod_{g\in G} g\right)$, one can merge them. A sufficient condition to detect such a situation is $\left(\prod_{f\in F} r_f = \prod_{g\in G} r_g\right)$. Using binary decision diagram, this improvement is easy to implement. One can identify each node F by the binary decision diagram of the expanded expression of $\prod_{f\in F} r_f$.

Table 2. Result of the improved construction

Formula	|Acc|	Improved Construction	
		Nodes	Transitions
$GFp \Rightarrow GFq$	2	4	9
$\neg(FFp \Leftrightarrow Fp)$	2	2	3

- The operator GF is commonly used when writing LTL formulas. The automaton construction introduces nodes with equivalent formulas $GFf = GFf \wedge Ff$. An efficient way to take into account the node reduction introduced by this equivalence is to add a new fundamental identity:

$$r_{GFf} = r_f \cdot r_{XGFf} + a_{GFf} \cdot r_{XGFf}$$

  As a result, for a formula $\bigwedge_i GFp_i$ that represents some fairness properties, one can obtain an automaton with a single node where otherwise the number of nodes would be exponential.

The automaton construction can be used for checking the validity of an LTL formula f for a program P [24]. We first build a transition Büchi automaton for the negation of the formula, $Bu(\neg f) = (Q, Acc, \rightarrow, q_0)$. Then we compute the product automaton $P \times Bu(\neg f) = (S \times Q, Acc, \rightarrow, (s_0, q_0))$ where

$$(s,q) \xrightarrow{(x,A)} (s',q') \Leftrightarrow \left(s \xrightarrow{x} s' \bigwedge q \xrightarrow{(X,A)} q' \bigwedge X(x)\right)$$

This product automaton accepts all traces of P which fulfil $\neg f$. Finally, we check whether the automaton $P \times Bu(\neg f)$ is empty. If $P \times Bu(\neg f)$ is empty, we conclude that every trace of P fulfils f.

Usually programs are defined using some high level language [13]. In such cases, one has to build the program automaton P and the transition Büchi automaton $Bu(\neg f)$, before constructing the product automaton and checking its

emptiness. A classic improvement of this procedure is on-the-fly model checking: construction of nodes for the program automaton and the transition Büchi automaton is be done on demand, while constructing the product automaton and checking its for emptiness. Thus, it is possible to stop the procedure when a violation of the checked property is detected.

To construct all of the successors of node (s, q), one has to build for every transition (s, x, s') of the program, all the transitions (q, X, A, q') of the transition Büchi automaton which match (s, x, s'), i.e $x \in X$. A simple procedure is to build all the successors of q and then select the ones which match. However, this procedure can be improved by just expanding $\Delta(F)(x)$, i.e. $\Delta(F)$ where variables of AP are bounded to Boolean value $[x \in X]$:

$$\prod_{f \in F} r_f = \sum_{(X, Nacc, Next) \in L_F} \left(\prod_{g \in Nacc} a_g \cdot \prod_{h \in Next} r_{Xh} \right)$$

In the underlying transition Büchi automaton this procedure considers that every propositional formula X of each implicant is a mapping x of 2^{AP}. As a proof of the efficiency of this procedure, $Bu\left(\bigwedge_i GFp_i\right)$ has a single node and an exponential number of transitions, and only one matching transition for a program transition (s, x, s').

4 Checking Algorithm

To check whether the transition Büchi automaton $P \times Bu(\neg f)$ is nonempty, one has to check whether there exists a failure cycle in $P \times Bu(\neg f)$, that is a cycle that contains at least one transition for each accepting condition and which is reachable from the initial state (s_0, q_0). Note that it is not necessary to consider all possible cycles of $P \times Bu(\neg f)$: it is sufficient to check if $P \times Bu(\neg f)$ contains a failure component, that is a strongly connected component that is reachable from the initial state and which includes at least one transition for each accepting condition.

