"More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking[*]

Roberto Sebastiani and Stefano Tonetta

DIT, Università di Trento, via Sommarive 14, 38050 Povo, Trento, Italy
{rseba,stonetta}@dit.unitn.it

Abstract. The standard technique for LTL model checking ($M \models \neg\varphi$) consists on translating the negation of the LTL specification, φ, into a Büchi automaton A_φ, and then on checking if the product $M \times A_\varphi$ has an empty language. The efforts to maximize the efficiency of this process have so far concentrated on developing translation algorithms producing Büchi automata which are "*as small as possible*", under the implicit conjecture that this fact should make the final product smaller. In this paper we build on a different conjecture and present an alternative approach in which we generate instead Büchi automata which are "*as deterministic as possible*", in the sense that we try to reduce as much as we are able to the presence of non-deterministic decision states in A_φ. We motivate our choice and present some empirical tests to support this approach.

1 Introduction

Model checking is a formal verification technique which allows for checking if the model of a system verifies some desired property. In LTL model checking, the system is modeled as a Kripke structure M, and (the negation of) the property is encoded as an LTL formula φ. The standard technique for LTL model checking consists on translating φ into a Büchi automaton A_φ, and then on checking if the product $M \times A_\varphi$ has an empty language. To this extent, the quality of the translation technique plays a key role in the efficiency of the overall process.

Since the seminal work in [6], the efforts to maximize the efficiency of this process have so far concentrated on developing translation algorithms which produce from each LTL formula a Büchi automaton (BA henceforth) which is "*as small as possible*" (see, e.g., [1,12,3,5,4,9,7]). This is motivated by the implicit heuristic conjecture that, as the size of the product $M \times A_\varphi$ of the Kripke structure M and the BA A_φ is in worst-case the product of the sizes of M and A_φ, reducing the size of A_φ is likely to reduce the size of the final product also in the average case. This conjecture is implicitly assumed in most of papers (e.g., [1,12,5,7]), which use the size of the BA's as the only measurement of efficiency in empirical tests.

Remarkably, Etessami and Holtzmann [3] tested their translation procedures by measuring both the size of resulting BA's and the actual efficiency of the LTL model checking

[*] This work has been sponsored by the CALCULEMUS! IHP-RTN EC project, contract code HPRN-CT-2000-00102, and has thus benefited of the financial contribution of the Commission through the IHP programme. The authors are also sponsored by a MIUR COFIN02 project, code 2002097822_003.

process, and noticed that "... a smaller number of states in the automaton does not necessarily improve the running time and can actually hurt it in ways that are difficult to predict" [3].

In this paper we propose and explore a new research direction. Instead of wondering what makes the BA A_φ smaller, we wonder directly what may make the *product automaton* $M \times A_\varphi$ smaller, independently on the size of the BA A_φ. We start from noticing the following fact: if a state s in $M \times A_\varphi$ is given by the combination of the states s' in M and s'' in A_φ, and if s'' is a *deterministic* decision state —that is, each label may match with at most only one successor of s''— then s has at most the same amount of successor states as s', no matter the number of successors of s''. From this fact, we conjecture that reducing the presence of non-deterministic decision states in the BA is likely to reduce the size of the final product in the average case, no matter if this produces bigger BA's. (Notice that it is not always possible to reduce completely the presence of non-deterministic decision states, as not every LTL formula φ can be translated into a deterministic BA, and even deciding whether the translation is possible belongs to EXPSPACE and is PSPACE-Hard [11].)

In order to explore the effectiveness of the above conjecture, we thus present a new approach in which we generate from each LTL formula a BA which is "*as deterministic as possible*", in the sense that we try to reduce as much as we are able to the presence of non-deterministic decision states in the generated automaton. This is done by exploiting the idea of *semantic branching*, which has proved very effective in the domain of modal theorem proving [8].

The rest of the paper is structured as follows. In Section 2 we present some preliminary notions. In Section 3 we describe the main ideas of our approach. In Section 4 we describe the LTL to BA algorithm we have implemented. In Section 5 we present the results of an extensive empirical test. In Section 6 we conclude, describing also some future work. For lack of space, the correctness and completeness of the algorithm is proved in an extended technical report, which is available at `http://www.science.unitn.it/~stonetta/modella.html`

2 Preliminaries

We use Linear Temporal Logic (LTL) with its standard syntax and semantics [2] to specify properties. Let Σ be a set of elementary propositions. A propositional literal (i.e., a proposition p in Σ or its negation $\neg p$) is a LTL formula; if φ_1 and φ_2 are LTL formulae, then $\neg\varphi_1, \varphi_1 \wedge \varphi_2, \varphi_1 \vee \varphi_2, \mathbf{X}\varphi_1, \varphi_1 \mathbf{U} \varphi_2, \varphi_1 \mathbf{R} \varphi_2$ are LTL formulae, where $\mathbf{X}$, $\mathbf{U}$ and $\mathbf{R}$ are the standard "next", "until" and "releases" temporal operators respectively. We see the familiar $\top$ (true), $\bot$ (false), $\mathbf{F}\varphi_1$ (eventually φ_1) and $\mathbf{G}\varphi_1$ (globally φ_1) as standard abbreviations of $p \vee \neg p$, $p \wedge \neg p$, $\top \mathbf{U} \varphi_1$ and $\bot \mathbf{R} \varphi_1$ respectively.

For every operator *op* in $\{\wedge, \vee, \mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{R}\}$, we say that φ is an *op*-formula if *op* is the root operator of φ (e.g., $\mathbf{X}(p\mathbf{U}q)$ is an $\mathbf{X}$-formula). We say that the occurrence of a subformula φ_1 in an LTL formula φ is a *top level occurrence* if it occurs in the scope of only boolean operators $\neg, \wedge, \vee$ (e.g., $\mathbf{F}p$ occurs at top level in $\mathbf{F}p \vee \mathbf{XF}q$, while $\mathbf{F}q$ does not).

A Kripke Structure M is a tuple $\langle S, S_0, T, \mathcal{L}\rangle$ with a finite set of states S, a set of initial states $S_0 \subseteq S$, a transition relation $T \subseteq S \times S$ and a labeling function $\mathcal{L} : S \to 2^{\Sigma}$, where Σ is the set of atomic propositions.

