Improved Automata Generation for Linear Temporal Logic*

Marco Daniele[1,2], Fausto Giunchiglia[3,2], and Moshe Y. Vardi[4]

[1] Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", 00198 Roma, Italy
[2] Istituto Trentino di Cultura
Istituto per la Ricerca Scientifica e Tecnologica
38050 Povo, Trento, Italy
[3] Dipartimento di Informatica e Studi Aziendali
Università di Trento, 38100 Trento, Italy
[4] Department of Computer Science
Rice University, Houston TX 77251, USA

Abstract. We improve the state-of-the-art algorithm for obtaining an automaton from a linear temporal logic formula. The automaton is intended to be used for model checking, as well as for satisfiability checking. Therefore, the algorithm is mainly concerned with keeping the automaton as small as possible. The experimental results show that our algorithm outperforms the previous one, with respect to both the size of the generated automata and computation time. The testing is performed following a newly developed methodology based on the use of randomly generated formulas.

1 Introduction

This paper focuses on the explicit-state automata-based approach to model checking of linear temporal logic specifications [VW86,VW94,Hol97]. In this approach, both the system and the *negation* of the specifications are turned into automata on infinite words [Tho90]. The former automaton recognizes the system execution sequences, while the latter one comprises all the execution sequences (models) violating the specifications. Verification amounts then to checking whether the language recognized by the synchronous product of the above automata is empty. Similarly, satisfiability checking amounts to checking that the language recognized by the automaton built for the formula to be checked is non-empty. Satisfiability also plays an important role in model checking, for avoiding model checking unsatisfiable or valid specifications.

The automaton for the specifications can have as many as $2^{\mathcal{O}(n)}$ states, where n is the number of subformulas of the specifications [VW94]. Therefore, the size

* Supported in part by NSF grants CCR-9628400 and CCR-9700061, and by a grant from the Intel Corporation. Part of this work was done while the first author was a visiting student and the third author was a Varon Visiting Professor at the Weizmann Institute of Science.

N. Halbwachs and D. Peled (Eds.): CAV'99, LNCS 1633, pp. 249–260, 1999.

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 system states. For these reasons, it is clearly desirable to keep the specification automaton as small as possible, and to work on-the-fly, that is, to detect that a system violates its specifications by constructing and visiting only some part of the search space containing the bug. Note that even though in practice the assertions being verified are typically expressed by short formulas, it is often impossible to verify these properties without making some assumptions on the environment of the system being verified. Thus, in practice, one typically model checks formulas of the form $\phi \rightarrow \psi$, where the assertion ψ may be quite simple, but the assumption ϕ may be rather complicated.

The state-of-the-art on-the-fly algorithms for turning specifications into automata and performing the emptiness check can be found in [CVWY91] and [GPVW95]. Such algorithms define the kernel of the model checker SPIN [Hol97]. We refer to the algorithm described in [GPVW95] as GPVW. That paper also discusses several possible improvements. We refer to the improved algorithm as GPVW+. An alternative automata construction for temporal specifications [KMMP93] starts with a two-state automaton that is repeatedly "refined" until all models of the specifications are realized. Due to this refinement process, however, this algorithm can not be used in an on-the-fly fashion. Another approach could be turning the on-the-fly decision procedure presented in [Sch98] into a procedure for automata construction. It is not clear, however, whether and how this modification could be done, for that procedure is geared towards finding and representing one model, but not all models.

In this paper we present, and describe experiments with, an algorithm for building an automaton from a linear temporal logic formula. Our algorithm, hereafter LTL2AUT, though being based on GPVW+, is geared towards building smaller automata in less time. Our improvements are based on simple syntactic techniques, carried out on-the-fly when states are processed, that allow us to eliminate the need of storing some information. Experimental results demonstrate that GPVW+ significantly outperforms GPVW and show that LTL2AUT further outperforms GPVW+, with respect to both the size of the generated automata and computation time. The testing has been performed following a newly developed methodology, which, inspired by the methodologies proposed in [MSL92] and [GS96] for propositional and modal K logics, is based on randomly generated formulas.

The rest of the paper is structured as follows. Section 2 introduces linear temporal logic and automata on infinite words. Section 3 presents the core underlying GPVW, GPVW+, and LTL2AUT, and Section 4 shows how GPVW, GPVW+, and LTL2AUT can be obtained by suitably instantiating such core. The test is divided between Section 5, where our test method is discussed, and Section 6, where a comparison of the three algorithms is given. Finally, we make some concluding remarks in Section 7.

