Practical Model Checking of LTL with Past*

M. Pradella[2], P. San Pietro[1], P. Spoletini[1], A. Morzenti[1]

[1]Dipartimento di Elettronica e Informazione, Politecnico di Milano,

P.za Leonardo da Vinci 32, 20133 Milano, Italia

[2]CNR IEIIT-MI, Via Ponzio 34/5, 20133 Milano, Italia

email: {pradella, sanpietr, spoleti, morzenti}@elet.polimi.it

ABSTRACT

LTL (Linear Temporal Logic) has become the standard language for linear-time model checking. LTL has only future operators, while it is widely accepted that many specifications are easier, shorter and more intuitive when also past operators are allowed. Moreover, adding past operators does not increase the complexity of LTL model checking, which is still PSPACE-complete. However, model checking past formulae is not very easy in practice, and it is not clear how to efficiently reuse existing model checkers like SPIN. In this paper, we propose a reasonably efficient approach to model (and satisfiability) checking of LTL-with-past formulae in a quasi-separate normal form, which occurs very often in applications.

1 Introduction

Model checking is the automatic verification that a model (typically a transition system) of a system possesses certain (un)desired properties. Linear Temporal Logic (*LTL*, which was proposed by [Pnu77] to study the correctness of concurrent programs), is one of the standard languages for expressing properties of transition systems, and it is supported by many model checkers such as SPIN [Hol97]. LTL, however, has only future modalities, while it is widely recognized that its extension with past operators [Kam68] allows one to write specifications that are easier, shorter and more intuitive [LPZ85]. A traditional example, taken from [Sch02], is the specification "Every alarm is due to a fault", which using the globally operator G and the "previously" operator O (Once) may be written as:

(1) $$G\ (alarm \rightarrow O\ fault)$$

From [GPSS80] we know that LTL with past does not add expressive power to future only LTL. Moreover, a well-known separation theorem by Gabbay [Gab89] allows for the elimination of past operators, obtaining an LTL formula to be evaluated in the initial instant only. The following is one of the simplest LTL versions of the same specification, using the U (Until) operator:

(2) $$\neg\ (\neg fault\ U\ (alarm \wedge \neg\ fault))$$

Clearly, the latter version is much harder to read and understand. Also, the elimination of past operators may in general introduce a blow-up that is nonelementary in the alternation depth of future and past operators of the original formula. Even though the separation theorem gives only an upper bound for elimination of past operators, an exponential lower bound in the length of the formula is shown in [LMS02], where a LTL+past formula ϕ (actually, a purely-past formula in the scope of a globally G operator) is shown to be representable in LTL only with a formula of size $\Omega(2^{|\phi|})$.

On the other hand, adding past operators does not increase the complexity of model checking, which is still PSPACE-complete (the fact was known, but rarely exploited, and noticed again in [LMS02]).

* Work partially supported by the MIUR projects: "QUACK: Piattaforma per la qualità di sistemi embedded integrati di nuova generazione" and "FIRB: Applicazioni della Teoria degli Automi all'Analisi, alla Compilazione e alla Verifica di Sistemi Critici e in Tempo Reale."

However, in practice model checking LTL+past formulae is not very easy, and it is not clear how to efficiently reuse existing model checkers like SPIN. A few older works, such as [VW94], and more recent works such as [BC03, GO03, Mar02] have dealt with model checking LTL+past, but, at the best of our knowledge, no algorithm has been completely implemented.

The version of metric temporal logic we used, called TRIO, was first introduced in [GMM90], around at the same time of MTL (Metric Temporal Logic, [Koy90]), and in the general case it is much more expressive than LTL. TRIO allows metric operators also in the past, and has the feature of being interpretable on dense or discrete, finite or infinite, domains, such as the integers, the reals, the natural numbers [MMG92], and of allowing modular, object-oriented constructs [MS94] to support specification of large systems. TRIO tools were mainly based on testing techniques [MMM95, SMM00] (and, to a lesser extent, on theorem proving [GM02]) rather than model checking. In [MPSS03] we showed how to deal with metric operators when using SPIN [Hol97] as a satisfiability or model checker for a restricted version of TRIO, more succinct than, but substantially equivalent to, LTL.

