Simple On-the-fly Automatic Verification of Linear Temporal Logic

R. Gerth
Technical University Eindhoven
Den Dolech 2, Eindhoven, The Netherlands

D. Peled
AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA

M. Y. Vardi
Rice University
Department of Computer Science, Houston TX 77251, USA

P. Wolper[1]
Université de Liège
Institut Montefiore, B28, 4000 Liège, Belgium

Abstract

We present a tableau-based algorithm for obtaining an automaton from a temporal logic formula. The algorithm is geared towards being used in model checking in an "on-the-fly" fashion, that is the automaton can be constructed simultaneously with, and guided by, the generation of the model. In particular, it is possible to detect that a property does not hold by only constructing part of the model and of the automaton. The algorithm can also be used to check the validity of a temporal logic assertion. Although the general problem is PSPACE-complete, experiments show that our algorithm performs quite well on the temporal formulas typically encountered in verification. While basing linear-time temporal logic model-checking upon a transformation to automata is not new, the details of how to do this efficiently, and in "on-the-fly" fashion have never been given.

Keywords

Automatic Verification, Linear Temporal Logic, Büchi Automata, Concurrency, Specification.

1 Introduction

Checking automatically that a protocol, especially a concurrent one with many parallel activities, satisfies its specification has gained a lot of attention during the last 15 years. The main

[1]The work of this author was supported by the Esprit BRA action REACT and by the Belgian Incentive Program "Information Technology" - Computer Science of the future, initiated by the Belgian State - Prime Minister's Office - Science Policy Office. The scientific responsibility is assumed by its authors.

dichotomy between approaches to automated protocol verification can be characterized as logic-based versus state-space based methods. The former type of methods proceed by translating both the protocol and its specification into formulas in some formal logic and by showing logical implication of the specification by the protocol formula. In contrast, state-space based methods proceed by analyzing the possible configurations the protocol can be in, i.e. its state space, and how the protocol evolves from one configuration to another. None of these methods offer a uniform advantage; both have strengths and weaknesses when compared to the other.

This paper concentrates on a class of state-space based methods, often called "model checking". The idea of model checking is to view verification as checking whether the graph representing the state space of the protocol satisfies (is a *model* of) the property to be checked. Specifically, we focus on model checking for linear-time temporal logic formulas [9]. In this context, what one actually checks is that all infinite execution sequences that can be extracted from the state-space graph satisfy (are models of) the temporal logic formula, or equivalently, that none of these sequences falsifies the formula.

A classical approach to solving this problem [12] is to proceed as follows. One first constructs the state spaces for both the protocol to be verified and for the *negation* of the property, the latter state space thus comprises all execution sequences (models) on which the property is violated. The two state spaces are then analyzed for the existence of a common execution sequence; finding one means that the property can be violated by the protocol. Given that one is interested in the infinite sequences that can be generated by the two state spaces, these can be interpreted as automata over infinite words, i.e., as ω-automata [11]. The analysis to be done thus amounts to the standard problem of checking if the language accepted by the (synchronous) product of the automata is empty or not. A general approach for solving this problem proceeds by checking for strongly connected components as is done in [8], but one can also reduce the problem to a simpler cycle detection for which simpler algorithms can be used [6, 4].

The model-checking problem as well as the validity problem for linear temporal logic are PSPACE-complete [10]. In practice, applications of model-checking methods face two complexity related limits:

1. The size of the automata, both for the protocol and for the property, since the execution time is proportional to the product of the number of nodes in the automata;
2. The size of that part of the product automaton that has to be kept in memory in order to check for emptiness, since available memory sets a firm bound on the size of the problems that can be treated.

As to the latter problem, the cycle detection approach of [6, 4] uses a simple depth-first-search (DFS) strategy and, in contrast with [8], only needs a small part of the product automaton to be in main memory at any one time: the part corresponding to the computation that the depth-first-search is currently exploring. It implies that the protocol automaton may be constructed on-the-fly, i.e. as is needed, while checking for its emptiness. This means that, if the property does not hold, the algorithm can detect so after constructing and visiting only a small part of the state space

The automaton corresponding to the property can have as many as $2^{\mathcal{O}(n)}$ nodes where n is the number of subformulas in the property formula [13]. Thus, the size of the product automaton, which determines the overall complexity of the method is proportional to $N \cdot 2^{\mathcal{O}(n)}$, where N is the number of (reachable) protocol states. It is clearly desirable to keep property automata small and to avoid the exponential blowup that can occur in their construction whenever possible.

The standard automaton construction for a temporal logic property [13] (see also [16, 8]) is a global one and starts by generating a node for each (maximally consistent) set of subformulas of the property. While this is a simple way to describe the construction, it is clearly not a reasonable way to implement it, since it immediately realizes the worst case exponential complexity. A subsequent construction, proposed as a basis for an implementation [7], starts with a two state automaton that is repeatedly 'refined' until all models of the property are realized. Although the worst case remains exponential, this construction often achieves a substantial reduction in the number of generated nodes. On the other hand, the algorithm cannot be used on-the-fly during a depth-first search, as it repeatedly inspects the whole graph and "corrects" it by removing and adding edges and nodes. Moreover, the emptiness check proceeds by determining and inspecting the strongly connected components of the automaton and is thus less easily applicable to verifying whether a protocol satisfies a property. It should be said that the authors of [7] were not so much interested in protocol verification as in checking validity of a formula that include past operators.

In this paper we present, and describe experiments with, a pragmatic algorithm for constructing an automaton from a temporal logic formula. Though having its roots in the construction of [13], our algorithm is designed to yield small automata whenever possible and to be simple to implement. Furthermore, it proceeds on-the-fly in the sense that the automaton is only generated as needed during the verification process. Technically, the algorithm translates a propositional linear temporal logic formula into a Generalized Büchi automaton [4] using a very simple depth-first search. The interesting point is that, even though the algorithm produces a Generalized Büchi automaton, a simple transformation of this automaton yields a classical Büchi automaton for which the emptiness check can be done using a simple cycle detection scheme as in [4]. The result is that we obtain a protocol verification algorithm in which both the protocol and the property automata (and, hence, the product automaton) are constructed on-the-fly during a depth-first search that checks for emptiness.