Searching for maximal strongly connected components can be done with the Tarjan algorithm [19], [21]. This algorithm is based on a depth-first search. It uses two additional variables with each node NFNUMBER and LOWLINK: NFNUMBER gives the order of the first visit of a node and LOWLINK characterizes roots of strongly connected components as nodes with NFNUMBER=LOWLINK. During the depth-first search, values of NFNUMBER and LOWLINK are updated. When a node with NFNUMBER=LOWLINK is reached by backtracking, a strongly connected component is computed and removed from the graph. Any property on a strongly connected component can be checked when it is removed. The main problem of this algorithm is that a strongly connected component is first completely traversed before it is checked. The algorithm cannot detect graph failures in the fly, that is, stop the algorithm when the traversed graph contains a failure cycle.

The new checking algorithm presented in the section is a simple variation of the Tarjan algorithm. The LOWLINK node variables are replaced by a stack of the NFNUMBER values of the roots of strongly connected components of the traversed graph. Each NFNUMBER value of the root stack is associated with the set of accepting conditions of its connected component. During the depth-first search, this stack is updated and then the sets of accepting conditions of strongly connected components of the traversed graph are always known. The algorithm detects graph failures in the fly, and one can stop the computation when the traversed graph contains a failure component.

A description of the new algorithm is given in Figure 3. The data structure we use for representing transition Büchi automata contains sufficient information for the checking algorithm:

- $S0$ is the initial node,
- ACC is the accepting condition set,
- $RELATION \subseteq S \times ACC \times S$ is the transition relation (only accepting conditions are used for the checking algorithm).

The additional data structures of the new algorithm are:

- Num gives the number of nodes of the current graph; it is used to set the order of the first visit of a node,
- $Hash$ is a hash table of pairs $(node, integer)$; it is used to store visiting nodes. The node component is the search key and the integer component gives the node order. When the order $order(v)$ of a node v is null, it means that node v has been already visited and is removed from the graph.
- *Root* is a stack of pairs $(integer, accepting\ condition\ set)$; it stores the root number of strongly connected components of the current graph and its accepting condition set.
- *Arc* is a stack of accepting condition sets; it gives the accepting condition set of the arcs which connect strongly connected components of the current graph.

During the execution of the algorithm, one can consider

- the removed graph: the subgraph containing nodes store in table $Hash$ with order null,
- the current graph: the sub-graph of the traversed graph containing nodes store in table $Hash$ with order non null,
- the search path: the path in the graph induces by the depth-first search.

The new algorithm is designed to preserve the following properties each time a transition is traversed or a new node is visited.

Property 1. The removed graph is a union of nonfailure strongly connected components of the full graph;

Property 2. Stack *Root* contains only nodes of the search path in the same order;

```
1       Check(){
2          Num = 1 ;
3          Hash.put(S0,1)
4          Root.push(1,EMPTYSET) ;
5          Arc.push(EMPTYSET) ;
6          Explore(S0,1) ;
7       }

8       Explore(Node v,int vorder){
9          for all (A,w) such that (v,A,w) in RELATION do {
10            (b,wo) = Hash.get(w) ;
11            if not(b) then {
12            (* w is a new explored node *)
13                     Num = Num+1 ;
14               Hash.put(w,Num) ;
15               Root.push(Num,EMPTYSET) ;
16               Arc.push(A) ;
17               Explore(w,Num) ;
18            }
19            else if (worder <> 0) then {
20            (* w is a node of the current graph *)
21               (i,B) = Root.pop() ; B = B union A ;
22                while i>worder do {
23                   A = Arc.pop() ; B = B union A ;
24                   (i,A) = Root.pop() ; B = B union A ;
25               }
26               Root.push(i,B) ;
27               If B == ACC then report erreur ;
28            }
29         }
30         (i,B) = Root.top() ;
31         if vorder == i then {
32         (* v is the root of a strongly connected component *)
33            Num = vi-1 ;
34            Root.pop() ;
35            Arc.pop() ;
36            Remove(v) ;
37         }
38      }

39      Remove(Node v){
40         b=Hash.testset0(v) ;
41         if b then
42            for all w such that (v,A,w) in RELATION do
43               Remove(w) ;
44      }
```