In our experience, formulae of the form *Gp*, where p is a formula without future operators, are used very often in specifications, probably even more than formulae with future. Typically, formulae which are not already in purely-past or purely-future form are in separate normal form [Gab89], i.e., in the form $G(\mathcal{B}(\varphi_1, .. \varphi_n, \pi_1, .., \pi_m))$, where G is the globally operator, each φ_i is a formula with no past operators, each π_j is a formula with no future operators, and $\mathcal{B}$ is a boolean combination of its m+n arguments. Moreover, in practice the boolean combination $\mathcal{B}$ is usually very shallow, without deep nesting of boolean operators. In rarer cases, formulae are not separate, but past and future operators have a very shallow alternation level (typically, only one when the external Globally operator is ignored). Higher level of alternation between past and future operators are unheard of, for the simple reason that they are unreadable and counter-intuitive. Also intricate theoretical examples, such as the above one of [LMS02], are in separate normal form.

The goal of this paper is to show that when formulae are separate, or quasi-separate, model checking of LTL+past formulae is practically feasible, without any particular loss in performance (or even with a gain) when considering similar LTL formulae.

Notice that by Gabbay's separation theorem, LTL+past formulae can always be transformed in separate normal form, but with the cited nonelementary blow-up in the alternation depth of future and past operators. Hence, the technique we illustrate here should not be considered as a new theoretical result on model checking a generic LTL+past formula through a preliminary application of the separation theorem, since the separation procedure may be far less efficient than directly model checking the formula with techniques such as the incremental tableau of [KMMP93]. Our goal is instead to apply the SPIN model checker to already separate (or quasi-separate) formulae, which are by far the most frequent in applications, and to show the advantages of the approach.

The main idea of the paper is to split the automaton equivalent to a given LTL+past formula in the composition of two parts: an "efficient" Büchi automaton for the (separated) past components of a formula (here called *past automaton*) and a traditional alternating automaton for the (separated) future components [KV97]. We show that the composition of the two automata can be efficiently implemented in SPIN, and that the checking of the past component of the formula by means of a separate Büchi automaton is quite efficient compared with that of the future component. This rules out an alternative approach that one may consider for checking separate LTL+past formulae, based on the translation of the past component into an equivalent strictly-future LTL formula.

Our approach can naturally be compared with recent work aiming at the translation of LTL properties into Büchi automata, such as LTL2BA [GO01] and Wring [SB00]: a comparison is provided in Section 4. We point out, however, that these tools build, as in the traditional model-checking scenario, a so-called never claim, i.e., an automaton specifying the negation of a temporal logic property, to be applied typically to an operational model of the system. The formula is usually small and the states of the corresponding

automaton are explicitly listed. In our approach, we often lack an operational model of the system and the model checker is used as a satisfiability checker to show the validity of the implication *specification* $\rightarrow$ *property*, where both *specification* and *property* are formulae. This means that the explicit construction of all the states of the automaton corresponding may be too large to be handled. Instead, in our approach we apply an "on the fly" translation procedure: the Promela program derived from the formula *specification* $\rightarrow$ *property* does not enumerate explicitly all the state but use parallel processes and variables to represent the state space. Hence, the explicit construction of all reachable states is delayed until the verification is run, allowing for very large formulae to be handled. Also, since we often use the model checker as a satisfiability checker, the automata we derive from the formulae are language acceptors: they must be coupled with some additional Promela fragments generating the values, over time, of the logical variables. This "generative" component can trivially be obtained by encoding a systematic, exhaustive enumeration of all possible variable values over time, but this can potentially lead to a combinatorial explosion of the search state space, thus making the proposed approach infeasible in practice. This issue has been dealt with in a previous paper [MPSS03] using techniques that exploit the modular structure of the specification and the logical dependencies among specification items, and it is not repeated here, since the main focus of the present work is the definition and combination of two automata for the separated pure-past and pure-future components of a specification.

In the remainder of the paper we use Emerson's convention [Eme90] of denoting with LTLB the (propositional) version of Linear Temporal Logic with Both past and future operators, reserving the name LTLP for the version with no future operators.

The paper is organized as follows: Section 2 introduces LTLB and the past automaton technique for LTLP; Section 3 merges a past automaton with a one way alternating automaton for LTL; Section 4 reports some experimental results. Section 5 draws our conclusions.