The rest of the paper starts with some preliminaries defining temporal logic and its interpretations. Section 3 presents the basic algorithm, discusses optimizations and its application to model checking. The correctness proof occupies Section 4. In Section 5 we make some more detailed comparisons with existing constructions. The paper finishes with some experimental results and conclusions in Sections 6.

2 Preliminaries

The set of well-formed linear temporal logic (LTL) are constructed from a set of atomic propositions, the standard Boolean operators, and the temporal operators X and U. Precisely, given a finite set of propositions $\mathcal{P}$, formulas are defined inductively as follows:

- every member of $\mathcal{P}$ is a formula,
- if φ and ψ are formulas, then so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$, $\mathsf{X}\varphi$ and $\varphi \;\mathsf{U}\; \psi$.

An interpretation for a linear-time temporal logic formula is an infinite word $\xi = x_0 x_1 \cdots$ over the alphabet $2^{\mathcal{P}}$, i.e. a mapping from the naturals to $2^{\mathcal{P}}$. As made precise below, the elements of $2^{\mathcal{P}}$ are interpreted as assigning truth values to the elements of $\mathcal{P}$: elements in the set are assigned true, elements not in the set are assigned false. We write ξ_i for the suffix of ξ starting at x_i. The semantics of LTL is then the following.

- $\xi \models q$ iff $q \in x_0$, for $q \in \mathcal{P}$,
- $\xi \models \neg\varphi$ iff not $\xi \models \varphi$,
- $\xi \models \varphi \wedge \psi$ iff $\xi \models \varphi$ and $\xi \models \psi$,
- $\xi \models \varphi \vee \psi$ iff $\xi \models \varphi$ or $\xi \models \psi$,
- $\xi \models \mathsf{X}\varphi$ iff $\xi_1 \models \varphi$,
- $\xi \models \varphi \,\mathsf{U}\, \psi$ iff there is an $i \geq 0$ such that $\xi_i \models \psi$ and $\xi_j \models \varphi$ for all $0 \leq j < i$.

We introduce $\mathbf{T}$ as an abbreviation for $p \vee \neg p$, and $\mathbf{F}$ as an abbreviation for $\neg\mathbf{T}$. We also introduce additional temporal operators as abbreviations: $\mathsf{F}\varphi = \mathbf{T} \,\mathsf{U}\, \varphi$, $\mathsf{G}\varphi = \neg\mathsf{F}\neg\varphi$. Finally, we also use the temporal operator V which is defined as the dual of U: $\varphi \mathsf{V} \psi = \neg(\neg\varphi \,\mathsf{U}\, \neg\psi)$.

3 A Tableau Construction

Our goal is to build an automaton (transition system) that generates all infinite sequences satisfying a given temporal logic formula φ. The automata we build are generalized Büchi automata, namely Büchi automata with multiple sets of accepting states, as opposed to simple Büchi automata that have only one set of accepting states [11].

A generalized Büchi automaton [4] is a quadruple $\mathcal{A} = \langle Q, I, \longrightarrow, \mathcal{F}\rangle$, where Q is a finite set of *states*, $I \subseteq Q$ is the set of *initial states*, $\longrightarrow \subset Q \times Q$ is the *transition relation*, and $\mathcal{F} \subseteq 2^{2^Q}$ is a set of sets of accepting states $\mathcal{F} = \{F_1, F_2, \ldots F_n\}$. Notice that $\mathcal{F}$ can be empty.

An *execution* of $\mathcal{A}$ is an infinite sequence $\sigma = q_0\, q_1\, q_2 \ldots$ such that $q_0 \in I$ and, for each $i \geq 0$, $q_i \longrightarrow q_{i+1}$. An *accepting execution* σ is an execution such that, for each acceptance set $F_i \in \mathcal{F}$, there exists at least one state $q \in F_i$ that appears infinitely often in σ.

The automata we have defined so far have no input, and hence do not define any sequences. We thus need to add labels to our automata. The most common approach is to add labels to transitions. Here, we proceed slightly differently and add labels to states. A labeled generalized Büchi automaton, or LGBA for short, is a triple $\langle \mathcal{A}, \mathcal{D}, \mathcal{L}\rangle$, where $\mathcal{A}$ is a generalized Büchi automaton, $\mathcal{D}$ is some finite domain, and $\mathcal{L} : Q \to 2^{\mathcal{D}}$ is a *labeling function* from the states of $\mathcal{A}$ to subsets of the domain $\mathcal{D}$ (a state has a set of labels from $\mathcal{D}$). An LGBA *accepts* a word $\xi = x_0\, x_1\, x_2 \ldots$ from $\mathcal{D}^\omega$ iff there exists an accepting execution $\sigma = q_0\, q_1\, q_2 \ldots$ of $\mathcal{A}$ such that for each $i \geq 0$, $x_i \in \mathcal{L}(q_i)$. We also say that the execution σ accepts ξ.

The central part of the automaton construction algorithm is a tableau-like procedure related to the ones described in [14, 15]. The tableau procedure builds a graph, which will define the states and transitions of the automaton. The nodes of the graph are labeled by sets of formulas and are obtained by decomposing formulas according to their Boolean structure, and by expanding the temporal operators in order to separate what has to be true immediately from what has to be true from the next state on. The fundamental identity used to this is $\mu \mathsf{U} \psi \equiv \psi \vee (\mu \wedge \mathsf{X}(\mu \mathsf{U} \psi))$. Before describing the graph construction algorithm, we introduce the data structured used to represent the graph nodes.

3.1 The Data Structure

The data structure we use for representing graph nodes contains sufficient information for the graph construction algorithm to be able to operate in a DFS order. A *graph node* contains the following fields:

Name A string that is the name of the node.

Incoming The incoming edges represented by the names of the nodes with an outgoing edge leading to the current node. A special name, init is used to mark initial nodes. init is not the name of any node, hence does not represent a real edge.