A labeled generalized BA (LGBA) [6] is a tuple $A := \langle Q, Q_0, T, \mathcal{L}, D, \mathcal{F}\rangle$, where Q is a *finite set of states*, $Q_0 \subseteq Q$ is the set of *initial states*, $T \subseteq Q \times Q$ is the *transition relation*, $D := 2^{\Sigma}$ is the finite *domain* (*alphabet*), $\mathcal{L} : Q \to 2^D$ is the *labeling function*, and $\mathcal{F} \subseteq 2^Q$ is the set of *accepting conditions* (fair sets). A *run* of A is an infinite sequence $\sigma := \sigma(0), \sigma(1), ...$ of states in Q, such that $\sigma(0) \in Q_0$ and $T(\sigma(i), \sigma(i+1))$ holds for every $i \geq 0$. A run σ is an *accepting run* if, for every $F_i \in \mathcal{F}$, there exists $\sigma(j) \in F_i$ that appears infinitely often in σ. An LGBA A *accepts* an infinite word $\xi := \xi(0), \xi(1), ... \in D^{\omega}$ if there exists an accepting run $\sigma := \sigma(0), \sigma(1), ...$ so that $\xi(i) \in \mathcal{L}(\sigma(i))$, for every $i \geq 0$. Henceforth, if not otherwise specified, we will refer to an LGBA simply as a Büchi automaton (BA).

Notice that each state in a Kripke structure is labeled by one total truth assignment to the propositions in Σ, whilst the label of a state in a BA represents a *set* of such assignments. A partial assignment represents the set of all total assignments/labels which entail it. We represent truth assignments indifferently as sets of literals $\{l_i\}_i$ or as conjunctions of literals $\bigwedge_i l_i$, with the intended meaning that a literal p (resp. $\neg p$) in the set/conjunction assigns p to true (resp. false).

Notationally, we use ξ for representing an infinite word over 2^{Σ}; $\xi(i)$ is the i-th element and ξ_i is the suffix starting from $\xi(i)$. We use σ for an infinite sequence of states (runs); $\sigma(i)$ is the i-th element and σ_i is the suffix starting from $\sigma(i)$. We use μ for truth assignments. We use φ, ψ, ϑ for general formulae. We denote by $succ(s, A_{\varphi})$ $[succ(s, M)]$ the set of successor states of the state s in a BA A_{φ} [Kripke structure M].

If μ is a truth assignment and φ is an LTL formula, we denote by $\varphi[\mu]$ the formula obtained by substituting every top level literal $l \in \mu$ in φ with $\top$ (resp. $\neg l$ with $\bot$) and by propagating the $\top$ and $\bot$ values in the obvious ways. (E.g., $(p \vee \mathbf{X}\varphi_1) \wedge (q \vee \mathbf{X}\varphi_2)[\{p, \neg q\}] = \mathbf{X}\varphi_2$.)

An *elementary formula* is an LTL formula which is either a constant in $\{\top, \bot\}$, a propositional literal or a $\mathbf{X}$-formula. A *cover* for a set of LTL formulae $\{\varphi_k\}_k$ is a set of sets of elementary formulae $\{\{\vartheta_{ij}\}_j\}_i$ s.t. $\bigwedge_k \varphi_k \leftrightarrow \bigvee_i \bigwedge_j \vartheta_{ij}$. (Henceforth, we indifferently represent covers either as sets of sets or as disjunctions of conjunctions of elementary formulae.) A cover for $\{\varphi_k\}_k$ is typically obtained by computing the *disjunctive normal form* (DNF) of $\bigwedge_k \varphi_k$, considering $\mathbf{X}$-subformulae as boolean propositions.

The general translation schema of an LTL formula φ into a BA A_{φ} works as follows [6]. First, φ is written in negative normal form (NNF), that is, all negations are pushed down to literal level. Second, φ is expanded by applying the *tableau rewriting rules*:

$$\varphi_1 \mathbf{U} \varphi_2 \Longrightarrow \varphi_2 \vee (\varphi_1 \wedge \mathbf{X}(\varphi_1 \mathbf{U} \varphi_2)), \quad \varphi_1 \mathbf{R} \varphi_2 \Longrightarrow \varphi_2 \wedge (\varphi_1 \vee \mathbf{X}(\varphi_1 \mathbf{R} \varphi_2)) \quad (1)$$

until no $\mathbf{U}$-formula or $\mathbf{R}$-formula occurs at top level. Then the resulting formula is rewritten into a cover by computing its DNF. Each disjunct of the cover represents a state of the automaton: all propositional literals represent the label of the state —that is, the condition the input word must satisfy in that state— and the remaining $\mathbf{X}$-formulae

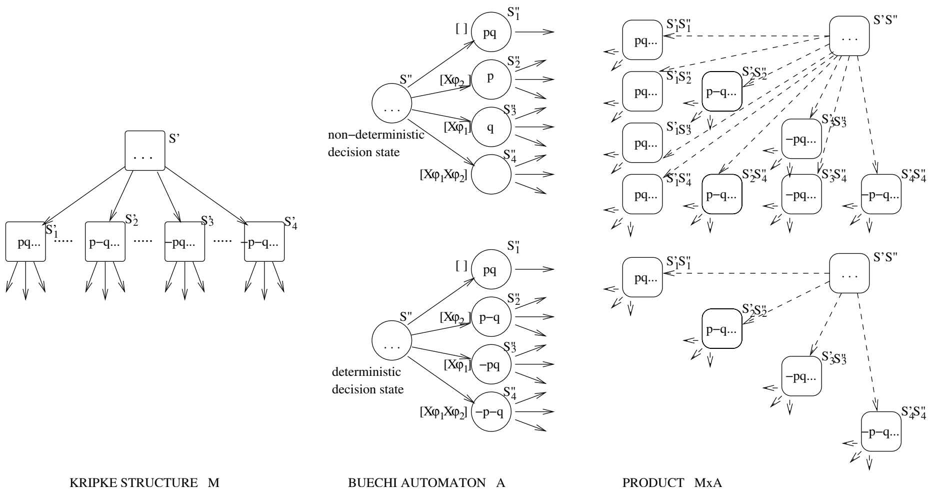

Fig. 1. Product of a generic Kripke structure with a non-deterministic (up) and a deterministic (down) cover expansion of $\varphi := (p \vee \mathbf{X}\varphi_1) \wedge (q \vee \mathbf{X}\varphi_2)$.

represent the *next part* of the state —that is, the obligations that must be fulfilled to get an accepting run— and determine the transitions outcoming from the state.

The process above is applied recursively to the next part of each state, until no new obligation is produced. This results into a closed set of covers, so that, for each cover $\mathcal{C}$ in the set, the next part of each disjunct in $\mathcal{C}$ has a cover in the set. Then $A_\varphi = \langle Q, Q_0, T, \mathcal{L}, D, \mathcal{F} \rangle$ is built as follows. The initial states are given by the cover of φ. The transition relation is given by connecting each state to those in the cover of its next part. An acceptance condition F_i is added for every elementary subformula in the form $\psi \mathbf{U} \vartheta$, so that F_i contains every state $s \in Q$ such that $s \not\models (\psi \mathbf{U} \vartheta)$ or $s \models \vartheta$.