2 LTLB, LTLP and past automata

Let Ap be a finite set of atomic propositions. An LTLB formula has the following syntax:

$\xi ::= p \mid \xi \wedge \xi \mid \neg\xi \mid \xi\ U\ \xi \mid \xi\ S\ \xi \mid X\ \xi \mid Y\ \xi$

where $p \in$ Ap and X (neXt), Y (Yesterday), U (Until), S (Since), are the basic temporal operators.

Derived temporal operators may be defined as follows: $F\ \xi \leftrightarrow \text{true}\ U\ \xi$ (sometimes in the Future); $G\ \xi \leftrightarrow \neg F \neg\ \xi$ (Globally); $O\ \xi \leftrightarrow \text{true}\ S\ \xi$ (Once); $H\ \xi \leftrightarrow \neg O \neg\ \xi$ (Historically).

The semantics of LTLB has a pretty standard definition on ω-words. Given a finite alphabet Σ, Σ^* denotes the set of finite words over Σ. An ω–word over Σ is an infinite sequence $w = a_0\ a_1\ a_2 \ldots$, with $a_j \in \Sigma$. The set of all ω–words over Σ is denoted as Σ^ω. We denote an element a_j of $w = a_0\ a_1\ a_2 \ldots$ as w(j), and the finite prefix $a_0\ a_1 \ldots a_i$ of w with w_i.

Let $\mathbb{N}$ be the set of natural numbers. For all LTLB formulae ξ for all $w \in (2^{Ap})^\omega$, for all $i \in \mathbb{N}$ the satisfaction relation $\models$ is defined as follows.

if $\xi \in$ Ap, $w,i \models \xi$ iff $\xi \in w(i)$;

$w,i \models \neg\psi$ iff $w,i \models \psi$ does not hold;

$w,i \models \psi \wedge \eta$ iff $w,i \models \psi$ and $w,i \models \eta$;

$w,i \models \psi\ S\ \eta$ iff $\exists 0 \le j \le i$ s.t. $w,j \models \eta$ and $\forall j < k \le i$, $w,k \models \psi$;

$w,i \models Y\psi$ iff $i>0$ and $w,i-1 \models \psi$;

$w,i \models \psi\ U\ \eta$ iff $\exists\ j \ge i$ $w,j \models \eta$ and $\forall i \le k < j$, $w,k \models \psi$;

$w,i \models X\psi$ iff $w,i+1 \models \psi$.

A formula ξ is *initially true* on a ω-word w if $w,0 \models \xi$.

The language of infinite words defined by a LTLB formula ξ is $L^{\omega}(\xi) = \{w \in (2^{Ap})^{\omega} | w,0 \models \xi\}$, which describes *the set of infinite (initial) models* of ξ.

Traditional LTL is obtained from LTLB by forbidding past operators (i.e., with the following syntax: $\varphi ::= p \mid \varphi \wedge \varphi \mid \neg \varphi \mid \varphi \, U \, \varphi \mid X \, \varphi$, with $p \in Ap$). LTLP is the set of LTLB formula *without future* operators, i.e., defined by the following syntax: $\pi ::= p \mid \pi \wedge \pi \mid \neg \pi \mid \pi \, S \, \pi \mid Y \, \pi$, with $p \in Ap$. The semantics and the languages of LTL and of LTLP are defined as for LTLB (the clauses for, respectively, past and future operators simply do not apply).

In the above definition, the semantics of LTLP applies to ω-words as well as to finite words, since past clauses only refer to a finite prefix of a word (it would not be difficult to deal with finite words also for future clauses, but this is not necessary here). Hence, for every finite word $z \in (2^{Ap})^*$, the definition of $z, i \models \pi$ is exactly the same if z is finite or infinite. We can define the *set of finite models* for a formula π as a language $L^*(G\pi)$ on finite words: $L^*(G\pi) = \{z \in (2^{Ap})^* \mid \text{for every } i \in [0..|z|-1], z,i \models \pi\}$.

Let Pref(W) be the set of finite prefixes of words in $W \subseteq \Sigma^*$: $\{x \in \Sigma^* \mid \exists\, y \in \Sigma^*: xy \in W\}$.