New A set of temporal properties (formulas) that must hold at the current state and have not yet been processed.

Old The properties that must hold in the node and have already been processed. Eventually, *New* will become empty, leaving all the obligations in *Old*.

Next Temporal properties that must hold in all states that are immediate successors of states satisfying the properties in *Old*.

Father During the construction, nodes will be split. This field will contain the name of the node from which the current one has been split. This field is used for reasoning about the correctness of the algorithm only, and is not important for the construction.

We keep a list of nodes *Nodes_Set* whose construction was completed, each having the same fields as above. We denote the field *New* of the node q by $New(q)$, etc..

3.2 The algorithm

To simplify the representation of the algorithm, we assume first that the given formula φ for which the automaton should be built does not contain the Nextime operator 'X'. We will show later how to lift this restriction. Without loss of generality, we may further assume that the formula does not contain the operators 'F' and 'G', and that all the negations are pushed inside until they only precede propositional variables. That is, the formula is first transformed to contain only the operators U and V. In fact, the operator V, which is the dual of the operator 'U', was specifically introduced in order to allow pushing the negations without causing an exponential blowup in the size of the translated formula.

The line numbers in the following description refer to the algorithm that appears in Figure 1. The algorithm for translating the formula φ starts with a single node (lines 34–35). This node has a single (dummy) incoming edge, labeled init, to mark the fact that it is an initial node. Thus, by the end of the construction, a node will be initial iff it contains this label in its list of incoming nodes. It has initially one new obligation in *New*, namely, φ, and the sets *Old* and *Next* are initially empty. For example, the upper node in Figure 2 is the one with which the algorithm starts for constructing the automaton for $p \ \mathsf{U}\ q$.

With the current node *N*, the algorithm checks if there are unprocessed obligations left in *New* (line 4). If not, the current node is fully processed and ready to be added to *Nodes_Set*. If there already is a node in *Nodes_Set* with the same obligations in both its *Old* and *Next* fields (line 5), the copy that already exists needs only to be updated w.r.t. its set of incoming edges; the set of edges incoming to the new copy are added to the ones of the old copy in *Nodes_Set* (line 6).

If no such node exists in *Nodes_Set*, then the current node is added to this list, and a new current node is formed for its successor as follows (lines 8–10):

- There is initially one edge from N to the new current node.
- The set *New* is set initially to the *Next* field of N.
- The sets *Old* and *Next* of the new current node are initially empty.

When processing the current node, a formula η in *New* is removed from this list. In the case that η is a proposition or the negation of a proposition (a literal), then, if $\neg\eta$ is in *Old* (we identify $\neg\neg\eta$ with η), the current node is discarded, as it contains a contradiction (lines 16–17). Otherwise, η is added to *Old* (if it is not already there).

When η is not a literal, the current node can be split into two (lines 21–26) or not split (lines 29–31), and new formulas can be added to the fields *New* and *Next* (lines 22–23,25–26,30–31). The exact actions depend on the form of η and are the following:

$\eta = \mu \wedge \psi$ Then, both μ and ψ are added to *New* as the truth of both formula is needed to make η hold.

$\eta = \mu \vee \psi$ Then, the node is split, adding μ to *New* of one copy, and ψ to the other. These nodes correspond to the two ways in which η can be made to hold.

$\eta = \mu \ \mathsf{U}\ \psi$ Again, the node is split: for the first copy, μ is added to *New* and $\mu \ \mathsf{U}\ \psi$ to *Next*. For the other copy, ψ is added to *New*. This splitting is explained by observing that $\mu \ \mathsf{U}\ \psi$ is equivalent to $\psi \vee (\mu \wedge \mathsf{X}(\mu \ \mathsf{U}\ \psi))$. This is depicted in Figure 2.

$\eta = \mu \mathsf{V} \psi$ Then, the node is split: ψ is added to *New* of both copies, μ is added to *New* of one copy, and $\mu \mathsf{V} \psi$ is added to *Next* of the other. This splitting is explained by observing that $\mu \mathsf{V} \psi$ is equivalent to $\psi \wedge (\mu \vee \mathsf{X}(\mu \mathsf{V} \psi))$.

The copies are processed in DFS order, i.e., when expansion of the current node and its successors are finished, the expansion of the second copy and its successors is started.

The algorithm is listed in Figure 1 in a pseudo-code language. The function **new_name()** generates a new string for each successive call. The function **Neg**, is defined as follows: **Neg**(P_n)=$\neg P_n$, **Neg**($\neg P_n$)=P_n, and similarly for the boolean constants T and F. The functions **New1**(η), **New2**(η) and **Next1**(η) are defined in the following table:

η	**New1**(η)	**Next1**(η)	**New2**(η)
$\mu \mathsf{U} \psi$	$\{\mu\}$	$\{\mu \mathsf{U} \psi\}$	$\{\psi\}$
$\mu \mathsf{V} \psi$	$\{\psi\}$	$\{\mu \mathsf{V} \psi\}$	$\{\mu, \psi\}$
$\mu \vee \psi$	$\{\mu\}$	$\emptyset$	$\{\psi\}$

3.3 Using the Automaton for Automatic Protocol Verification

The graph constructed by the algorithm in Section 3.2 can now be used to define an LGBA accepting the infinite words satisfying the formula. The set of states Q will be the nodes returned by the algorithm. Notice that only nodes for which New is empty are placed in this set. In other words, only fully expanded nodes are returned. The initial states I are those nodes q such that $\mathsf{init} \in Incoming(q)$. The transitions $p \longrightarrow q$ are exactly those satisfying that $p \in Incoming(q)$.