2 Preliminaries

The set of Linear Temporal Logic formulas (LTL) is defined inductively starting from a finite set $\mathcal{P}$ of *propositions*, the standard Boolean operators, and the temporal operators X ("next time") and $\mathcal{U}$ ("until") as follows:

- each member of $\mathcal{P}$ is a formula,
- $\neg\mu_1$, $\mu_1 \vee \mu_2$, $\mu_1 \wedge \mu_2$, $X\mu_1$, and $\mu_1 \mathcal{U} \mu_2$ are formulas, if so are μ_1 and μ_2.

An LTL interpretation is a function $\xi : N \to 2^{\mathcal{P}}$, i.e., an infinite word over the alphabet $2^{\mathcal{P}}$, which maps each instant of time into the propositions holding at such instant. We write ξ_i for denoting the interpretation $\lambda t.\xi(t+i)$. LTL semantics is then defined inductively as follows:

- $\xi \models p$ iff $p \in \xi(0)$, for $p \in \mathcal{P}$,
- $\xi \models \neg\mu_1$ iff $\xi \not\models \mu_1$,
- $\xi \models \mu_1 \vee \mu_2$ iff $\xi \models \mu_1$ or $\xi \models \mu_2$,
- $\xi \models \mu_1 \wedge \mu_2$ iff $\xi \models \mu_1$ and $\xi \models \mu_2$,
- $\xi \models X\mu_1$ iff $\xi_1 \models \mu_1$,
- $\xi \models \mu_1 \mathcal{U} \mu_2$ iff there exists $i \geq 0$ such that $\xi_i \models \mu_2$ and, for all $0 \leq j < i$, $\xi_j \models \mu_1$.

As usual, we have $\neg\neg\mu \doteq \mu$, $\mathrm{T} \doteq p \vee \neg p$ and $\mathrm{F} \doteq \neg\mathrm{T}$. Moreover, we define $\mu_1 \mathcal{V} \mu_2 \doteq \neg(\neg\mu_1 \mathcal{U} \neg\mu_2)$. This latter operator allows each formula to be turned into *negation normal form*, that is, it allows the pushing of the $\neg$ operator inwards until it occurs only before propositions, without causing an exponential blow up in the size of the translated formula. From now on, each formula is considered to be in negation normal form.

A *literal* is either a proposition or its negation, an *elementary* formula is either T, or F, or a literal, or an X-formula. A set of formulas is said to be elementary if all its formulas are. A non-elementary formula μ can be decomposed, according to the tableau rules of Figure 1, so that $\mu \leftrightarrow \bigwedge_{\beta_1 \in \alpha_1(\mu)} \beta_1 \vee \bigwedge_{\beta_2 \in \alpha_2(\mu)} \beta_2$.

μ	$\alpha_1(\mu)$	$\alpha_2(\mu)$
$\mu_1 \wedge \mu_2$	$\{\mu_1, \mu_2\}$	$\{\mathrm{F}\}$
$\mu_1 \vee \mu_2$	$\{\mu_1\}$	$\{\mu_2\}$
$\mu_1 \mathcal{U} \mu_2$	$\{\mu_2\}$	$\{\mu_1, X(\mu_1 \mathcal{U} \mu_2)\}$
$\mu_1 \mathcal{V} \mu_2$	$\{\mu_2, \mu_1\}$	$\{\mu_2, X(\mu_1 \mathcal{V} \mu_2)\}$

Fig. 1. Tableau rules.

Finally, a *cover* of a set A of formulas is a, possibly empty, set of sets $C = \{C_i : i \in I\}$ such that $\bigwedge_{\mu \in A} \mu \leftrightarrow \bigvee_{i \in I} \bigwedge_{\eta_i \in C_i} \eta_i$.

We represent formulas via *labeled generalized Büchi automata*. A generalized Büchi automaton is a quadruple $\mathcal{A} = \langle \mathcal{Q}, \mathcal{I}, \delta, \mathcal{F} \rangle$, where $\mathcal{Q}$ is a finite set of

states, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of *initial states*, $\delta : \mathcal{Q} \to 2^{\mathcal{Q}}$ is the *transition function*, and $\mathcal{F} \subseteq 2^{2^{\mathcal{Q}}}$ is a, possibly empty, set of sets of accepting states $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$. An *execution* of $\mathcal{A}$ is an infinite sequence $\rho = q_0 q_1 q_2 \ldots$ such that $q_0 \in \mathcal{I}$ and, for each $i \geq 0$, $q_{i+1} \in \delta(q_i)$. ρ is *accepting execution* if, for each $F_i \in \mathcal{F}$, there exists $q_i \in F_i$ that appears infinitely often in ρ. A labeled generalized Büchi automaton 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 $\mathcal{L} : \mathcal{Q} \to 2^{\mathcal{D}}$ is the *labeling function*. A labeled generalized Büchi automaton *accepts* a word $\xi = x_0 x_1 x_2 \ldots$ from $\mathcal{D}^\omega$ iff there exists an accepting execution $\rho = q_0 q_1 q_2 \ldots$ of $\mathcal{A}$ such that $x_i \in \mathcal{L}(q_i)$, for each $i \geq 0$.

3 The Core

LTL2AUT, GPVW+, and GPVW can be obtained by suitably instantiating the core we are about to present. The instantiation affects some functions that, in what follows, are highlighted through the SMALL CAPITAL font. The central part of the core is the computation of a cover of a set of formulas, which is used for generating states. The propositional information will be used for defining the labeling, while the X information will be used to define the transition function.

3.1 Cover Computation

The algorithm for computing covers is defined by extending the propositional tableau in order to allow it to deal with temporal operators. The fundamental rules used for decomposing temporal operators are the identity $\mu \mathcal{U} \eta \equiv \eta \vee (\mu \wedge X(\mu \mathcal{U} \eta))$ and its dual $\mu \mathcal{V} \eta \equiv \eta \wedge (\mu \vee X(\mu \mathcal{V} \eta))$. The line numbers in the following description refer to the algorithm appearing in Figure 2. The algorithm handles the following data structures:

ToCover The set of formulas to be covered but still not processed.
Current The element of the cover currently being computed.
Covered The formulas already processed and covered by *Current*.
Cover The cover so far computed.

When computing the current element of the cover, the algorithm first checks whether all the formulas have been covered (line 4). If so, *Current* is ready to be added to *Cover* (line 5). If a formula μ has still to be covered (line 6), the algorithm checks whether μ has to be stored in the current element of the cover (line 8) and, if so, adds it to *Current* (line 9). Processing μ can be avoided in two cases: If there is a contradiction involving it (line 10) or it is redundant (line 12). In the former case, *Current* is discarded (line 11), while in the latter one μ is discarded (line 13). Finally, if μ does need to be covered, it is covered according to its syntactic structure. If μ is elementary, it is covered simply by itself (line 15). Otherwise, μ is covered by covering, according to the tableau rules appearing in Figure 1, either $\alpha_1(\mu)$ (line 16) or $\alpha_2(\mu)$ (line 18). This is justified by recalling that $\mu \leftrightarrow \bigwedge_{\beta_1 \in \alpha_1(\mu)} \beta_1 \vee \bigwedge_{\beta_2 \in \alpha_2(\mu)} \beta_2$.

```
1  function Cover(A)
2   return cover(A, ∅, ∅, ∅)

3  function cover(ToCover, Current, Covered, Cover)
4   if ToCover = ∅
5     then return Cover ∪ {Current}
6     else  select μ from ToCover
7           remove μ from ToCover and add it to Covered
8           if HAS_TO_BE_STORED(μ)
9             then Current = Current ∪ {μ}
10          if CONTRADICTION(μ, ToCover, Current, Covered)
11            then return Cover
12            else  if REDUNDANT(μ, ToCover, Current, Covered)
13                    then return cover(ToCover, Current, Covered, Cover)
14                    else  if μ is elementary
15                            then return cover(ToCover, Current ∪ {μ}, Covered, Cover)
16                            else  return cover(ToCover ∪ (α1(μ) \ Current),
17                                               Current, Covered,
18                                        cover(ToCover ∪ (α2(μ) \ Current),
19                                               Current, Covered, Cover))
```

Fig. 2. Cover computation.

3.2 The Automaton Construction

Our goal is to build a labeled generalized Büchi automaton recognizing exactly all the models of a linear time temporal logic formula ψ. The algorithm is presented in two phases. First, we introduce the automaton structure, i.e., its states, which are obtained as covers, initial states, and transition function. The line numbers in the following description refer to this part of the algorithm, which appears in Figure 3. Then, we complete such structure by defining labeling and acceptance conditions.