Lemma 1: $\mathrm{Pref}(L^*(G\pi)) = L^*(G\pi)$

Proof: Clearly, $L^*(G\pi) \subseteq \mathrm{Pref}(L^*(G\pi))$. To show the converse, let $x \in \mathrm{Pref}(L^*(G\pi))$. The word x is a prefix z_j of a word $z \in L^*(G\pi)$ for some $j \geq 0$. But if $z \in L^*(G\pi)$ then for every $i \in [0..|z|-1]$, $z,i \models \pi$, which obviously entails that for every $i \in [0..|z|-1]$, $z_i,i \models \pi$. Hence, each $z_i \in L^*(G\pi)$, and thus also z_j. □

The rest of the section is devoted to the definition of a Büchi automaton accepting $L^{\omega}(G\pi)$, where π is a LTLP formula. A well-known concept of the theory of ω-automata is the *limit* of a language $W \subseteq \Sigma^*$: $\vec{W} = \{w \in \Sigma^{\omega} \mid w \text{ has an infinite number of prefixes in } W\}$. A fundamental property of every *deterministic* ω-automaton A is that $L^{\omega}(A) = \overrightarrow{L^*(A)}$ (see e.g. [PP03]).

Lemma 2: For every LTLP formula π, $L^{\omega}(G\pi) = \overrightarrow{L^*(G\pi)}$

Proof: We first notice that (1) $w \in L^{\omega}(G\pi)$ if, and only if, for every $i \in \mathbb{N}$, $w_i \in L^*(G\pi)$.

In fact, $w \in L^{\omega}(G\pi) \Leftrightarrow$ for every $i \in \mathbb{N}$, $w,i \models \pi \Leftrightarrow$ for every $i \in \mathbb{N}$, $w_i,i \models \pi \Leftrightarrow$ for every $i \in \mathbb{N}$, $w_i \in L^*(G\pi)$.

Let $w \in L^{\omega}(G\pi)$. By (1), for every $i \in \mathbb{N}$, $w_i \in L^*(G\pi)$, entailing, by definition of limit, $w \in \overrightarrow{L^*(G\pi)}$.

Let $w \in \overrightarrow{L^*(G\pi)}$. By definition, there are infinitely many prefixes w_i in $L^*(G\pi)$. But if a $w_i \in L^*(G\pi)$, then, by Lemma 1, also $w_j \in L^*(G\pi)$ for every $j<i$. Therefore, for every $i \in \mathbb{N}$, $w_i \in L^*(G\pi)$, and, by (1), $w \in L^{\omega}(G\pi)$. □

Given a finite automaton A, let $L^*(A)$ be the language on finite words recognized by A and let $L^{\omega}(A)$ be the language recognized by A when A is interpreted as a Büchi automaton over ω-words.

Statement 3: Given a LTLP formula π, there exist two deterministic finite automata $A_{G\pi}$ and A_{π}, called the *past automaton* of $G\pi$ and π respectively, such that:

1. For every $i \in \mathbb{N}$, the automaton A_{π}, after reading the first i symbols of an ω-word w, is in a final state iff $w,i \models \pi$.
2. $L^*(A_{G\pi}) = L^*(G\pi)$.
3. $L^{\omega}(A_{G\pi}) = L^{\omega}(G\pi)$.
4. The number of states of each automaton is $2^{O(|\pi|)}$.

Sketch of the proof:

We first prove (1). In [LPZ85], it has been shown that for every LTLP formula π on the alphabet Ap of propositional letters, it is possible to define a *deterministic* finite-state automaton $A_\pi = (I, Q, \tau, q^0, F)$ with the input alphabet $I = 2^{Ap}$, a set of states Q, a transition function $\tau: Q \times I \rightarrow Q$, an initial state q^0, and a set $F \subseteq Q$ of final states such that for all $w \in (2^{Ap})^\omega$, for all $j \geq 0$, $\tau(q^0, w_j) \in F$ iff $w,j \models \pi$. Part (2) follows since it is immediate to modify A_π into $A_{G\pi}$, by restricting Q to F and thus making it accept $L^*(G\pi)$.

Part (3) also follows, since $L^\omega(A_{G\pi})$ = (by determinism of $A_{G\pi}$) $\overrightarrow{L^*(A_{G\pi})} = \overrightarrow{L^*(G\pi)}$ = (by Lemma 2) $L^\omega(G\pi)$.