The domain $\mathcal{D}$ is $2^{\mathcal{P}}$ and the label of a node q is all sets in $2^{\mathcal{P}}$ that are compatible with $Old(q)$. Indeed, a node of the graph does not necessarily assign truth values to all atomic propositions, and the label of a node can be any element of $2^{\mathcal{P}}$ that agrees with the literals that appear in $Old(q)$. Precisely, let $Pos(q)$ be $Old(q) \cap \mathcal{P}$ and $Neg(q)$ be $\{\eta \mid \neg\eta \in Old(q) \wedge \eta \in \mathcal{P}\}$, i.e., $Pos(q)$ and $Neg(q)$ are the positive and negative occurrences of the propositions in q, respectively. Then, $\mathcal{L}(q) = \{X | X \subseteq \mathcal{P} \wedge X \supseteq Pos(q) \wedge X \cap Neg(q) = \emptyset\}$.

Finally, we have to impose accepting conditions. Indeed, observe that not every maximal path $\sigma = q_0\, q_1 \cdots$ in the graph determines models of the formula: the construction allows some node to contain $\mu \mathsf{U} \psi$ while none of the successor nodes contain ψ. This is solved by imposing the generalized Büchi acceptance conditions. For each subformula of φ of the type $\mu \mathsf{U} \psi$, there will be a set $F \in \mathcal{F}$ which includes the nodes $q \in Q$ such that either $\mu \mathsf{U} \psi \notin Old(q)$, or $\psi \in Old(q)$.

Let us show that, with these acceptance conditions, one can no longer accept a sequence in which $\mu \mathsf{U} \psi$ appears from some node q_i onwards without ψ occurring later. First, notice that from the construction, if $\mu \mathsf{U} \psi \in Old(q_i)$ and $\psi \notin Old(q_{i+1})$, then $\mu \mathsf{U} \psi \in q_i$ and $\psi \notin q_{i+1}$, then $\mu \mathsf{U} \psi \in Old(q_{i+1})$. Thus, in the above scenario, $\mu \mathsf{U} \psi$ propagates from q_i onwards, since ψ never occurs. Let $F \in \mathcal{F}$ be the accepting subset that is associated with $\mu \mathsf{U} \psi$. Then, none of the states with index greater or equal to i can be in F. But then the sequence σ does not contain infinitely many occurrences of any state from F, and is not accepting.

As explained in the introduction, a protocol is verified w.r.t. a property by constructing an automaton for the *negation* of the property, and by exploring the synchronous product of the protocol and the property automaton for emptiness. Since the automaton representing the

```
record graph_node = [Name:string, Father:string, Incoming:set of string,
      New:set of formula, Old:set of formula, Next:set of formula];

function expand (Node, Nodes_Set)
   if New(Node)=∅ then
      if ∃ND ∈Nodes_Set with Old(ND)=Old(Node) and Next(ND)=Next(Node)
      then Incoming(ND) = Incoming(ND)∪Incoming(Node);
         return(Nodes_Set);
      else return(expand([Name⇐Father ⇐new_name(),
         Incoming⇐ {Name(Node)}, New⇐Next(Node),
         Old⇐ ∅, Next⇐ ∅], {Node}∪Nodes_Set))
   else
      let η ∈New;
      New(Node) := New(Node)\{η};
      case η of
      η = P_n, or ¬P_n or η =T or η =F=>
         if η =F or Neg(η) ∈Old(Node) (* Current node contains a contradiction *)
            then return(Nodes_Set) (* Discard current node *)
            else Old(Node):=Old(Node)∪{η};
            return(expand(Node, Nodes_Set));
      η = μ U ψ, or μVψ, or μ ∨ ψ =>
         Node1:=[Name⇐new_name(), Father⇐ Name(Node), Incoming⇐Incoming(Node),
            New⇐New(Node)∪({New1(η)}\Old(Node)),
            Old⇐Old(Node)∪{η}, Next=Next(Node)∪{Next1(η)} ];
         Node2:=[Name⇐new_name(), Father⇐Name(Node), Incoming⇐Incoming(Node),
            New⇐New(Node)∪({New2(η)}\Old(Node)),
            Old⇐Old(Node)∪{η}, Next⇐Next(Node)];
         return(expand(Node2, expand(Node1, Nodes_Set)));
      η = μ ∧ ψ =>
         return(expand([Name⇐ Name(Node), Father⇐Father(Node),
            Incoming⇐Incoming(Node), New⇐New(Node)∪({μ, ψ}\Old(Node)),
            Old⇐Old(Node)∪{η}, Next=Next(Node)], Nodes_Set))
end expand;

function create_graph (φ)
   return(expand([Name⇐Father⇐new_name(), Incoming⇐{init},
      New⇐ {φ}, Old⇐ ∅, Next⇐ ∅], ∅))
end create_graph;
```

Figure 1: The algorithm

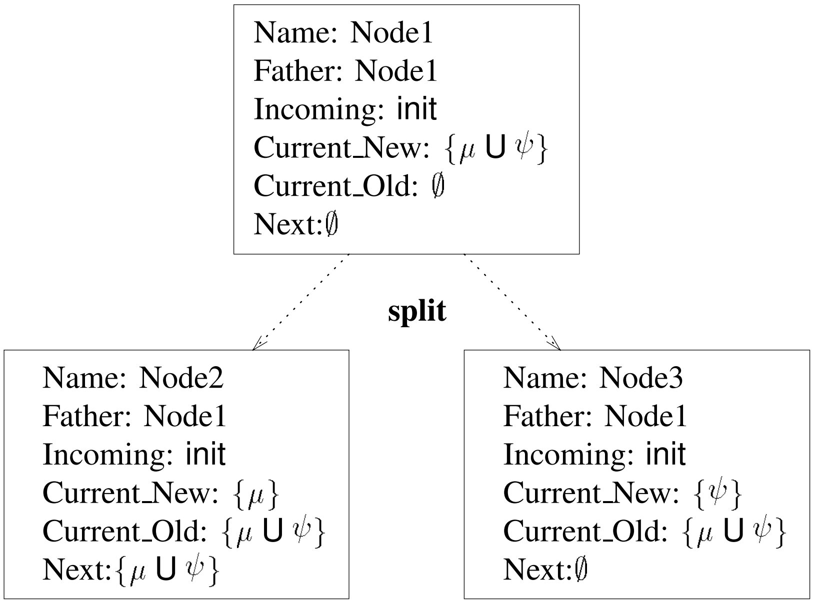

Figure 2: Splitting a node

protocol has an empty acceptance condition ($\mathcal{F} = \emptyset$), the product automaton simply inherits the accepting sets of the property automaton.