The algorithm starts by computing the initial states as cover of $\{\psi\}$ (line 2). A set U of states whose transition function has still to be defined is kept. All the initial states are clearly added to U (line 2). When defining the transition function for the state s (line 4), we first compute its successors as cover of $\{\mu : X\mu \in s\}$ (line 5). For each computed successor r, the algorithm checks whether r has been previously generated as a state r' (line 6). If so, it suffices to add r' to $\delta(s)$ (line 7). Otherwise, r is added to Q and $\delta(s)$ (lines 8 and 9). Moreover, r is also added to U (line 10), for $\delta(r)$ to be eventually computed.

The domain $\mathcal{D}$ is $2^{\mathcal{P}}$ and the label of a state s consists of all subsets of $2^{\mathcal{P}}$ that are compatible with the propositional information contained in s. More in detail, let $Pos(s)$ be $s \cap \mathcal{P}$ and $Neg(s)$ be $\{p \in \mathcal{P} : \neg p \in s\}$. Then, $\mathcal{L}(s) = \{X : X \subseteq \mathcal{P} \wedge Pos(s) \subseteq X \wedge X \cap Neg(s) = \emptyset\}$. Finally, we have to impose acceptance conditions. Indeed, our construction allows some executions inducing interpretations that are not models of ψ. This happens because it is possible to procrastinate forever the

```
1  procedure create_automaton_structure(ψ)
2    U = Q = I = Cover({ψ}), δ = ∅
3    while U ≠ ∅
4          remove s from U
5          for r ∈ Cover({μ : Xμ ∈ s})
6              if ∃r' ∈ Q such that r = r'
7              then δ(s) = δ(s) ∪ {r'}
8              else  Q = Q ∪ {r}
9                    δ(s) = δ(s) ∪ {r}
10                   U = U ∪ {r}
```

Fig. 3. The algorithm.

fulfilling of U-formulas, and arises because the formula $\mu\mathcal{U}\eta$ can be covered by covering μ and by promising to fulfill it later by covering $X(\mu\mathcal{U}\eta)$. The problem is solved by imposing generalized Büchi acceptance conditions. Informally, for each subformula $\mu\mathcal{U}\eta$ of ψ, we define a set $F_{\mu\mathcal{U}\eta} \in \mathcal{F}$ containing states s that either do not promise it or immediately fulfill it. In this way, postponing forever fulfilling a promised $\mathcal{U}$-formula gives not rise to accepting executions anymore. Formally, we set $F_{\mu\mathcal{U}\eta} \doteq \{s \in \mathit{Cover} : \textsc{satisfy}(s, \mu\mathcal{U}\eta) \rightarrow \textsc{satisfy}(s, \eta)\}$ where, again, SATISFY is a function that will be subject to instantiation.

4 GPVW, GPVW+, and LTL2AUT

GPVW is obtained by instantiating the Boolean functions parameterizing the previously described core in the following way. HAS_TO_BE_STORED(μ) returns T. CONTRADICTION(μ, *ToCover*, *Current*, *Covered*) returns T iff μ is F or μ is a literal such that $\neg\mu \in$ *Current*. REDUNDANT(μ, *ToCover*, *Current*, *Covered*) returns F. SATISFY(s, μ) returns T iff $\mu \in s$.

For GPVW+ we have the following instantiations. HAS_TO_BE_STORED(μ) returns T iff μ is a $\mathcal{U}$-formula or μ is the righthand argument of a $\mathcal{U}$-formula. CONTRADICTION(μ, *ToCover*, *Current*, *Covered*) returns T iff μ is F or the negation normal form of $\neg\mu$ is in *Covered*. REDUNDANT(μ, *ToCover*, *Current*, *Covered*) returns T iff μ is $\eta\mathcal{U}\nu$ and $\nu \in \mathit{ToCover} \cup \mathit{Current}$, or μ is $\eta\mathcal{V}\nu$ and η, $\nu \in \mathit{ToCover} \cup \mathit{Current}$. SATISFY($s, \mu$) returns T iff $\mu \in s$.

GPVW+ attempts to generate less states than GPVW by reducing the formulas to store in *Current* and by detecting redundancies and contradictions as soon as possible. Indeed, by reducing the formulas to store in *Current*, GPVW+ increases the possibility of finding matching states, while early detection of contradictions and redundancies avoids producing the part of the automaton for dealing with them. However, GPVW+ still does not solve some basic problems. First, states obtained by dealing with a $\mathcal{U}$-formula contain either the $\mathcal{U}$-formula or its righthand argument. So, for example, states generated for the righthand argument of $\mu\mathcal{U}\eta$ are equivalent to, but do not match, prior existing states generated for η. Second, redundancy and contradiction checks are performed by

explicitly looking for the source of redundancy or contradiction. So, for example, a $\mathcal{U}$-formula whose righthand argument is a conjunction is considered redundant if such conjunction appears among the covered formulas, but it is not if, instead of the conjunction, its conjuncts are present.