Part (4) can be proved by actually defining the automaton (which stores in its state the truth value of all $O(|\pi|)$ subformulae of π). □

Statement 3 allows for very simple treatment of LTLP formulae, since $A_{G\pi}$ (and A_π) can be easily obtained in SPIN by creating one process with local variables of total length $O(|\pi|)$ bits, storing up to $2^{O(|\pi|)}$ states. The automaton implemented in SPIN, as in [LPZ85], is actually taking care at once of all LTLP formulae $\pi_1, .., \pi_k$ of a specification in separate normal form: the state set Q can be extended to $2^{subf(\pi_1)} \cup ... \cup 2^{subf(\pi_k)}$, by introducing k sets of final states $F_1, ..., F_k$, each F_i identifying the truth of formula π_i.

By contrast, it is well known that there are LTL formulae whose satisfiability is different when interpreted on finite or infinite words (e.g., GX true is satisfiable on infinite words only). Hence, Lemma 2 does not hold for LTL: if φ is a LTL formula, $L^\omega(G\varphi) \neq \overrightarrow{L^*(G\varphi)}$, since a Büchi automaton accepting $L^\omega(G\varphi)$, as defined with the traditional tableau construction of [Wol87] or with alternating automata [Var94, KPV01, GO01], may in general be nondeterministic. Hence, LTLP is actually easier to model check than LTL, even though the computational complexity is the same.

3 Alternating Automata for LTL and their composition with Past Automata

We briefly and intuitively introduce here Büchi Alternating Automata (or *BAA* for short) [CKS81]. In a *deterministic* automaton, the transition function maps a *⟨state, input symbol⟩* pair to a *single* state, called the next state. The automaton accepts its input if either *state* is final and the input is finished, or the remaining suffix of the input word is accepted from the next state. On the other hand, in a nondeterministic automaton a ⟨state, input symbol⟩ pair is mapped to a *set* of states. Here we have two possible different interpretations of the transition function: either as an existential branching mode, or as a universal branching mode. In the existential mode, which is the standard interpretation of nondeterminism, the automaton accepts if at least *one* of the states of the set accepts the remaining input suffix; in the universal mode, it accepts if *all* the states of the set accept the remaining input suffix. An alternating automaton provides both existential and universal branching modes. Its transition function maps a ⟨state, input symbol⟩ pair into a (positive) boolean combination of states. Quite naturally, $\wedge$ is used to denote universality, while $\vee$ denotes existentiality. Alternating automata are very convenient since they may be exponentially more succinct than nondeterministic automata and are very well suited for dealing with logic formulae.

Büchi Alternating Automata (BAA)

Here are some preliminary definitions, following standard terminology (e.g., [Tho97]). Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{N}^*$ the set of finite word on $\mathbb{N}$, and let $x \in \mathbb{N}^*$ and $c \in \mathbb{N}$. A *tree* is a set $T \subseteq \mathbb{N}^*$ such that $x.c \in T \Rightarrow x \in T$ (c is called a *child* of x). The empty word ε is called the *root* of T. Elements of T are called *nodes*. A node is a *leaf* if it has no children. A *path* P of a tree T is a set $P \subseteq T$ which contains ε and such that for every $x \in P$, either P is a leaf or there exists a unique c such that $x.c \in P$.

A *Büchi Alternating Automaton* (BAA) is a quintuple $A = (\sum, Q, q^0, \tau, F)$, where $\sum$ is the (finite) *alphabet*, Q is the set of *states*, $q^0 \in Q$ is the *initial state*, τ is the *transition function*, $F \subseteq Q$ is the set of *final states*. The transition function is τ: $Q \times \sum \rightarrow \mathcal{B}^+(Q)$, where, for every M, $\mathcal{B}^+(M)$ indicates a positive boolean combination of elements in M, i.e. a boolean combination using $\wedge$ and $\vee$ but not using $\neg$.

Consider a word $w \in \sum^\omega$. A *run* of A on w is a $Q \times \mathbb{N}$-labeled tree (T, ρ), where ρ is the labeling function, such that $\rho(\varepsilon) = (q^0, 0)$ and for all $x \in T$, with $\rho(x) = (q,n)$, the set $\{q' \mid c \in \mathbb{N}, x.c \in T, \rho(x.c) = (q', n+1)\}$ satisfies the formula $\tau(q, w(n))$.