Fig. 3. Checking algorithm

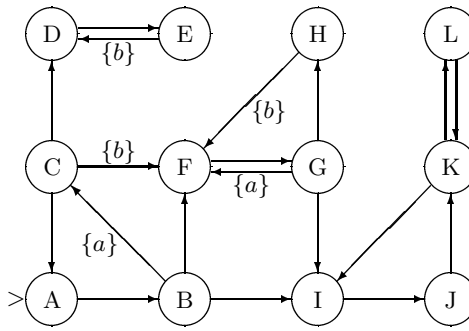

Fig. 4. A simple product transition Büchi automaton with $Acc = \{a, b\}$

Property 3. If $Root = (o_1, A_1)(o_2, A_2)\ldots(o_p, A_p)$ then $o_1 = 1$ and the strongly connected components C_i of the current graph are exactly

$$\forall i < p : C_i = \{v \in current\ graph / o_i \leq order(v) \leq o_{i+1} - 1\}$$

and

$$C_p = \{v \in current\ graph / o_p \leq order(v) \leq Num\}$$

Property 4. Given $Arc = B_1 \cdot B_2 \ldots B_p$, the strongly connected components are only connected in the current graph with the transitions $(order^{-1}(o_1 - 1), B_i, order^{-1}(o_i))$ with $i > 1$. B_1 is the accepting condition set of an artificial transition which is connected to the initial nodes, it is always set to $\emptyset$.

Example. A simple execution of the checking algorithm.

In order to illustrate all of the parts of the checking algorithm, let us apply it to the automaton in Figure 4. Unlabelled transitions mean that the corresponding accepting condition set is empty. Table 3 gives the values of the data structures at different steps of the checking algorithm. Notice that the properties 1-4 defined below are always fulfilled.

1. The initial part of the algorithm, Lines 2-6, is executed and $Explore(A, 1)$ is running.
2. The transition $A \rightarrow B$ is traversed. B is a new node ($Hash.get(B)$ return $(false, undefined)$) and then Lines 12-17 are executed. Node B defines a new strongly connected component of the current graph: its order is push in *Root* and the accepting condition set of transition $A \rightarrow B$ is pushed in *Arc*.
3. The transition $B \rightarrow C$ is traversed. C is also a new node and Lines 12-17 are executed.
4. The transition $C \rightarrow A$ is traversed. A is a node with order 1 ($Hash.get(A)$ returns $(true, 1)$) and then Lines 20-27 are executed. *Root* and *Arc* are popped until the head of *Root* is less or equal to 1. This operation merges components $\{A\}$, $\{B\}$ and $\{C\}$ in one strongly connected component $\{A, B, C\}$.

Moreover the accepting condition set of this component is updated to $\{a\}$. At Line 27, $\{a\} \neq \{a, b\}$ and so no error is reported.

5. The transition $C \rightarrow D$ is traversed. D is also a new node and Lines 12-17 are executed.
6. The transition $D \rightarrow E$ is traversed. E is also a new node and Lines 12-17 are executed.
7. The transition $E \rightarrow D$ is traversed. D is a node with order 4 and then Lines 20-27 are executed. *Root* and *Arc* are popped until the head of *Root* is less than or equal to 4. This operation merges components $\{D\}$ and $\{E\}$ into one strongly connected component.
8. All the successor transitions of Node E have been already traversed. Lines 30-31 check whether Node E is a root of a strongly connected component. This is not the case and the Node E are just backtracked.
9. All the successor transitions of Node D have been already traversed. Lines 30-31 check if Node D is a root of a strongly connected component. It is the case and then Lines 32-36 are executed: this operation removes the strongly connected component $\{D, E\}$ from the current graph. In the calling procedure *Remove* (Lines 39-44), $Hash.testset0(v)$ return $false$ if the order of v is 0 (v is already removed) else sets the order of v to 0 and return *true*.
10. The transition $C \rightarrow F$ is traversed. F is a new node and Lines 12-17 are executed.
11. The transition $F \rightarrow G$ is traversed. G is a new node and Lines 12-17 are executed.
12. The transition $G \rightarrow F$ is traversed. F is a node with order 4 and then Lines 20-27 are executed. *Root* and *Arc* are popped until the head of Stack *Root* is less or equal to 4. This operation merges components $\{F\}$ and $\{G\}$ into one strongly connected component $\{F, G\}$. At this point, the accepting condition set is updated to $\{b\}$.
13. The transition $G \rightarrow H$ is traversed. H is a new node and Lines 12-17 are executed.
14. The transition $H \rightarrow F$ is traversed. F is a node with order 4 and then Lines 20-27 are executed. *Root* and *Arc* are popped until the head of Stack *Root* is less than or equal to 4. This operation merges components $\{F, G\}$ and $\{H\}$ in one strongly connected component $\{F, G, H\}$. The accepting condition set is updated to $\{a, b\}$. At Line 27, an error is reported and the algorithm stops: $\{F, G, H\}$ is a failure connected component.