3 A New Approach

3.1 Deterministic and Non-deterministic Decision States

We say that two states are *mutually consistent* if their respective labels are mutually consistent, *mutually inconsistent* otherwise. We say that a state s in a BA is a *deterministic decision state* if the labels of all successor states of s are pairwise mutually inconsistent, a *non-deterministic decision state* otherwise. Intuitively, if s is a deterministic decision state, then every label in the alphabet is consistent with (the label of) at most one successor of s. A BA is deterministic if its states are all deterministic decision states and if its initial states are pairwise mutually inconsistent.

We consider an LTL model checking problem $M \models \neg\varphi$, where M is a Kripke structure and φ is an LTL formula. A_φ is the BA into which φ is converted, and $M \times A_\varphi$ is the product of M and A_φ. Each state s in $M \times A_\varphi$ is given by the (consistent) pairwise combination $s's''$ of some states s' in M and s'' in A_φ, and the successor states of s are given by all the consistent combinations of one successor of s' and one of s'':

$$succ(s, M \times A_\varphi) = \{s'_i s''_j | s'_i \in succ(s', M), s''_j \in succ(s'', A_\varphi), s'_i s''_j \not\models \bot\}, \quad (2)$$
$$|succ(s, M \times A_\varphi)| \leq |succ(s', M)| \cdot |succ(s'', A_\varphi)|, \quad (3)$$

where $s's''$ denotes the combination of the states s' and s'' and "$s'_i s''_j \not\models \bot$" denotes the fact that the combination of s' and s'' is consistent.

We make the following key observation: if s'' is a deterministic decision state, then each successor state of s' can combine consistently with at most one successor of s'', so that s has at most as many successor states as s'. Thus (3) reduces to

$$|succ(s, M \times A_\varphi)| \leq |succ(s', M)|. \quad (4)$$

The above observation suggests to us the following heuristic consideration: in order to minimize the size of the product $M \times A_\varphi$, we should try to make A_φ "*as deterministic as we can*" —that is, to reduce as much as we can the presence of non-deterministic decision states in A_φ— no matter if the resulting BA is greater than other equivalent but "less deterministic" BA's.

Example 1. Consider the state s' of a Kripke structure M in Figure 1 (left) and its successor states s'_1, s'_2, s'_3 and s'_4 with labels $\{p, q, ...\}$, $\{p, \neg q, ...\}$, $\{\neg p, q, ...\}$ and $\{\neg p, \neg q, ...\}$ respectively. Consider the LTL formula $\varphi := (p \vee \mathbf{X}\varphi_1) \wedge (q \vee \mathbf{X}\varphi_2)$ for some LTL subformulae φ_1 and φ_2. Consider the two covers of φ:

$$\mathcal{C}_1 := \{\{p, q\}, \{p, \mathbf{X}\varphi_2\}, \{q, \mathbf{X}\varphi_1\}, \{\mathbf{X}\varphi_1, \mathbf{X}\varphi_2\}\}, \quad (5)$$
$$\mathcal{C}_2 := \{\{p, q\}, \{p, \neg q, \mathbf{X}\varphi_2\}, \{\neg p, q, \mathbf{X}\varphi_1\}, \{\neg p, \neg q, \mathbf{X}\varphi_1, \mathbf{X}\varphi_2\}\}, \quad (6)$$

which generate the two BA's A in Figure 1 (center) respectively. In the first BA the state s'' is a non-deterministic decision state. Thus the successors of $s's''$ in $M \times A$ are the consistent states belonging to the cartesian product of the successor sets of s' and s''. In particular, s'_1 matches with all successor states of s'', s'_2 matches with s''_2 and s''_4, s'_3 matches with s''_3 and s''_4, and s'_4 matches with s''_4. In the second BA s'' is a deterministic decision state. Thus, each successor of s' matches with only one successor of s''. $\diamond$

Remark 1. It is well-known (see, e.g., [11]) that converting a non-deterministic BA A into a deterministic one A' (when possible) may make the size of the latter blow up exponentially wrt. the size of the former in the worst case. This is due to the fact that each state s' of A' represents a subset of states $\{s_i\}_i$ of A, so that $|A'| \leq 2^{|A|}$, and hence $|M \times A'| \leq |M| \cdot 2^{|A|}$, whilst $|M \times A| \leq |M| \cdot |A|$. Thus, despite the local effect described above (4), one may suppose that globally our approach worsens the global performance.

We notice instead that $\mathcal{L}(s') \models \bigwedge_i \mathcal{L}(s_i)$, so that the set of states in M matching with s' is a *subset of the intersection* of the set of states in M matching with each s_i:

$$\{s^* \in M \mid s^* s' \not\models \bot\} \subseteq \bigcap_i \{s^* \in M \mid s^* s_i \not\models \bot\}.^1 \quad (7)$$

Thus, the process of determinization may increase the number of states in the BA, but reduces as well the number of states in M with which each state in the BA matches. $\square$

Example 2. Consider the LTL formula and the covers of Example 1. (Notationally, we denote by $\mathcal{C}_{ij}$ the jth element of $\mathcal{C}_i$.) Then $\mathcal{C}_{21}$, $\mathcal{C}_{22}$, $\mathcal{C}_{23}$ and $\mathcal{C}_{24}$ match with $1/4$ of the possible labels, whilst $\mathcal{C}_{11}$, $\mathcal{C}_{12}$, $\mathcal{C}_{13}$ and $\mathcal{C}_{14}$ match with $1/4$, $1/2$, $1/2$ and $1/1$ of the possible labels respectively.

3.2 Deterministic and Non-deterministic Covers

Let $\{\varphi_k\}_k$ be a set of LTL formulae in NNF, let φ denote $\bigwedge_k \varphi_k$, and let $\mathcal{C} := \{\{\vartheta_{ij}\}_j\}_i$ be a cover for φ. $\mathcal{C}$ can be written as $\{\mu_i \cup \chi_i\}_i$, where $\mu_i := \{\vartheta_{ij} \in \{\vartheta_{ij}\}_j | \vartheta_{ij}\ prop.\ literal\}$ and $\chi_i := \{\vartheta_{ij} \in \{\vartheta_{ij}\}_j | \vartheta_{ij}$ **X**-formula$\}$ are the set of propositional literals and **X**-formulae in $\{\vartheta_{ij}\}_j$ respectively. Thus

$$\varphi \leftrightarrow \bigvee_i (\mu_i \wedge \chi_i). \tag{8}$$

We say that a cover $\mathcal{C} = \{\mu_i \cup \chi_i\}_i$ as in (8) is a *deterministic cover* if and only if all μ_i's are pairwise mutually inconsistent, *non-deterministic* otherwise.