For a path P, $\mathrm{Inf}(\rho, P) := \{s \mid$ there are infinitely many $x \in P$ with $\rho(x) \in \{s\} \times \mathbb{N}\}$. A run (T,ρ) of a BAA is accepting if all infinite paths P in T have $\mathrm{Inf}(\rho,P) \cap F \neq \emptyset$.

From LTL to BAA

The translation of LTL formulae into their equivalent BA automata follows the classic approach presented e.g. in [Var94].

Let φ be a LTL formula on the set Ap of atomic propositions, and subf(φ) be the set of subformulae of φ.

The BAA automaton for φ is $A_\varphi = (2^{Ap}, Q, q^0, \tau, F)$ where:

$Q = \{\psi \mid \psi \in \mathrm{subf}(\varphi)$ or $\neg\psi \in \mathrm{subf}(\varphi)\}$, $q^0 = \varphi$,

and $F = \{\psi \mid \psi \in Q$ and ψ has the form $\neg(p\ U\ q)\}$.

The *dual operation* dual(φ) is defined for every formula φ as the formula φ' obtained from φ, by switching true and false, $\vee$, $\wedge$, and by complementing all subformulae of φ.

The transition function is defined as follows, for every $p, q \in Q$ and $a \in 2^{Ap}$:

$\tau(p, a) = \mathrm{true}$ for $p \in Ap$ and $p \in a$

$\tau(p, a) = \mathrm{false}$ for $p \in Ap$ and $p \notin a$

$\tau(p \wedge q, a) = \tau(p, a) \wedge \tau(q, a)$

$\tau(\neg p, a) = \mathrm{dual}(\tau(p, a))$

$\tau(X\ p, a) = p$

$\tau(p\ U\ q, a) = \tau(q, a) \vee (\tau(p, a) \wedge p\ U\ q)$

The transition function is undefined for every case not listed above.

Composition of Past and Future Automata

Let us now consider a separated LTLB specification $G(\mathcal{B}(\varphi_1,.., \varphi_k, \pi_1,.., \pi_m))$, where each π_i is in LTLP and each φ_j is in LTL. By transformation in conjunctive normal form, absorption of the negation into the separated formulae and noticing that $G(p \wedge q) \equiv G(p) \wedge G(q)$, the separated specification can be put in the form $\wedge_i G(\varphi_i \vee \pi_i)$. This operation, while in general exponential in the size of $\mathcal{B}$, is in practice reasonably efficient since, as already noticed, $\mathcal{B}$ is very shallow. Hence, without loss of generality, we may assume that the specification is in the form $G(\varphi \vee \pi)$ (the conjunction being treated very simply with BAA).

For our convenience, we introduce two new propositional symbols, $\underline{\varphi}$ and $\underline{\pi}$, such that $Ap \cap \{\underline{\varphi}, \underline{\pi}\} = \emptyset$. Let $A_{\pi'} = (2^{Ap \cup \{\underline{\varphi}, \underline{\pi}\}}, Q_{\pi'}, \tau_{\pi'}, q^0_{\pi'}, F_{\pi'})$ be the Büchi automaton of Statement 3, considering a pure past formula π' defined as $\underline{\pi} \leftrightarrow \pi$, i.e. such that $L^\omega(A_{\pi'}) = L^\omega(G(\underline{\pi} \leftrightarrow \pi))$; moreover, let $A_{\varphi'} = (2^{Ap \cup \{\underline{\varphi}, \underline{\pi}\}}, Q_{\varphi'}, \tau_{\varphi'}, q^0_{\varphi'}, F_{\varphi'})$ be the BAA automaton, defined as above, such that $L^\omega(A_{\varphi'}) = L^\omega(G(\varphi'))$, where φ' is the purely future formula $\underline{\varphi} \leftrightarrow \varphi$.

Now, let Σ be $2^{Ap \cup \{\underline{\varphi}, \underline{\pi}\}}$ and consider the ω-regular language $L_u = L^\omega(A_{\pi'}) \cap L^\omega(A_{\varphi'}) \cap (\{f \in \Sigma \mid \underline{\varphi} \in f\} \cup \{p \in \Sigma \mid \underline{\pi} \in p\})^\omega$, and the homomorphism $h: 2^{Ap \cup \{\underline{\varphi}, \underline{\pi}\}} \rightarrow 2^{Ap}$, such that $h(x) = x \setminus \{\underline{\varphi}, \underline{\pi}\}$ (where \ denotes set difference).