Theorem 2. *If there exists at least one failure connected component in a transition Büchi automaton, the checking algorithm will report an error. Moreover, the checking algorithm reports an error as soon as the traversed graph contains a failure component.*

Proof. The proof is simple but tedious. One has simply to verify the assertion properties 1-4 set on lines:

- Line 8 : before Procedure *Explore* is called
- Line 9 : before selecting a transition

Table 3. Execution of the checking algorithm

	Node	Transition	Root	Arc	Hash
1	A.1		1.∅	∅	A.1
2	A.1	A→B	1.∅, 2.∅	∅ ∅	A.1, B.2
3	B.2	B→C	1.∅, 2.∅, 3.∅	∅ ∅ a	A.1, B.2, C.3
4	C.3	C→A	1.{a}	∅	A.1, B.2, C.3
5	C.3	C→D	1.{a}, 4. ∅	∅ ∅	A.1, B.2, C.3, D.4
6	D.4	D→E	1.{a}, 4.∅, 5.∅	∅ ∅ ∅	A.1, B.2, C.3, D.4, E.5
7	E.5	E→D	1.{a}, 4.{b}	∅ ∅	A.1, B.2, C.3, D.4, E.5
8	E.5		1.{a}, 4.{b}	∅ ∅	A.1, B.2, C.3, D.4, E.5
9	D.4		1.{a}	∅	A.1, B.2, C.3, D.0, E.0
10	C.3	C→F	1.{a}, 4.∅	∅ {b}	A.1, B.2, C.3, F.4, D.0, E.0
11	F.4	F→G	1.{a}, 4.∅,5.∅	∅ {b} ∅	A.1, B.2, C.3, F.4, G.5, D.0, E.0
12	G.5	G→F	1.{a}, 4.{a}	∅ {b}	A.1, B.2, C.3, F.4, G.5, D.0, E.0
13	G.5	G→H	1.{a}, 4.{a}, 6.∅	∅ {b} ∅	A.1, B.2, C.3, F.4, G.5, H.6, D.0, E.0
14	H.6	H→G	1.{a}, 4.{a,b}	∅ {b}	A.1, B.2, C.3, F.4, G.5, H.6, D.0, E.0

- Line 18 : after visiting of a new node
- Line 27 : after visiting of a current node
- Line 28 : after checking for a failure component
- Line 30 : after visiting all the successor transitions
- Line 37 : after removing a possible non-failure strongly connected component
- Line 38 : at the end of Procedure *Explore*

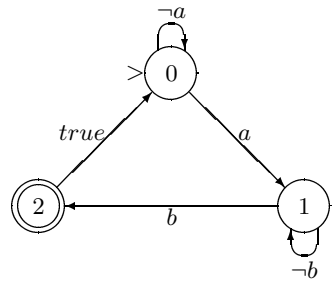

Fig. 5. Counter automaton

The previous algorithms [7], [13], [11] work an on-fly-way only for simple Büchi automata (one accepting condition). They consist of two depth-first searches. In the nicest and lastest version, the magic algorithm [11], each time an accepting state is backtracked by the first search, the second search checks if the accepting state is reachable from itself through a nontrivial path (see [11] for a complete presentation of this algorithm). This algorithm reduces the size of the required memory and is compatible with efficient partial verification techniques

such as bit-state hashing techniques [12]. However it needs some adaptation in order to manipulate multiple accepting conditions, which increases the size of the graph (linear time in the number of accepting conditions). Moreover, if a part of the traversed graph contains a failure, this algorithm does not necessary detect it immediately.