LTL2AUT overcomes the above problems in a very simple way: Only the elementary formulas are stored in *Current*, while information about the non-elementary ones is derived from the elementary ones and the ones stored in *ToCover* using quick syntactic techniques. More in detail, we inductively define the set $\mathcal{SI}(A)$ of the formulas *syntactically implied* by the set of formulas A as follows

- $\mathrm{T} \in \mathcal{SI}(A)$,
- $\mu \in \mathcal{SI}(A)$, if $\mu \in A$,
- $\mu \in \mathcal{SI}(A)$, if μ is non-elementary and either $\alpha_1(\mu) \subseteq \mathcal{SI}(A)$ or $\alpha_2(\mu) \subseteq \mathcal{SI}(A)$.

LTL2AUT requires then the following settings. HAS_TO_BE_STORED(μ) returns F. CONTRADICTION(μ, *ToCover*, *Current*, *Covered*) returns T iff the negation normal form of $\neg\mu$ belongs to $\mathcal{SI}(\mathit{ToCover} \cup \mathit{Current})$. REDUNDANT($\mu$, *ToCover*, *Current*, *Covered*) returns T iff $\mu \in \mathcal{SI}(\mathit{ToCover} \cup \mathit{Current})$ and, if μ is $\eta\mathcal{U}\nu$, $\nu \in \mathcal{SI}(\mathit{ToCover} \cup \mathit{Current})$. SATISFY($s, \mu$) returns T iff $\mu \in \mathcal{SI}(s)$. The special attention to the righthand arguments of $\mathcal{U}$-formulas in the redundancy check is for avoiding discarding information required to define the acceptance conditions. The proof of correctness of LTL2AUT is described in [DGV99].

5 The Test Method

The existent bibliography on problem sets and testing-generating methods for LTL and model checking is very poor. Indeed, papers usually come along with testing their results over, in the best cases, few instances. The method we have adopted is based on two analyses:

Average-behavior analysis: For a fixed number N of propositional variables and for increasing values L of the length of the formulas, a problem set $\mathcal{PS}_{\langle F,N,L\rangle}$ of F random formulas is generated and given in input to the procedures to test. After the computation, a statistical analysis is performed and the results are plotted against L. The process can be repeated for different values of N.

Temporal-behavior analysis: For a fixed number N of propositional variables, a fixed length L of the formulas, and for increasing values P of the probability of generating the temporal operators $\mathcal{U}$ and $\mathcal{V}$, a problem set $\mathcal{PS}_{\langle F,N,L,P\rangle}$ of F random formulas is generated and given in input to the procedures to test. After the computation, a statistical analysis is performed and the results are plotted against P. The process can be repeated for different values of N and L.

When generating random formulas from a formula space, for example defined by the parameters N, L, and P, our target is to cover such space as uniformly as possible. This requires that, when generating formulas of length L, we produce formulas of length exactly L, and not up to L. Indeed, in the latter way, varying L, we give preference to short formulas. Random formulas parameterized by N, L, and P, are then generated as follows. A unit-length random formula is generated by randomly choosing, according to uniform distribution, one variable. From now on, unless otherwise specified, randomly chosen stands for randomly chosen with uniform distribution. A random formula of length 2 is generated by generating $op(p)$, where op is randomly chosen in $\{\neg, X\}$ and p is a randomly chosen variable. Otherwise, with probability $\frac{P}{2}$ of choosing either $\mathcal{U}$ or $\mathcal{V}$ and probability $\frac{1-P}{4}$ of choosing $\neg$, X, $\wedge$, or $\vee$, the operator op is randomly chosen. If op is unary, the random formula of length L is generated as $op(\mu)$, for some random formula μ of length $L-1$. Otherwise, if op is binary, for some randomly chosen $1 \leq S \leq L-2$, two random formulas μ_1 and μ_2 of length S and $L-S-1$ are produced, and the random formula $op(\mu_1, \mu_2)$ of length L is generated. Since the set of operators we use is $\{\neg, X, \wedge, \vee, \mathcal{U}, \mathcal{V}\}$, random formulas for the average-behavior analysis are generated by setting $P = \frac{1}{3}$. Note that parentheses are not considered. Indeed, our definition generates a syntax tree that makes the priority between the operators clear.