Checking for emptiness can be done on-the-fly, i.e., during the generation of the product. For a simple Büchi automaton (one for which $\mathcal{F}$ is a singleton), one only needs to find a reachable accepting state that is also reachable from itself. An algorithm for doing this is described in [4]. Furthermore, that paper also shows how generalized Büchi conditions can also be handled. The idea is to transform a generalized Büchi automaton into a simple one. This is done by using a counter: each state becomes a pair $\langle q, i\rangle$ where i is a counter. The counter is initialized to 0 and counts modulo n, where $n = |\mathcal{F}|$. It is updated from i to $i + 1$ whenever one reaches an element of the ith set $F_i \in \mathcal{F}$. One then only needs one set of accepting states, for instance $F_0 \times \{0\}$.

3.4 Improvements to the Basic Algorithm

Adding Nextime Formulas All that is needed to be able to handle formulas involving the Nextime operator (X) is to add an extra case to the algorithm.

$\eta = \mathsf{X}\mu$ =>
  return(expand([*Name*⇐*Name(Node)*, *Father*⇐*Father(Node)*,
    Incoming⇐*Incoming(Node)*, *New*⇐*New(Node)*, *Old*⇐*Old(Node)*$\cup\{\eta\}$,
    Next⇐*Next(Node)*$\cup\{\mu\}$], *Nodes_Set*))

Pure "On-the-fly" Construction. The algorithm presented here generates an LGBA that can be used for model-checking or checking the validity of a temporal formula. However, one does not have to complete the construction of this automaton in order to do the model-checking. Construction of nodes can be done "on-demand", while intersecting them with the protocol automaton. Then, when the successors of a node in the property automaton are constructed, one does not immediately continue to construct their own successors, and so forth. Instead, one chooses the successors that can match the current state of the protocol. Thus, it is possible that a violation of the checked property will be discovered before generating the entire property automaton.

Improving the Efficiency. The algorithm as presented here was written in such a way that its proof of correctness will be simplified. Therefore, it contains some redundancies. The following improvements can be made:

- The field *Father* is not needed, except for the proof of correctness.
- When splitting a node (lines 21–26), there is no need to generate two new nodes; instead one can update one of them with additional information, and after generating all its descendents, create the other one. This is also true when adding the conjuncts to a node (lines 28–30).
- An eventuality of the form $\eta U \mathbf{T}$ does not generate a set $F \in \mathcal{F}$. Indeed, such a formula is equivalent to $\mathbf{T}$.
- Inconsistencies are only detected at the level of atomic propositions so that nodes that are semantically inconsistent may still appear in the automaton. Certain inconsistencies can be detected earlier using syntactic means. For instance, before adding a formula μ to a node one can 'compute' $\neg\mu$ (by pushing the negation inside) and check whether it already occurs. If it occurs, the current node is abandoned.
- Every processed formula is currently stored in the *Old* field. This is not always necessary. For instance, after a conjunction $\mu_1 \wedge \mu_2$ has been analyzed, it need not be added to the *Old* field because both μ_1 and μ_2 will be added, and the presence of these formula tells us that the conjunction will also be true in this node. Note however that, if $\mu_1 \wedge \mu_2$ is the righthand argument of an Until subformula $\eta \mathsf{U} \psi$, it must still be stored, since it is used to define the acceptance conditions. Similar observations apply to disjunctions, U and V formulas, but care must be also taken to retain the information needed for identifying the acceptance conditions, i.e. the righthand arguments of U formulas. As a consequence, the generated automata may become smaller, since nodes that differed previously might become identical.
- In the case of treating at line 20 a subformula of the type $\mu \mathsf{U} \psi$, if ψ already appears in $New(q) \cup Old(q)$, then there is no need to split the node q into two. It is then sufficient to move the subformula $\mu \mathsf{U} \psi$ from $New(q)$ to $Old(q)$. The same holds when treating a formula of the type $\mu \mathsf{V} \psi$, and both ψ and μ are in $New(q) \cup Old(q)$.

4 Proof of Correctness

In this section, the proof of correctness will be sketched. The main theorem is the following:

Theorem 4.1 *The automaton $\mathcal{A}$ constructed for a property φ accepts* exactly *the sequences over $(2^{\mathcal{P}})^{\omega}$ that satisfy φ.*

Proof. The two directions are proved in Lemma 4.8 and Lemma 4.9 below. ∎

Let $\Delta(q)$ denote the value of $Old(q)$ at the point where the construction of the node q is finished, i.e. when it is added to *Nodes_Set*, at line 10 of the algorithm., Let $\bigwedge \Xi$ denote the conjunction of a set of formulas Ξ, the conjunction of the empty set being taken equal to T.

Let $\xi = x_0x_1x_2\ldots$ be a *propositional sequence*, i.e., a sequence over $(2^{\mathcal{P}})^{\omega}$, and let $\sigma = q_0q_1q_2\ldots$ be a sequence of states of $\mathcal{A}$ such that for each $i \geq 0$, $q_i \longrightarrow q_{i+1}$. Recall that ξ_i denotes the suffix of the sequence ξ, i.e., $x_ix_{i+1}x_{i+2}\ldots$.

Lemma 4.1 *Let σ be an execution of $\mathcal{A}$, and let $\mu \,\mathsf{U}\, \eta \in \Delta(q_0)$. Then one of the following holds:*

1. $\forall i \geq 0 :\ \mu,\ \mu \,\mathsf{U}\, \eta \in \Delta(q_i)$ *and* $\eta \notin \Delta(q_i)$.
2. $\exists j \geq 0 \forall i\, 0 \leq i < j :\ \mu,\ \mu \,\mathsf{U}\, \eta \in \Delta(q_i)$ *and* $\eta \in \Delta(q_j)$.