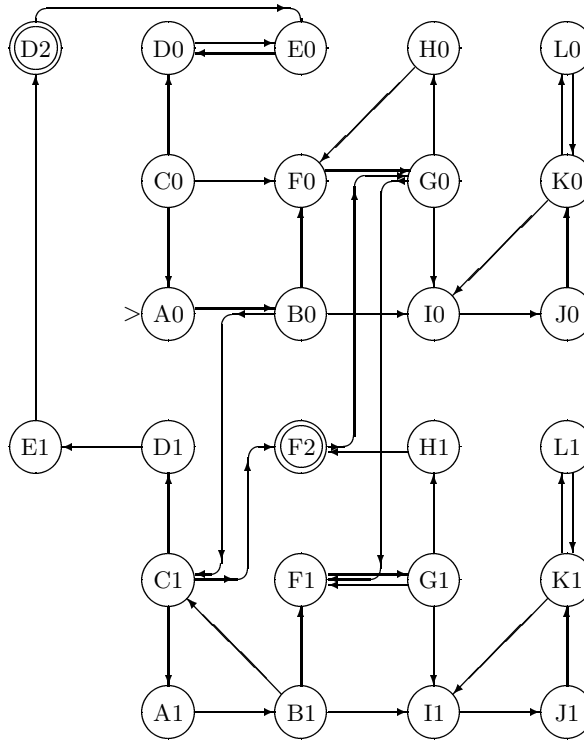

Fig. 6. Expansion of the product transition Büchi automaton

Example To apply the magic algorithm [11] to the transition Büchi automaton presented in Figure 4, one need first to expand it in a simple Büchi automaton (only one accepting condition associated with states, that is, a set of accepting states). This can be done by synchronizing it with the counter automaton (Figure 5). Figure 6 gives the resulted product Büchi automaton.

Table 4 traces the execution of the first depth-first search of the magic algorithm [11]. The second depth-first search starts when F2 is backtracked and detects the failure cycle $F2 \rightarrow G0 \rightarrow F1 \rightarrow G1 \rightarrow H1 \rightarrow F2$. At Step 13 of the execution of first search, all the transitions of the nontrivial path have been traversed and the magic algorithm does not immediately detect the failure.

Table 4. First depth search of the magic algorithm

	Node	Transition	Hash
1	A0		A0
2	A0	A0→B0	A0,B0
3	B0	B0→C1	A0,B0,C1
4	C1	C1→A1	A0,B0,C1,A1
5	A1	A1→B1	A0,B0,C1,A1,B1
6	B1	B1→C1	A0,B0,C1,A1,B1,C1
7	B1	B1→F1	A0,B0,C1,A1,B1,C1,F1
8	F1	F1→G1	A0,B0,C1,A1,B1,C1,F1,G1
9	G1	G1→F1	A0,B0,C1,A1,B1,C1,F1,G1
10	G1	G1→H1	A0,B0,C1,A1,B1,C1,F1,G1, H1
11	H1	H1→F2	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2
12	F2	F2→G0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0
13	G0	G0→F1	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0
14	G0	G0→H0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0
15	H0	H0→F0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0
16	F0	F0→G0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0
17	F0		
18	H0		
19	G0	G0→I0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0
20	I0	I0→J0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0
21	J0	J0→K0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0,K0
22	K0	K0→I0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0,K0
23	K0	K0→L0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0,K0,L0
24	L0	L0→K0	A0,B0,C1,A1,B1,C1,F1,G1, H1,F2,G0,H0, F0,I0,J0,K0,L0
25	L0		
26	K0		
27	J0		
28	I0		
29	G0		
30	F2		

5 Concluding Remarks

We have presented two new practical and pragmatic algorithms designed for solving the two key on-the-fly model-checking problems for linear temporal logic. The new automata construction has the same nice characteristics than the one in [9]: it is simple, it appears to produce reasonablly-sized automata and it operates on-the-fly. The new construction is always better: in that it produces smaller automata. The key point of our method is the use of symbolic computation. It allows us simplify expressions in a natural way and thus to reduce the number of nodes. A simple and efficient implementation can be done using Binary Decision Diagrams.