In both the above analyses, the parameters we are interested in are the size of the automata, namely states and transitions, and the time required for their generation. When comparing two procedures Π_1 and Π_2 with respect to some problem set $\mathcal{PS}_{\langle F,N,L,P\rangle}$ and parameter θ, we perform the following statistical analysis. First, we compute the mean value of the outputs of Π_1 and Π_2 separately, and then consider their ratio that, hereafter, is denoted by $\frac{E(\Pi_1,\theta,\mathcal{PS}_{\langle F,N,L,P\rangle})}{E(\Pi_2,\theta,\mathcal{PS}_{\langle F,N,L,P\rangle})}$. A different statistical analysis of the data is described in [DGV99].

6 Results

LTL2AUT, GPVW, and GPVW+ have been implemented on the top of the same kernel, and are accessible through command line options. The code consists of 1400 lines of `C` plus 110 lines for a `lex/yacc` parser. The code has been compiled through `gcc version 2.7.2.3` and executed under the `SunOS 5.5.1` operating system on a `SUNW UltraSPARC-II/296 1G.`

LTL2AUT and GPVW+ have been compared, according to the test method discussed in Section 5, on 5700 randomly generated formulas. The results are shown in Figure 4. For the average behavior analysis, LTL2AUT and GPVW+ have been compared on 3300 random formulas generated, according to our test method, for $F = 100$, $N = 1, 2, 3$, and $L = 5, 10, \ldots, 55$. Formulas have been collected in 3 groups, for $N = 1, 2, 3$, and inside each group partitioned into 11 problem sets of 100 formulas each, for $L = 5, 10, \ldots, 55$. For each group, $\frac{E(\text{LTL2AUT, states },\mathcal{PS}_{\langle 100,N,L\rangle})}{E(\text{GPVW+, states },\mathcal{PS}_{\langle 100,N,L\rangle})}$, $\frac{E(\text{LTL2AUT, transitions },\mathcal{PS}_{\langle 100,N,L\rangle})}{E(\text{GPVW+, transitions },\mathcal{PS}_{\langle 100,N,L\rangle})}$, and $\frac{E(\text{LTL2AUT, time },\mathcal{PS}_{\langle 100,N,L\rangle})}{E(\text{GPVW+, time },\mathcal{PS}_{\langle 100,N,L\rangle})}$ have been plotted against L. The results show

that LTL2AUT clearly outperforms GPVW+, with respect to both the size of automata and computation time. Indeed, just considering formulas of length 30, LTL2AUT produces on the average less than 60% of the states of GPVW+