Example 3. Consider the LTL formula and the covers of Example 1. $\mathcal{C}_1$ is non-deterministic because, e.g., $\{p, q\}$ and $\{p\}$ are mutually consistent. $\mathcal{C}_2$ is deterministic because $\{p, q\}$, $\{p, \neg q\}$, $\{\neg p, q\}$ and $\{\neg p, \neg q\}$ are pairwise mutually inconsistent. $\diamond$

In the construction of a BA, each element $\mu_i \wedge \chi_i$ in a cover $\mathcal{C}$ represents a state s_i, where μ_i is the label of the state and χ_i is its next part (by abuse of notation, we henceforth call such a formula "state"). Thus, a deterministic cover $\mathcal{C}$ represents a set of states whose labels are pairwise mutually inconsistent. Consequently, deterministic covers (when admissible) give rise to deterministic decision states.

3.3 Computing Deterministic Covers

As said in the previous sections, the standard approach for computing covers is based on the recursive application of the tableau rules (1) and on the subsequent computation of the DNF of the resulting formula. The latter step is achieved by applying recursively to the top level formulae the rewriting rule

$$\varphi' \wedge (\varphi_1 \vee \varphi_2) \Longrightarrow (\varphi' \wedge \varphi_1) \vee (\varphi' \wedge \varphi_2) \tag{9}$$

and then by removing every disjunct which propositionally implies another one. As in [8], we call step (9) *syntactic branching* because it splits "syntactically" on the disjuncts of the top level $\vee$-subformulae. As noticed in [8], a major weakness of syntactic branching is that it generates subbranches which are not mutually inconsistent, so that, even after the removal of implicant disjuncts, the distinct disjuncts of the final DNF may share models. As a consequence, if the boolean parts of two disjuncts in a cover are mutually consistent, non-deterministic decision states are generated.

To avoid this fact we compute a cover in a new way. After applying the tableau rules, we apply recursively to the top level boolean propositions the Shannon expansion

$$\varphi \Longrightarrow (p \wedge (\varphi[\{p\}])) \vee (\neg p \wedge (\varphi[\{\neg p\}])). \tag{10}$$

As in [8], we call step (10) *semantic branching* because it splits "semantically" on the truth values of top level propositions. The key issue of semantic branching is that it generates subbranches which are all mutually inconsistent [8].Thus, after applying (10) to all top level literals in φ, we obtain an expression in the form

$$\bigvee_i (\mu_i \wedge \varphi[\mu_i]), \tag{11}$$

such that all μ_i's are all pairwise mutually inconsistent and $\varphi[\mu_i]$ is a boolean combination of $\mathbf{X}$-formulae. If all $\varphi[\mu_i]$'s are conjunctions of $\mathbf{X}$-formulae, then (11) is in the form (8), so that we have obtained a deterministic cover. If not, every disjunct $(\mu_i \wedge \varphi[\mu_i])$ in (11) represents a set of states S_i such that all states belonging to the same set S_i have the same label μ_i but different next-part, whilst any two states belonging to different sets S_i's are mutually inconsistent.

As a consequence, the presence of non-unary sets S_i is a potential source of non-determinism. Thus, if this does not affect the correctness of the encoding (see below), we rewrite each formula $\varphi[\mu_i]$ into a single $\mathbf{X}$-formula by applying the rewriting rules:

$$\mathbf{X}\varphi_1 \wedge \mathbf{X}\varphi_2 \Longrightarrow \mathbf{X}(\varphi_1 \wedge \varphi_2), \tag{12}$$
$$\mathbf{X}\varphi_1 \vee \mathbf{X}\varphi_2 \Longrightarrow \mathbf{X}(\varphi_1 \vee \varphi_2). \tag{13}$$

The result is clearly a deterministic cover. We call this step *branching postponement* because (13) allows for postponing the or-branching to the expansion of the next part.

Example 4. Consider the LTL formula and the covers of Example 1. The cover $\mathcal{C}_1$ is obtained by applying syntactic branching to φ from left to right, whilst $\mathcal{C}_2$ is obtained by applying semantic branching to φ, splitting on p and q. (As all $\varphi[\mu_i]$'s are conjunctions of $\mathbf{X}$-formulae, no further step is necessary.) $\diamond$

Unfortunately, branching postponement is not always safely applicable. In fact, while rule (12) can always be applied without affecting the correctness of the encoding, this is not the case of rule (13). For example, it may be the case that $\mathbf{X}\varphi_1$ and $\mathbf{X}\varphi_2$ in (13) represent two states s_1 and s_2 respectively so that s_1 is in a fair set F_1 and s_2 is not, and that the state corresponding to $\mathbf{X}(\varphi_1 \vee \varphi_2)$ is not in F_1; if so, we may loose the fairness condition F_1 if we apply (13). This fact should not be a surprise: if branching postponement were always applicable, then we could always generate a deterministic BA from an LTL formula, which is not the case [11]. Our idea is thus to apply branching postponement only to those formulae $\varphi[\mu_i]$ for which we are guaranteed it does not cause incorrectness, and to apply standard DNF otherwise. This will be described in detail in the next section.

To sum up, semantic branching allows for partitioning the next states into mutually inconsistent sets of states S_i, whilst branching postponement, when applied, collapses each S_i into only one state. Notice that

– unlike syntactic branching, semantic branching guarantees that the only possible sources of non-determinism (if any) are due to the next-part components $\varphi[\mu_i]$'s. No source of non-determinism is introduced by the boolean components μ_i's;

```
cover compute_cover($\varphi$) {
  $apply\_tableau\_rules(\varphi)$;
  for each $p$ occurring at top level in $\varphi$ {
    $\varphi := (p \wedge \varphi[\{p\}]) \vee (\neg p \wedge \varphi[\{\neg p\}])$;   // semantic branching on labels
    $simplify(\varphi)$;                                                               // boolean simplification
  }                                                                                    // now $\varphi = \bigvee_{i \in I}(\mu_i \wedge \varphi[\mu_i])$
  $\varphi := \bigvee_{i \in I}(\mu_i \wedge DNF(\varphi[\mu_i]))$;                    // now $\varphi = \bigvee_{i \in I}(\mu_i \wedge \bigvee_{j \in J_i} \bigwedge_{k \in K_{ij}} \mathbf{X}\psi_{ijk})$
  $\varphi := \bigvee_{i \in I}(\mu_i \wedge \bigvee_{j \in J_i} \mathbf{X} \bigwedge_{k \in K_{ij}} \psi_{ijk})$;   // factoring out the $\mathbf{X}$ operators
  $\mathcal{C}^*(\varphi) := \bigvee_{i \in I, j \in J_i}(\mu_i \wedge \mathbf{X}\psi_{ij})$;          // $\psi_{ij}$ being $\bigwedge_{k \in K_{ij}} \psi_{ijk}$
  $\mathcal{C}(\varphi) := \bot$;                                                      // initialization of $\mathcal{C}(\varphi)$
  for each $i \in I$ {
    $s_i := (\mu_i \wedge \mathbf{X} \bigvee_{j \in J_i} \psi_{ij})$;
    $Subs(s_i) := \bigvee_{j \in J_i}(\mu_i \wedge \mathbf{X}\psi_{ij})$;
    if $(Postponement\_is\_Safe(s_i))$
    then $\mathcal{C}(\varphi) := \mathcal{C}(\varphi) \vee s_i$;                       // postponement applied
    else $\mathcal{C}(\varphi) := \mathcal{C}(\varphi) \vee Subs(s_i)$;                 // postponement not applied
  }
  return $\mathcal{C}(\varphi)$;}
```

Fig. 2. The schema of the cover computation algorithm

– branching postponement reduces the number of states sharing the same labels even if it is applied only to a strict subset of the subformulae $\varphi[\mu_i]$ in (11). Thus, also *partial* applications of branching postponement make the BA "more deterministic".