Proof. Follows directly from the construction. ∎

Lemma 4.2 *When a node q is split during the construction in lines 21–26 into two nodes q_1 and q_2, the following holds:*

$$(\bigwedge \mathrm{Old}(q) \wedge \bigwedge \mathrm{New}(q) \wedge \mathsf{X} \bigwedge \mathrm{Next}(q)) \longleftrightarrow$$
$$((\bigwedge \mathrm{Old}(q_1) \wedge \bigwedge \mathrm{New}(q_1) \wedge \mathsf{X} \bigwedge \mathrm{Next}(q_1)) \vee (\bigwedge \mathrm{Old}(q_2) \wedge \bigwedge \mathrm{New}(q_2) \wedge \mathsf{X} \bigwedge \mathrm{Next}(q_2)))$$

Similarly, when a node q is updated to become a new node q', as in lines 28–31, the following holds:

$$(\bigwedge \mathrm{Old}(q) \wedge \bigwedge \mathrm{New}(q) \wedge \mathsf{X} \bigwedge \mathrm{Next}(q)) \longleftrightarrow (\bigwedge \mathrm{Old}(q') \wedge \bigwedge \mathrm{New}(q') \wedge \mathsf{X} \bigwedge \mathrm{Next}(q'))$$

Proof. Directly from the algorithm and the definition of LTL. ∎

Using the field $Father$ we can link each node to the one from which it was split. This defines an ancestor relation R, where $(p, q) \in R$ iff $Father(q) = Name(p)$. Let R^* be the transitive closure of R. Nodes q such that $Father(q) = Name(q)$, i.e., $(p, p) \in R$ are called *rooted*. A rooted node p can be one of the following two:

1. p is the initial node with which the search started at lines 34–35. Thus, it has $New(p) = \{\varphi\}$.
2. p is obtained at lines 8–9 from some node q whose construction is finished. Thus, we have $New(p)$ set to $Next(q)$.

Let $first(q)$ be the node p such that $(p, q) \in R^*$, and $(p, p) \in R$.

Lemma 4.3 *Let p be a rooted node, and q_1, q_2, $\ldots q_n$ be all its same-time descendant nodes, i.e. the nodes q_i such that $(p, q_i) \in R^*$. Let Ξ be the set of formulas that are in* $\mathrm{New}(p)$*, when it is created. Let* $\mathrm{Next}(q_i)$ *be the values of the fields* Next *for q_i at the end of the construction. Then, the following holds:*

$$\bigwedge \Xi \longleftrightarrow \bigvee_{1 \le i \le n} (\bigwedge \Delta(q_i) \wedge \mathsf{X} \bigwedge \mathrm{Next}(q_i))$$

Moreover, if $\xi \models \bigvee_{1 \le i \le n}(\bigwedge \Delta(q_i) \wedge \mathsf{X} \bigwedge \mathrm{Next}(q_i))$, then there exists some $1 \le i \le n$ such that $\xi \models \bigwedge \Delta(q_i) \wedge \mathsf{X} \bigwedge \mathrm{Next}(q_i)$ such that for each $\mu \,\mathsf{U}\, \eta \in \Delta(q_i)$ with $\xi \models \eta$, η is also in $\Delta(q_i)$.

Proof. By induction on the construction, using Lemma 4.2. ∎

Lemma 4.4 *Let ξ be a propositional sequence such that $\xi \models \bigwedge \Delta(q) \wedge \mathsf{X} \bigwedge \mathrm{Next}(q)$. Then, there exists a transition $q \longrightarrow q'$ in $\mathcal{A}$ such that $\xi_1 \models \bigwedge \Delta(q') \wedge \mathsf{X} \bigwedge \mathrm{Next}(q')$. Moreover, let $\Gamma = \{\eta \mid \mu \,\mathsf{U}\, \eta \in \Delta(q)$ and $\eta \notin \Delta(q)$ and $\xi_1 \models \eta\}$, then in particular there exists a transition $q \longrightarrow q'$ such that q' satisfies also that $\Gamma \subseteq \Delta(q')$.*

Proof. When the construction of node q was finished, a node r with $New(r) = Next(q) = \Xi$ was generated. Then, Lemma 4.3 guarantees that a successor as required exists. ∎

Lemma 4.5 *For every initial state $q \in I$ of an automaton $\mathcal{A}$ generated from the formula φ, we have $\varphi \in \Delta(q)$.*

Proof. Immediately from the construction. ∎

Lemma 4.6 *Let $\mathcal{A}$ be an automaton constructed for the LTL property φ. Then*

$$\varphi \leftrightarrow \bigvee_{q \in I} (\bigwedge \Delta(q) \wedge \mathsf{X} \bigwedge \mathrm{Next}(q)) \,.$$

Proof. From Lemma 4.3, since Ξ in that Lemma is initially $\{\varphi\}$. ∎

Lemma 4.7 *Let $\sigma = q_0 q_1 q_2 \ldots$ be a run of $\mathcal{A}$ that accepts the propositional sequence ξ when q_0 is taken to be an initial state. Then $\xi \models \bigwedge \Delta(q_0)$.*

Proof. By induction on the size of the formulas. The base case is for formulas of the form P, $\neg P$, where $P \in \mathcal{P}$. We will show only the case of $\mu \,\mathsf{U}\, \eta \in \Delta(q_0)$. Then, according to Lemma 4.1 there are two cases:

1. $\forall i \ge 0 :\ \mu,\ \mu \,\mathsf{U}\, \eta \in \Delta(q_i)$ and $\eta \notin \Delta(q_i)$.
2. $\exists j \ge 0 \forall i\, 0 \le i < j :\ \mu,\ \mu \,\mathsf{U}\, \eta \in \Delta(q_i)$ and $\eta \in \Delta(q_j)$.