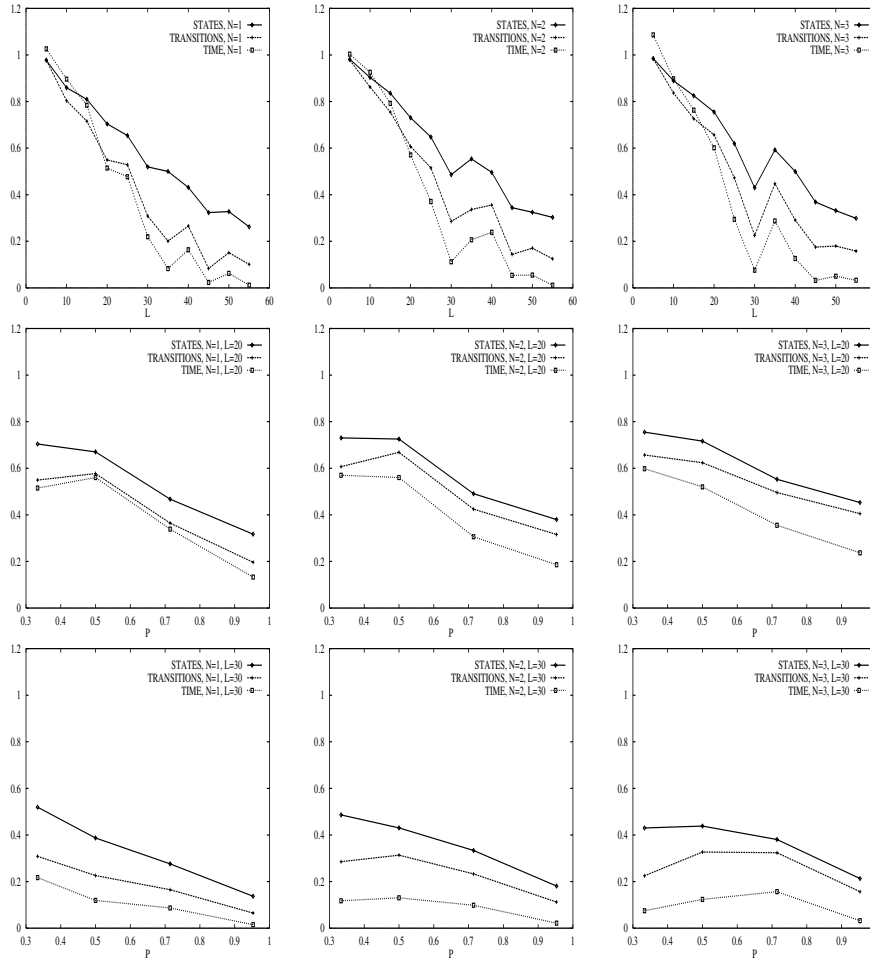

Fig. 4. LTL2AUT vs. GPVW+. Upper row: Average-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 5, 10, \ldots, 55$. Middle and lower rows: Temporal-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 20, 30$, $P = 0.\overline{3}, 0.5, 0.7, 0.95$.

(for transitions situation is even better) spending on the average less than 30% of the time of GPVW+. Moreover, the initial phase, in which LTL2AUT does have a time overhead with respect to GPVW+, affects formulas, for $L = 5$ and $N = 3$, which are solved by LTL2AUT in at most 0.000555 CPU seconds, as opposed to the most demanding sample for $L = 55$ and $N = 3$, which is solved by

LTL2AUT in 6659 CPU seconds. For the temporal-behavior analysis, LTL2AUT and GPVW+ have been compared over 2400 random formulas generated for $F = 100$, $N = 1, 2, 3$, $L = 20, 30$, and $P = 0.\overline{3}, 0.5, 0.7, 0.95$. Note that $P = 0.\overline{3}$

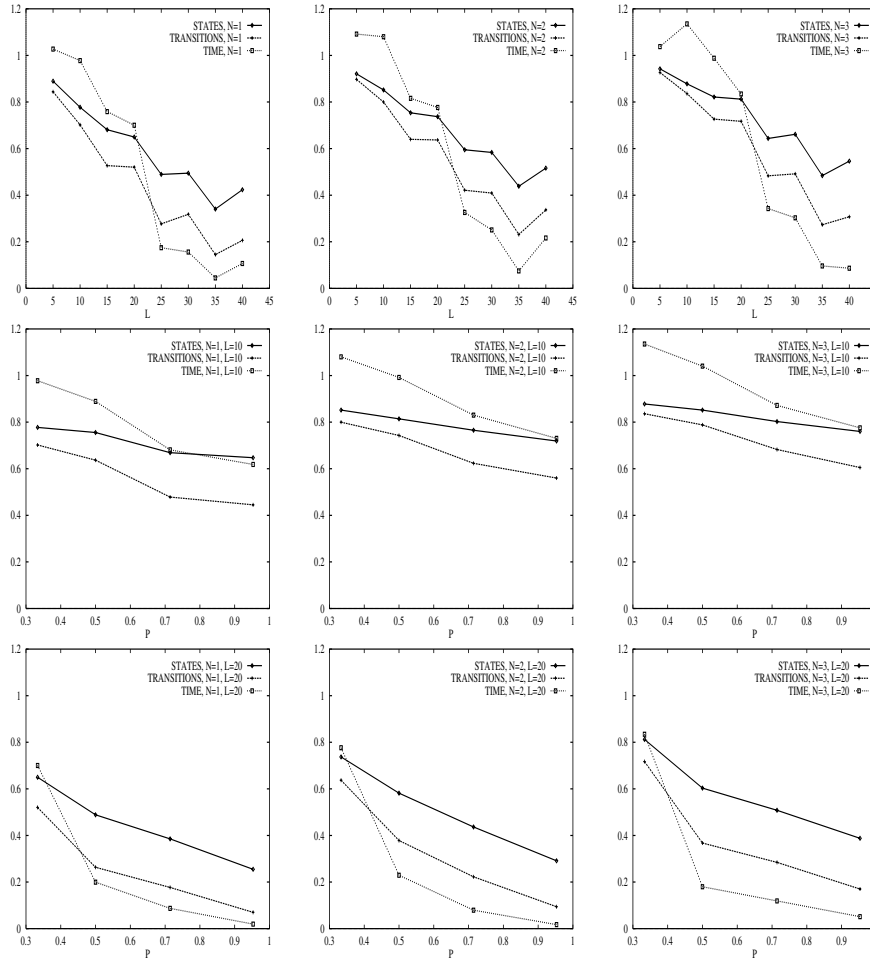

Fig. 5. GPVW+ vs. GPVW. Upper row: Average-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 5, 10, \ldots, 40$. Middle and lower rows: Temporal-behavior analysis, $F = 100$, $N = 1, 2, 3$, $L = 10, 20$, $P = 0.\overline{3}, 0.5, 0.7, 0.95$.