4 The MoDeLLA Algorithm

In the current state-of-the-art algorithms the translation from an LTL formula φ into a BA A_φ can be divided into three main phases:

1. *Formula rewriting*: apply a finite set of rewriting rules to φ in order to remove redundancies and make it more suitable for an efficient translation.
2. *BA construction from* φ: build a BA with the same language of the input formula φ.
3. *BA reduction*: reduce redundancies in the BA (e.g., by exploiting simulations).

In our work, we focus on phase 2. According to the new approach proposed in the previous section, we have conceived and implemented a new translation algorithm, called MoDeLLA (**Mo**re **De**terministic **LTL** to **A**utomata) which builds a BA from an LTL formula trying to apply branching postponement as often as it is able to.

4.1 The Basic Algorithm

The general schema of the BA construction in MoDeLLA, in its basic form, is the standard one proposed in [6] and briefly recalled in Section 2. MoDeLLA differs from previous conversion algorithms in two steps: the computation of the covers and the computation of the fair sets.

Computation of the cover. The function which computes the cover of the formula φ is described in Figure 2. First, we apply, as usual, the tableau rewriting rules (1) (line 1). The formula obtained is a boolean combination of literals and $\mathbf{X}$-formulae. After applying the semantic branching rules on labels (10), we get a disjunction of formulae in the form (11) (lines 2-5).

If now we applied branching postponement (12) and (13), denoting $\bigwedge_{k \in K_{ij}} \psi_{ijk}$ by ψ_{ij}, we would obtain the deterministic cover:

$$\mathcal{C}^D(\varphi) := \{\mu_i \wedge \mathbf{X} \bigvee_{j \in J_i} \psi_{ij}\}_{i \in I}. \tag{14}$$

Unfortunately, as pointed out in section 3.3, branching postponement may affect the correctness of the BA. Thus, we apply it only in "safe" cases. First, for every disjunct $\mu_i \wedge \varphi[\mu_i]$ we temporarily compute $DNF(\varphi[\mu_i])$ and then we factor $\mathbf{X}$ out of every conjunction in $DNF(\varphi[\mu_i])$ (lines 6-7). We obtain a temporary non-deterministic cover

$$\mathcal{C}^*(\varphi) := \{\mu_i \wedge \mathbf{X}\psi_{ij}\}_{i \in I, j \in J_i}. \tag{15}$$

Notice that every state s_i in $\mathcal{C}^D(\varphi)$ is equivalent to the disjunction of $|J_i|$ states in $\mathcal{C}^*(\varphi)$:

$$s_i = \mu_i \wedge \mathbf{X} \bigvee_{j \in J_i} \psi_{ij} = \bigvee_{j \in J_i} (\mu_i \wedge \mathbf{X}\psi_{ij}). \tag{16}$$

For every $i \in I$, we define the set of **substates** of s_i as:

$$Subs(s_i) := \{\mu_i \wedge \mathbf{X}\psi_{ij}\}_{j \in J_i} \tag{17}$$

($Subs(s_i)$ is the set S_i in Section 3.3.) We extend the definition to every state s^* of $\mathcal{C}^*(\varphi)$ by saying that $Subs(s^*) := \{s^*\}$.

Then, the cover $\mathcal{C}(\varphi)$ is built in the following way (lines 10-16): for every $i \in I$, we add to $\mathcal{C}(\varphi)$ s_i if postponement is safe for s_i, $Subs(s_i)$ otherwise. *Postponement_is_Safe(s)* decides if branching postponement is **safe** for a state s according to a sufficient condition described in the following paragraphs.

Computation of fair sets. If $\mathcal{U}_\varphi$ is the set of $\mathbf{U}$-formulae which are subformulae of φ, the usual set of accepting conditions is:

$$\mathcal{F}^* := \{F^*_{\psi\mathbf{U}\vartheta} | \psi\mathbf{U}\vartheta \in \mathcal{U}_\varphi\}, \tag{18}$$

$$F^*_{\psi\mathbf{U}\vartheta} := \{s \in Q | s \not\models \psi\mathbf{U}\vartheta \text{ or } s \models \vartheta\}. \tag{19}$$

We extend these definitions as follows

$$\mathcal{F} := \{F_{\mathcal{H}} | \mathcal{H} \in 2^{\mathcal{U}_\varphi}\}, \tag{20}$$

$$\begin{aligned} F_{\mathcal{H}} := \{s \in Q | & \text{ there exists } \psi\mathbf{U}\vartheta \in \mathcal{H} \text{ s.t.} \\ & \text{for each } s^* \in Subs(s), s^* \not\models \psi\mathbf{U}\vartheta \text{ or for each } s^* \in Subs(s), s^* \models \vartheta_h\}. \end{aligned} \tag{21}$$

Notice that, if $|\mathcal{H}| = 1$ and, for every $s \in Q$, $|Subs(s)| = 1$ (i.e. we have never applied branching postponement), this is the usual notion (i.e. $F_{\{\psi\mathbf{U}\vartheta\}} = F^*_{\psi\mathbf{U}\vartheta}$).

We say that the branching postponement is not *safe* for a state s if there exists $F_{\mathcal{H}} \in \mathcal{F}$ such that $s \notin F_{\mathcal{H}}$ and there exist $\psi\mathbf{U}\vartheta \in \mathcal{H}, s^* \in Subs(s)$ such that $s^* \in F_{\psi\mathbf{U}\vartheta}$.

With this condition we are guaranteed that if the BA A^*_φ built without branching postponement has an accepting run σ^* over a word ξ, then the correspondent run σ of the BA A_φ built with safe branching postponement is also accepting.