Since σ satisfies the acceptance conditions of $\mathcal{A}$, only case 2 is possible. But then, by the induction hypothesis, $\xi_j \models \eta$ and for each $0 \le i < j$, $\xi_i \models \mu$. Thus, by the semantic definition of LTL, $\xi \models \mu \,\mathsf{U}\, \eta$. The other cases are treated similarly. ∎

Lemma 4.8 *Let σ be an execution of the automaton $\mathcal{A}$, constructed for φ, that accepts the propositional sequence ξ. Then $\xi \models \varphi$.*

Proof. The node q_0 is now an initial state, i.e., in I. From Lemma 4.7 it follows that $\xi \models \bigwedge \Delta(q_0)$. By Lemma 4.5, if $q_0 \in I$ then $\varphi \in \Delta(q_0)$. Thus, $\xi \models \varphi$. ∎

Lemma 4.9 *Let $\xi \models \varphi$. Then there exists an execution σ of $\mathcal{A}$ that accepts ξ.*

Proof. First, by Lemma 4.6, there exists a node $q_0 \in I$ such that $\xi \models \bigwedge \Delta(q_0) \wedge \mathsf{X} \bigwedge Next(q_0)$. Now, one can construct the propositional sequence σ by repeatedly using Lemma 4.4. Namely, if $\xi_i \models \bigwedge \Delta(q_i) \wedge \mathsf{X} \bigwedge Next(q_i)$, then choose q_{i+1} to be a successor of q_i that satisfies $\xi_{i+1} \models \bigwedge \Delta(q_{i+1}) \wedge \mathsf{X} \bigwedge Next(q_{i+1})$. Furthermore, Lemma 4.4 also guarantees that we can choose q_{i+1} such that if for an U subformula $\mu \ \mathsf{U} \ \eta$ in $\Delta(q_i)$, η holds in ξ_{i+1}, then $\eta \in \Delta(q_{i+1})$. We also know from Lemma 4.1 that $\mu \ \mathsf{U} \ \eta$ will propagate to the successors of q_i unless η holds. Since $\xi_i \models \mu \ \mathsf{U} \ \eta$, there must be some minimal $j \geq i$ such that $\xi_j \models \eta$. hence by the above, $\eta \in \Delta(q_j)$. ∎

5 Comparison with Previous Work

The first translation from an LTL formula φ to a Büchi automaton was by Wolper, Vardi and Sistla [16, 13]. It is based on constructing the intersection of two automata. The first automaton takes care of the state-to-state consistency of the runs, and is called the *local automaton*. The other automaton, called the *eventuality automaton*, takes care that the eventualities i.e., subformulas of the type $\mu \ \mathsf{U} \ \psi$, will be satisfied. The set of formulas $cl(\varphi)$ are the subsets of φ. Then, each state A of the local automaton consists of the formulas from $cl(\varphi)$, either negated, or non-negated. The transitions of the local automaton reflect consistency conditions. E.g., if $p \longrightarrow q$, i.e., q is a possible successor of p, and $\mathsf{X}P$ belongs to node p, then P must belong to node q. The edges of this automaton are labeled identically to the nodes from which they emanate. The initial states of the local automaton are the ones that contain the formula φ itself.

The second automaton's states consists of a subset of U subformulas of φ . These are the set of goals that need to be satisfied along the execution sequence. The edges are labeled as in the local automaton. Once the righthand subformula of a U formula (i.e., ψ in $\varphi \ \mathsf{U} \ \psi$) appears on an edge, the U formula is removed from the set of goals (i.e., does not appear in the set of formulas of the next state). When all the goals are achieved, one starts with a new set of goals accumulated in the labels of the edge (which will later be linked to the goals accumulated in the state of the local automaton). The eventuality automaton accepts a word whenever all the goals are achieved infinitely often. The combination of the two automata is done by taking the Cartesian product of the node sets, and coordinating the edges. The acceptance condition of the product is fixed by the eventuality automaton: any node that has (in its second component) an empty set of goals is accepting.

This construction was meant first of all to show the theoretical connection between LTL and Büchi automata and establish its correctness. It was also designed to be applicable to temporal logic extended with operators defined by finite automata [14]. Applied blindly, it systematically leads to an automaton with a state set of exponential size for the following reasons.

1. Each node in the local automaton of this construction is maximal. Namely, it contains each subformula either negated or non-negated. Thus the number of nodes is exponential in the size of the formula (or equivalently in the number of its subformulas, $cl(\varphi)$). This

is unnecessary since many of theses nodes are often unreachable. Furthermore, this approach does not allow nodes that only differ on locally irrelevant members of $cl(\varphi)$ to be merged.

2. The eventuality automaton has states that consist of sets of U subformulas. Thus, it is exponential in the number of U subformulas. This is needed to handle extended temporal logics, but is not necessary for the logic we consider here. Indeed, since U formulas propagate unmodified until their righthand side argument is satisfied, one can, as we did here, directly write the requirement that U subformulas are satisfied as a generalized Büchi acceptance condition. Furthermore, converting this generalized Büchi acceptance condition to a simple one can de done with an increase in the size of the automaton that is linear in the number of U subformulas, rather than exponential in this number as in the eventuality automaton approach. (A similar observation is independently and implicitly made in [2].)

3. The nodes are generated in a "global" manner: first, all possible nodes are generated for both automata. Then, edges are constructed between pairs of nodes if they satisfy some consistency conditions. Finally, the product automaton is taken. Only at the end it is possible to check which nodes are really reachable from the initial states. This requires an additional search.

An improved tableau construction for temporal logic was given in [7]. It constructs a graph (the goal of that paper was checking satisfiability rather than using the translation for model-checking), but can similarly create the Büchi automaton that corresponds to a temporal property. This construction indeed uses the above observations to reduce the number of states and edges. It is also claimed that it operates "on-the-fly", as it starts with the property that needs to be translated, creating an initial graph, and then refining this graph until it corresponds to the appropriate translation. Thus, it constructs nodes and edges only "when needed". This construction globally checks pairs of adjacent nodes in the graph. If they do not satisfy the tableau consistency conditions, one of these nodes is refined: it is replaced by a set of nodes that satisfy the consistency conditions. The algorithm continues to refine nodes until all the edges satisfy the consistency conditions. This involves replacing old nodes by new ones, and adding and removing edges accordingly. With this algorithm, the construction of the automaton needs to be finished before it can be used for model-checking.