The new checking algorithm has the following features:

- The algorithm is designed to run on the fly, i.e., as soon as the traversed product automaton contains a failure component, the failure is detected.
- The algorithm works directly on transition Büchi automata with multiple accepting conditions, i.e, no expansion of the transition Büchi automaton into a simple Büchi automaton is need.
- The algorithm can be used for checking temporal properties under fairness assumptions of the form $\bigwedge_i GFp_i$, without needless overhead: a program running under a fairness assumption introduces accepting conditions (i.e. a program is viewed as a transition Büchi automaton) and the product automaton will still have the same size with new accepting conditions.

Previously existing algorithms [7], [11], [13] do not have any of the interesting properties mentioned above. However the new algorithm is not compatible with efficient partial verification as bit-state hashing technique [12]. For exhaustive verification, the storage of an additional integer with each state is not a problem in practical cases: usually the space to store a program state which is many order of magnitude larger than the space to store an integer. We claim that our algorithm is compatible with partial order methods [10], [17], [22], [23]. It does not need any modification as does the magic algorithm [14]. Moreover, our model-checking algorithm can be adapted in order to solve a classical partial order technique problem: When building a reduced state space, one has to assume some fairness properties in the construction. Informally an action must be executed if it is enabled forever; otherwise safety and liveness properties are not preserved.

References

[1] B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[2] A. Arnold. *Finite transition systems. Semantics of communicating systems.* Prentice-Hall, 1994.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *27th ACM/IEEE Design Automation Conference*, pages 40–45, Orlando, Florida, June 1990. ACM/IEEE, IEEE Computer Society Press.

[4] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[5] E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. *Lecture Notes in Computer Science*, 803, 1994.

[6] O. Coudert and J. C. Madre. Implicit and incremental computation of primes and essential implicant primes of boolean functions. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 36–39, 1992.

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

[8] Y. Dutuit and A. Rauzy. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia. *Reliability Engineering and System Safety*, 58:127–144, 1997.

[9] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.

[10] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer, Berlin, 1996.

[11] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proc. 13th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, pages 109–124, Liege, Belgium, May 1993.

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

[13] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[14] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *The Spin Verification System*, pages 23–32. American Mathematical Society, 1996. Proc. of the Second Spin Workshop.

[15] B. Lessaec. *Etude de la reconnaissabilité des langages de mots infinis*. PhD thesis, Université Bordeaux I, 1986.

[16] O. Lichtenstein and A. Pnueli. Checking the finite-state concurrent programs satisfy their linear specifications. In *popl85*, pages 97–107, 1985.

[17] D. Peled. All from one, one from all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification, Greece*, number 697 in Lecture Notes in Computer Science, pages 409–423, Berlin-Heidelberg-New York, 1993. Springer.

[18] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 32(3):733–749, July 1985.

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

[20] W. Thomas. Automata on infinite objects. In *Handbook of theoretical computer science, Volume B : Formal models and semantics*, pages 165–191. Elsevier Science Publishers, 1990.

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

[22] A. Valmari. Stubborn sets for reduced state space generation. *Lecture Notes in Computer Science*, 483:491–515, 1990.

[23] A. Valmari. On-the-fly verification with stubborn sets representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification, Greece*, number 697 in Lecture Notes in Computer Science, pages 397–408, Berlin-Heidelberg-New York, 1993. Springer.

[24] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[25] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1–2):72–99, 1983.

[26] P. Wolper. The tableau method for temporal logic: An overview. *Logique et Analyse*, (110–111):119–136, 1985.