Example 5. Consider the LTL formula $\varphi := \mathbf{FG}p$. After having applied the tableau rules and semantic branching on labels, we obtain $\varphi = (p \wedge (\mathbf{XFG}p \vee \mathbf{XG}p)) \vee (\neg p \wedge \mathbf{XFG}p)$. If $s = (p \wedge \mathbf{X}(\mathbf{FG}p \vee \mathbf{G}p))$, the branching postponement is not safe for s. Indeed, $Subs = \{(p \wedge \mathbf{XFG}p), (p \wedge \mathbf{XG}p)\}$ and $(p \wedge \mathbf{XG}p) \in F_{\{\mathbf{FG}p\}}$ but $s \notin F_{\{\mathbf{FG}p\}}$. Thus, *compute_cover* produces the cover:

$$\{(p \wedge \mathbf{XFG}p), (p \wedge \mathbf{XG}p), (\neg p \wedge \mathbf{XFG}p)\}. \quad \diamond \tag{22}$$

4.2 Improvements

We describe some improvements to the basic schema of MoDeLLA described in the previous section. Most of them are adapted from known optimizations.

Pruning the fair sets. In the previous section, we have noticed that the basic version of MoDeLLA computes $2^{|\mathcal{U}|}$ fair sets. Thus, in order to reduce this number, in the final computation of the fair conditions, $\mathcal{F}$, we apply the following simplification rules, which are a simple version of an optimization introduced in [12]:

- for all $F \in \mathcal{F}$, if $F = Q$ then $\mathcal{F} := \mathcal{F}\backslash\{F\}$,
- for all $F, F' \in \mathcal{F}$, if $F \subseteq F'$ then $\mathcal{F} := \mathcal{F}\backslash\{F'\}$.

Remark 2. Due to the existential quantifier in the definition (18) of $F_{\mathcal{H}}$, for every formula $\psi\mathbf{U}\vartheta \in \mathcal{H}$, we have that $F_{\{\psi\mathbf{U}\vartheta\}} \subseteq F_{\mathcal{H}}$. For this reason, after the above fair sets pruning, MoDeLLA will keep only those accepting condition $F_{\mathcal{H}}$ for which $\mathcal{H}$ is a singleton. Thus, we obtain that $|\mathcal{F}| \leq |\mathcal{U}_\varphi|$, as in the usual construction.

Merging states. After computing a cover, if two states $s_1 = (\mu_1, \chi), s_2 = (\mu_2, \chi)$ have the same next part χ and satisfy the following property:

for all $\psi\mathbf{U}\vartheta \in \mathcal{U}_\varphi$,
(for all $s_1^* \in Subs(s_1), s_1^* \models \psi\mathbf{U}\vartheta) \Leftrightarrow$ (for all $s_2^* \in Subs(s_2), s_2^* \models \psi\mathbf{U}\vartheta$) and
(for all $s_1^* \in Subs(s_1), s_1^* \models \vartheta) \Leftrightarrow$ (for all $s_2^* \in Subs(s_2), s_2^* \models \vartheta$),

then we substitute them with $s = (\mu_1 \vee \mu_2, \chi)$ where $Subs(s) := Subs(s_1) \cup Subs(s_2)$. Notice that for every $F \in \mathcal{F}$, we have $s_1 \in F \Leftrightarrow s_2 \in F \Leftrightarrow s \in F$. This technique is a simpler version of the one introduced in [7], which however applies the merging only after moving labels from the states to the transitions.

Example 6. Consider the formula of Example 5 and the cover produced by the basic version of MoDeLLA. After merging the states with the above technique, the cover (22) becomes $\{(\top \wedge \mathbf{XFG}p), (p \wedge \mathbf{XG}p)\}$. Notice that the labels $\top$ and p of the two states are mutually consistent so that the BA is still non-deterministic. However, we have reduced the number of states without increasing the non-determinism. $\diamond$

5 Empirical Results

MoDeLLA is an implementation in `C` of the algorithm described in Section 4. It implements only phase 2, so that it can be used as kernel of optimized algorithms including also formula rewriting (phase 1) and BA reduction (phase 3). (Indeed, we believe our technique is orthogonal to the rewriting rules of phase 1 and to BA reductions.)

We extensively tested MoDeLLA in comparison with the state-of-the-art algorithms. Unlike, e.g., [1,12,5,7], we did not consider as parameters for the comparison the size of the BA produced, but rather the number of states and transitions of the product $M \times A_\varphi$ between the BA and a randomly-generated Kripke structure. To accomplish this, we used LBTT 1.0.1 [13], a randomized testbench which takes as input a set of translation algorithms for testing their correctness. In particular, LBTT gives the same formula (either randomly-generated or provided by the user) to the distinct algorithms, it gets their output BA's and it builds the product of these automata with a randomly-generated Kripke structure M of given size $|M|$ and (approximated) average branching factor b. LBTT provides also a random generator producing formulae of given size $|\varphi|$ and maximum number of propositions P.

To compare MoDeLLA with state-of-the-art algorithms, we provided interfaces between LBTT and Wring 1.1.0 [12,9] and between LBTT and TMP 2.0 [3,4]. Since LBTT computes the direct product between the BA and the state space, the size of the product is not affected by the number of fair sets of the BA. Thus, to get more reliable results, we have dealt only with *degeneralized* BA, and we have applied a simple procedure described in [6] to convert a BA into a Büchi automata with a single fair set.

We have run LBTT on three PCs Dual Processor with 2GB RAM on `Linux RedHat`. All the tools and the files used in our experiments can be downloaded at `http://www.science.unitn.it/~stonetta/modella.html`.

5.1 Comparing Pure Translators

In a first session of tests, we wanted to verify the effectiveness of MoDeLLA as a pure "phase 2" translator. Thus, we compared MoDeLLA with "pure" translators (no formula rewriting, no BA reduction), i.e. with GPVW [6], LTL2AUT [1][2] and Wring [12] with rewriting rules and simulation-based reduction disabled (Wring(2) henceforth). Notice that TMP uses LTL2AUT as phase 2 algorithm [3]. For reasons which will be described in the next section, we run also a version of MoDeLLA without the merging of states (MS) optimization of Section 4.2 (which we call MoDeLLA–MS henceforth).

We fixed $|M|$ to 5000 states and we made b grow exponentially in $\{2, 4, 8, 16, 32, 64\}$. We did four series of tests: 1) tests with 200 random formulae with $|\varphi| = 15$ and $P = 4$; 2) tests with 200 random formulae with $|\varphi| = 15$ and $P = 8$; 3) tests on the 27 formulae proposed in [12]; 4) tests on the 12 formulae proposed in [3]. For every formula φ, we tested both $M \models \varphi$ and $M \models \neg\varphi$. The results are reported in Figure 3. (In the fourth series, the run of GPVW and LTL2AUT were stopped for $b \geq 16$ because they caused a memory blowup.)

[2] For GPVW and LTL2AUT, we have used the reimplementation provided by Wring.