is the probability we have assumed for the average-behavior analysis. Formulas have been collected in 3 groups, for $N = 1, 2, 3$, and inside each group partitioned into 2 sub-groups, for $L = 20, 30$. Each sub-group has still been partitioned into 4 problem sets, for $P = 0.\overline{3}, 0.5, 0.7, 0.95$. For each sub-group, we have plotted $\frac{E(\text{LTL2AUT, states }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}{E(\text{GPVW+, states }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}$, $\frac{E(\text{LTL2AUT, transitions }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}{E(\text{GPVW+, transitions }, \mathcal{PS}_{\langle 100,N,L,P\rangle})}$, and

$\frac{E(\text{LTL2AUT, time },\mathcal{PS}_{\langle 100,N,L,P\rangle})}{E(\text{GPVW+, time },\mathcal{PS}_{\langle 100,N,L,P\rangle})}$ against P. Again, the results demonstrate that LTL2AUT clearly outperforms GPVW+.

The comparison between GPVW+ and GPVW, whose results are shown in Figure 5, follows the lines of the previous one, by only changing some parameters for allowing GPVW to compute in reasonable time. The average-behavior analysis has been carried out over 2400 random formulas generated for $F = 100$, $N = 1, 2, 3$, and $L = 5, 10, \ldots, 40$. The temporal-behavior analysis has been performed over 2400 random formulas generated for $F = 100$, $N = 1, 2, 3$, $L = 10, 20$, and $P = 0.\overline{3}, 0.5, 0.7, 0.95$. The results show that GPVW+ clearly outperforms GPVW both in the size of automata and, after an expected initial phase, also in time. The initial phase interests formulas, for $L = 10$ and $N = 3$, which are solved by GPVW+ in at most 0.004226 CPU seconds, as opposed to the hardest sample for $L = 40$ and $N = 3$, which is solved by GPVW+ in 178 CPU seconds.

Finally, a direct comparison between LTL2AUT and GPVW can be found in [DGV99].

7 Conclusions

We have demonstrated that the algorithm for building an automaton from a linear temporal logic formula can be significantly improved. Moreover, we have proposed a test methodology that can be also used for evaluating other LTL deciders, and whose underlying concept, namely targeting a uniform coverage of the formula space, can be exported to other logics. Of course, the notion of uniform coverage can be further refined, and this is part of our future work. In particular, we plan to adapt to LTL the probability distributions proposed in [MSL92] for propositional logic and adapted in [GS96] to the modal logic K. These distributions assigns equal probabilities to formulas of the same structure (e.g., 3-CNF in the propositional case). We are also planning to extend the concept of syntactic implication to a semantic one and, finally, to explore automata generation in the symbolic framework.

References

[CVWY91] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In E. M. Clarke and R. P. Kurshan, editors, *Proceedings of Computer-Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 233–242, Berlin, Germany, June 1991. Springer.

[DGV99] M. Daniele, F. Giunchiglia, and M. Y. Vardi. Improved automata generation for linear time temporal logic. Technical Report 9903-10, ITC-IRST, March 1999.

[GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.

[GS96] F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures: the case study of modal K. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE-96)*, volume 1104 of *LNAI*, pages 583–597, Berlin, July30 August–3 1996. Springer.

[Hol97] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[KMMP93] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In C. Courcoubertis, editor, *Proceedings of Computer-Aided Verification (CAV'93)*, volume 697 of *LNCS*, pages 97–109, Elounda, Greece, June 1993. Springer.

[MSL92] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In W. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 459–465, San Jose, CA, July 1992. MIT Press.

[Sch98] S. Schwendimann. A new one-pass tableau calculus for PLTL. *Lecture Notes in Computer Science*, 1397:277–291, 1998.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 4, pages 133–191. Elsevier Science Publishers B. V., 1990.

[VW86] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *lics86*, pages 332–344, 1986.

[VW94] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 15 November 1994.