Our construction starts with the checked formula φ, constructs a node for it and continue to generate the graph in a depth-first-search order. The only cases where a node is discarded are where it is already found in the list of existing nodes, or when it contains a propositional contradiction. Moreover, it can be used on-the-fly. Thus, avoiding the need to construct the entire automaton if a violation of the checked property was found during its intersection with the protocol.

6 Experimental Results and Conclusions

The following table compares the global construction described in [13] and the algorithm described in Section 3. Both were implemented in Standard ML of New Jersey. Here, $\mathsf{F}p$ abbreviates $\mathbf{T}\ \mathsf{U}\ p$ and $\mathsf{G}p$ abbreviates $\neg\mathsf{F}\neg p$.

Num.	Formula	Global Construction Nodes	Transitions	New Construction Nodes	Transitions	Accepts
1	$p_1 \mathsf{U}\, p_2$	8	34	3	4	1
2	$p_1 \mathsf{U}\, (p_2 \mathsf{U}\, p_3)$	26	240	4	6	2
3	$\neg(p_1 \mathsf{U}\, (p_2 \mathsf{U}\, p_3))$	26	240	7	15	0
4	$\mathsf{GF}p_1 \rightarrow \mathsf{GF}p_2$	114	763	9	15	2
5	$\mathsf{F}p_1 \mathsf{U}\, \mathsf{G}p_2$	56	337	8	15	2
6	$\mathsf{G}p_1 \mathsf{U}\, p_2$	13	63	5	6	1
7	$\neg(\mathsf{FF}p_1 \leftrightarrow \mathsf{F}p_1)$	-	-	22	41	2

The rightmost column represents the number of pairs in the acceptance table of the constructed automaton. Notice that for the safety property 3, there are no U subformulas satisfy. Yet, for the automaton to be nonempty, it has to contain a reachable cycle.

The formulas that were used in the experiments are the following.

$\mathsf{GF}p_1 \longrightarrow \mathsf{GF}p_2$ This formula can describe a fairness condition: p_1 expresses the enabledness of some element (e.g., a process, a transition), and p_2 the execution of that element. Such a formula can be exploited when one wants to check some property under a fairness condition which is not already implemented in the model-checker.

$p_1 \mathsf{U}\, (p_2 \mathsf{U}\, p_3)$ and $\neg(p_1 \mathsf{U}\, (p_2 \mathsf{U}\, p_3))$ The purpose of these examples is to show that the construction does not impose an exponential blowup when negating a formula.

$\neg(\mathsf{FF}p_1 \leftrightarrow \mathsf{F}p_1)$ This can be used to verify that $(\mathsf{FF}p_1 \leftrightarrow \mathsf{F}p_1)$ is a tautology. Unfortunately there was insufficient memory for the ML program for the global construction to complete.

It is evident from the table that the exponential blowup occurs much faster using the global construction. This will not only be reflected in the memory and time that it takes to complete the construction, but also during the emptiness check, which takes time (linearly) proportional to the size of the constructed automaton.

In model-checking the size of the constructed property automaton is more critical, since one has to take the product of this automaton with the one representing the state space. Given that the size of the state space is itself also often a problem, it is all the more important that the property automaton be as small as possible.

For the same reason, the fact that the algorithm is on-the-fly is important. It means that the algorithm can often given an answer before the full state space and property automaton have been constructed.

Thus, we feel that the algorithm in this paper is a promising and potentially practical approach to both model-checking and validity checking: it is simple, it appears to produce reasonable sized automata and it operates on-the-fly.

Acknowledgment. The second author likes to thank Elsa Gunter for helping him with debugging the ML program.

References

[1] Y. Choueka, Theories of automata on ω-Tapes: a simplified approach, Journal of Computer and System Science 8 (1974), 117-141.

[2] G. Bhat, R. Cleaveland, O. Grumberg, Efficient on-the-fly model checking for CTL*, Proceedings of the 10*th* Symposium on Logic in Computer Science, 1995, San Diego, CA, To appear.

[3] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L.Dill, J. Hwang, Symbolic model checking: 10^{20} states and beyond, Information and Computation, 98(1992), 142–170.

[4] C. Courcoubetis, M. Vardi, P. Wolper, M. Yannakakis, Memory-efficient algorithms for the verification of temporal properties, Formal methods in system design 1 (1992) 275–288.

[5] O. Coudert, C. Berthet, J.C. Madre, Verification of synchronous sequential machines based on symbolic execution, Automatic Verification Methods for Finite State Systems, Grenoble, France, LNCS 407, Springer–Verlag, 1989, 365–373.

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

[7] Y. Kesten, Z. Manna, H. McGuire, A. Pnueli, A decision algorithm for full propositional temporal logic, CAV'93, Elounda, Greece, LNCS 697, Springer–Verlag, 97-109.

[8] O. Lichtenstein, A. Pnueli, Checking that finite-state concurrent programs satisfy their linear specification, $11th$ ACM POPL, 1984, 97–107.

[9] A. Pnueli, The temporal logic of programs, Proceedings of the 18*th* IEEE Symposium on Foundation of Computer Science, 1977, 46-57.

[10] A. P. Sistla, E. M. Clarke, The Complexity of propositional linear temporal logics, Journal of the ACM, 32(1985), 733–749.

[11] W. Thomas, Automata on infinite objects, Handbook of theoretical computer science, 1990, 165–191.

[12] M.Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, Proceedings of the 1*st* Symposium on Logic in Computer Science, 1986, Cambridge, England, 322–331.

[13] M.Y. Vardi, P. Wolper, Reasoning about infinite computations, Information and Computation, 115(1994), 1–37.

[14] P. Wolper, Temporal logic can be more expressive, Information and Control, 56(1983), 72–99.

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

[16] P. Wolper, M.Y. Vardi, A.P. Sistla, Reasoning about infinite computation paths, Proceedings of $24th$ IEEE symposium on foundation of computer science, Tuscan, 1983, 185–194.