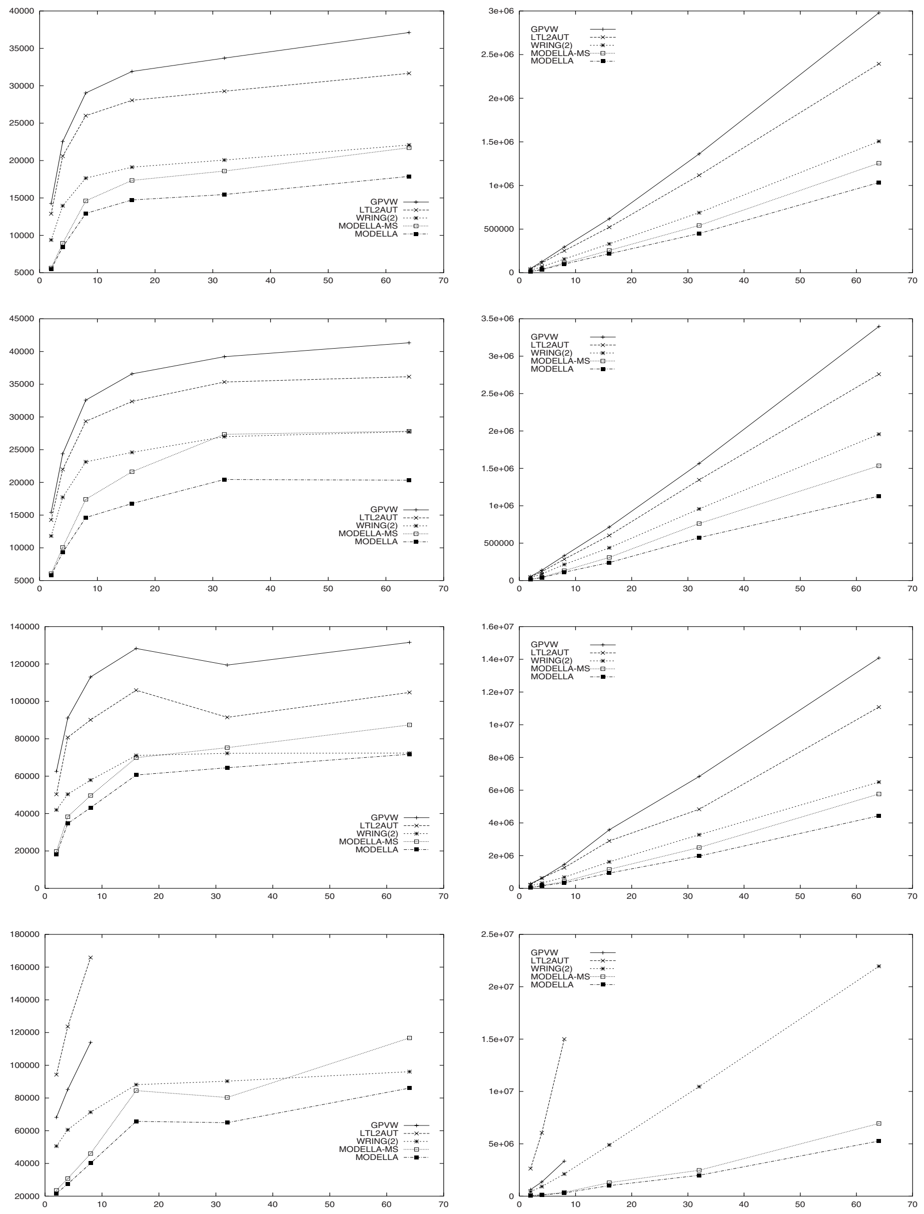

Fig. 3. Performances of the pure “phase 2” algorithms. X axis: approximate average branching factor of M. Y axis: mean number of states (left column) and of transitions (right column) of the product $M \times A_\varphi$. 1st row: 400 random formulae, 4 propositions; 2nd row: 400 random formulae, 8 propositions; 3rd row: 24 formulae from [12]; 4th row: 54 formulae from [3].