Statement 4: $h(L_u) = L^{\omega}(G(\varphi \vee \pi))$.

Proof: By construction, we know that $L^{\omega}(A_{\pi'}) = L^{\omega}(G(\underline{\pi}\leftrightarrow\pi))$, and $L^{\omega}(A_{\varphi'})=L^{\omega}(G(\underline{\varphi}\leftrightarrow\varphi))$. That is, $L^{\omega}(A_{\pi'}) = \{w \in 2^{Ap\cup\{\underline{\varphi},\underline{\pi}\}} \mid \text{for every i, } w,i \models \underline{\pi}\leftrightarrow\pi\}$, and $L^{\omega}(A_{\varphi'}) = \{w \in 2^{Ap\cup\{\underline{\varphi},\underline{\pi}\}} \mid \text{for every i, } w,i\models \underline{\varphi}\leftrightarrow\varphi\}$. Let $L_s = L^{\omega}(A_{\pi'}) \cap L^{\omega}(A_{\varphi'})$. L_s is such that $L_s = \{w \in 2^{Ap\cup\{\underline{\varphi},\underline{\pi}\}} \mid \text{for every i, } w,i \models \underline{\pi}\leftrightarrow\pi \text{ and } w,i \models \underline{\varphi}\leftrightarrow\varphi\}$. Let $L_t = (\{f \mid \underline{\varphi}\in f\} \cup \{p \mid \underline{\pi}\in p\})^{\omega}$. This means that $L_t = \{w \in 2^{Ap\cup\{\underline{\varphi},\underline{\pi}\}} \mid \text{for every i, } w,i\models \underline{\pi} \vee \underline{\varphi}\}$. $L_u = L_s \cap L_t = \{w \in 2^{Ap\cup\{\underline{\varphi},\underline{\pi}\}} \mid \text{for every i, } w,i \models (\underline{\pi}\leftrightarrow\pi)\wedge(\underline{\varphi}\leftrightarrow\varphi) \text{ and } w,i \models \underline{\pi}\vee\underline{\varphi}\} = \{w\in 2^{Ap\cup\{\underline{\varphi},\underline{\pi}\}} \mid \text{for every i, } w,i \models (\underline{\pi}\leftrightarrow\pi)\wedge(\underline{\varphi}\leftrightarrow\varphi) \text{ and } w,i \models \pi\vee\varphi\}$. Hence $h(L_u) = \{w\in 2^{Ap} \mid \text{for every i, } w,i\models \pi\vee\varphi\}$, which is $L^{\omega}(G(\varphi \vee \pi))$. □

Being L_u an ω-regular language, there exists a Büchi automaton whose language is $L^{\omega}(G(\varphi \vee \pi))$. In practice we may check "on the fly" whether or not a given string $w \in L_u$: we translate π and φ into Promela processes corresponding to $A_{\pi'}$ and $A_{\varphi'}$, respectively (as we will show in the next section), and then use a special process, called *Coordinator*, which checks at every instant (i.e. for every prefix w_i) if either π or φ (or both) are true. When both subformulae are false, an error is signaled and system halts.

4 Verification

In this section we briefly describe how to implement $A_{\varphi\vee\pi}$ in SPIN, and then we show some experimental results on a practical specification, namely the classical Railway Crossing problem [HM96]. To improve conciseness of LTL, let us introduce the following bounded metric temporal operators:

1. $X^1\phi \leftrightarrow X\phi$, $X^k\phi \leftrightarrow X\, X^{k-1}\phi$ (Y^k is its past analogous);
2. $G_{<k}\, \phi \leftrightarrow X\phi \wedge X^2\phi \wedge .. \wedge X^{k-1}\phi$ ($H_{<k}$ is its past analogous);
3. $F_{<k}\, \phi \leftrightarrow \neg G_{<k}\neg\phi$ ($O_{<k}$ is its past analogous).

The LTL to Promela translation technique is essentially adapted from the original TRIO version presented in [MPSS03]. The main methodological difference resides in the treatment of the past component: in [MPSS03] is mostly *ad hoc*, and unsupported by theoretical results.