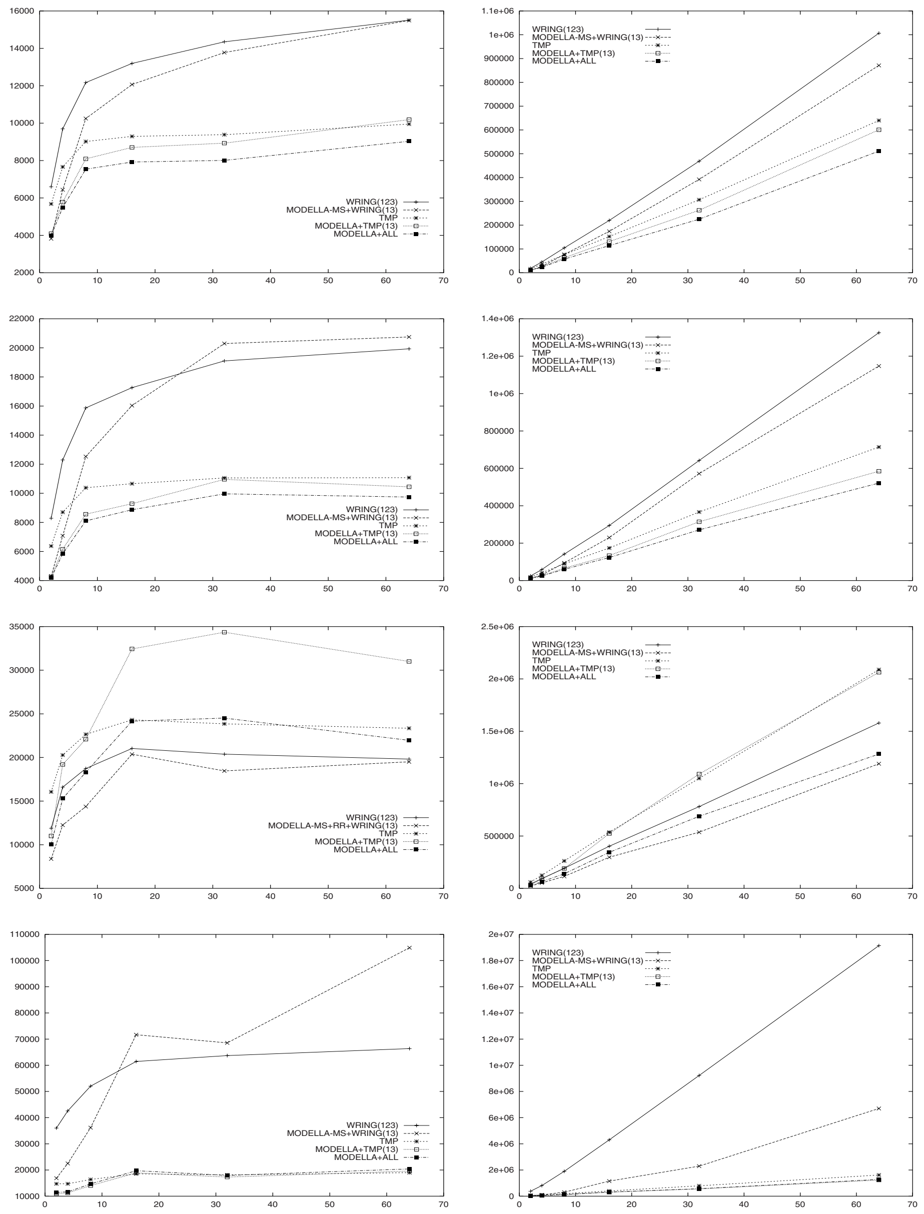

Fig. 4. Same experiments as in Figure 3, adding phases 1 and 3 to the pure “phase 2” algorithms.

Comparing the plots in the first column (number of states of $M \times A_\varphi$) we notice that (i) GPVW and LTL2AUT are significantly less performing than the other algorigthms; (ii) MoDeLLA performs better than Wring(2) in all the test series; (iii) even with MS optimization disabled, MoDeLLA performs mostly better than Wring(2).

Comparing the plots in the second column (number of transitions of $M \times A_\varphi$) we notice that Wring(2) performs much better than LTL2AUT and GPVW, and that both MoDeLLA and MoDeLLA–MS perform always better than Wring(2). In particular, the performance gaps are very relevant in the fourth test series.

5.2 Comparing Translators with Rewriting Rules and Simulation-Based Reduction

In a second section of tests, we investigated the behaviour of MoDeLLA as the kernel of a more general algorithm, embedding also the rewriting rules (phase 1) and the simulation-based reduction (phase 3) of Wring and TMP. This allows us for investigating the effective "orthogonality" of our new algorithm wrt. the introduction of rewriting rules and of simulation-based reduction.

First, we applied to our algorithm the rewriting rules described in [12] and interfaced MoDeLLA–MS with the simulation-based reduction engine of Wring. Unfortunately, since Wring accepts only states labeled with conjunctions of literals, we could interface Wring only with MoDeLLA–MS and not with the full version of MoDeLLA. (We denote the former as MoDeLLA–MS+Wring(13) henceforth.) Second, we applied to MoDeLLA the rewriting rules described in [3] and the simulation-based reduction described in [4] which are respectively the phase 1 and the phase 3 of TMP. (We call this enhanced version of our algorithm MoDeLLA+TMP(13) henceforth.) Finally, we implemented the optimization technique described in [7]. When we enable this technique, together with the rewriting rules and the TMP's automata reduction, we refer to it as MoDeLLA+ALL.

We run the tests with the same parameters of the first session of tests, obtaining the results of Figure 4. By looking at the plots, one can observe the following facts for both the columns (number of states and number of transitions of $M \times A_\varphi$): (i) if compared with the correspondent phase 2, MoDeLLA–MS+Wring(13) and MoDeLLA+TMP(13) benefit a lot respectively from Wring's and TMP's rewriting rules and simulation-based reduction, although slightly less than Wring and TMP theirselves do; (ii) MoDeLLA–MS+Wring(13) and MoDeLLA+TMP(13) perform mostly better respectively than Wring(123) and than TMP, although the gap we had with "pure" algorithms is reduced; (iii) MoDeLLA+ALL performs better than all the others, except with the third test series where MoDeLLA–MS+Wring(13) is the best performer.

6 Conclusions and Future Work

In this paper we have presented a new approach to build BA from LTL formulae, which is based on the idea of reducing as much as possible the presence of nondeterministic decision states in the automata; we have motivated this choice and presented a new conversion algorithm, MoDeLLA, which implements these ideas; we have presented an extensive empirical test, which suggests that MoDeLLA is a valuable alternative as a core engine for state-of-the-art algorithms.

We plan to extend our work on various directions. From the implementation viewpoint, we want to implement in MoDeLLA the simulation-based reduction techniques presented in [12] in order to have a tool which exploits the power of all state-of-the-art automata reductions. From an algorithmic viewpoint, we want to investigate new optimizations steps ad hoc for our approach. From a theoretical viewpoint, we want to investigate more general sufficient conditions for branching postponement.

Another interesting research direction, though much less straightforward, might be to investigate the feasibility and effectiveness of introducing semantic branching in the alternating-automata based approach of [5].

Finally, we would like to test the performance (wrt. time and memory consuming) of state-of-the-art LTL model checkers, e.g. SPIN [10], on real-world benchmarks by using the automata built by MoDeLLA.

References

1. M. Daniele, F. Giunchiglia, and M. Vardi. Improved Automata Generation for Linear Time Temporal Logic. In *Proc. CAV'99*, volume 1633 of *LNCS*. Springer, 1999.
2. E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science Publisher B.V., 1990.
3. K. Etessami and G. Holtzmann. Optimizing Büchi Automata. In *Proc. CONCUR'2000*, volume 1877 of *LNCS*, 2000. Springer.
4. K. Etessami, R. Schuller, and T. Wilke. Fair Simulation Relations, Parity Games, and State Space Reduction for Buechi Automata. In *Automata, Languages and Programming, 28th international colloquium*, volume 2076 of *LNCS*. Springer, July 2001.
5. P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *Proc. CAV'01*, volume 2102 of *LNCS*, pages 53–65. Springer, 2001.
6. R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th IFIP/WG6 .1 Symposium on Protocol Specification, Testing and Verification*, Warzaw, Poland, 1995. Chapman & Hall.
7. D. Giannakopoulou and F. Lerda. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *Proc. FORTE'02.*, volume 2529 of *LNCS*. Springer, 2002.
8. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures – the case study of modal K(m). *Information and Computation*, 162(1/2), October/November 2000.
9. S. Gurumurty, R. Bloem, and F. Somenzi. Fair Simulation Minimization. In *Proc. CAV'02*, number 2404 in LNCS. Springer, 2002.
10. G. Holtzmann. The Model Checker Spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.
11. O. Kupferman and M.Y. Vardi. Freedom, Weakness, and Determinism: From Linear-time to Branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, June 1998.
12. F. Somenzi and R. Bloem. Efficient Büchi Automata from LTL Formulae. In *Proc CAV'00*, volume 1855 of *LNCS*. Springer, 2000.
13. H. Tauriainen. A Randomized Testbench for Algorithms Translating Linear Temporal Logic Formulae into Büchi Automata. In *Proceedings of the Concurrency, Specification and Programming 1999 Workshop (CS&P'99)*, pages 251–262. Warsaw University, September 1999.