The Promela code

The expressiveness of the Promela language makes the implementation of the resulting automaton a quite straightforward task. Indeed, we use the Promela code to directly simulate the composed alternating automaton. Here we recall the technique developed in [MPSS03], adjusted for LTL, for the future component of the specification.

Conceptually, every state of the automaton will correspond to a single type of process (*proctype*). As in classical nondeterministic automata, an or-combination of states ($s_1 \vee s_2$) in the transition function will correspond to a nondeterministic choice (*if ::s_1; ::s_2; fi*). Analogously, an and-combination $s_1 \wedge s_2$ will correspond to the starting of two new processes, having type s_1 and s_2, respectively.

As far as process synchronization is concerned, we have to proceed bottom-up: quite naturally, processes corresponding to simpler subformulae must be evaluated before more complex ones. The system does not require asynchronous communication among processes. In fact, it is possible to determine an arbitrary total evaluation order, starting from the original partial order defined by the *subf* relation. Therefore, we used a single rendezvous channel.

Bounded temporal operators are directly implemented and use simple counting loops and variables to determine where to start and stop evaluating, and to store partial evaluations.

As an example, consider the formula: $G(push \rightarrow G_{<7}(on))$. The non-optimized Promela code contains two process types, one for *G* (called Globally) and one for $G_{<k}$ (called Lasts). In general, it is not necessary to define multiple process types for boolean operators applied to atomic propositions. For instance, in our example the implication '*push* →' can be handled directly within the Globally process.

```
#define MAXP 6 /* maximum number of running Lasts processes */
proctype Globally(chan environment; chan sync) {
```

```
bool push,on; byte n; bool ex[MAXP], dying, result;
chan to_lasts = [0] of {bool, byte}; chan from_lasts = [0] of {bool, bool, byte};
do
:: environment?push,on;
           n = 0;
           do
           :: n < MAXP ->
             if
             :: ex[n] -> to_lasts!on,n;
               from_lasts?dying,result,eval(n);
               if
               :: dying -> ex[n] = 0;
               :: else;
               fi;
               if
               :: !result -> sync!0; goto stop; /* error */
               :: else;
               fi;
             :: else;
             fi;
             n++;
           :: n == MAXP -> break;
           od;
           if
           :: !push -> sync!1;
           :: push -> n = 0;
             do
             :: n < MAXP ->
               if
               :: !ex[n] -> break;
               :: else -> n++;
               fi
             :: n == MAXP -> sync!0; goto stop; /* overflow */
             od;
             ex[n] = 1;
             run Lasts(to_lasts,from_lasts,MAXP,n);
             sync!1;
           fi;
od;
stop: skip;
}

proctype Lasts(chan from_alw; chan to_alw; byte k; byte id) {
bool on;
do
:: from_alw?on,eval(id);
           if
           :: on && k == 1 -> to_alw!1,1,id; break;
           :: on && k > 1 -> to_alw!0,1,id; k--;
           :: !on -> to_alw!1,0,id; break;
           fi;
od;
}
```

In this case, the Globally process may launch at most six different instances of the Lasts process, since the boolean argument of Lasts must be checked in six different instants. This bound is dealt with by the constant definition of MAXP in the very first line of the Promela code.

In the previous piece of code, we use two channels to manage the communication between the Globally process and its children. First, Globally sends to every alive instance of its children the value of *on* coming from the environment, then it reads the results of their evaluation. A Lasts process may send two boolean signals to Globally: the first is about its immediate termination, while the second is the result of

its evaluation. Both the Globally process and the Lasts processes use an identifier (*n* and *id*, respectively) for synchronization purposes.

The past component of the specification is quite naturally and almost immediately translated into a single process, following the approach presented in Section 2.

In traditional model checking, a property is verified against a model of the system (an automaton such as a Promela program). When translating a whole LTLB specification in order to check its satisfiability, however, no automaton model is already present. As a result, a special automaton, called a *generator*, is introduced and added to the process network. The generator simply produces random input values at each instant, then it sends them to the Promela program. The generator is able to generate any system behavior, hence it can also generate input that do not satisfy the specification. In this case an error signal is sent to the Promela program and the system is stopped. In our example this process is not shown but is called *environment*.

The case study

As a case study we considered the standard railroad crossing problem [HM96] (RC, with only one train at a time inside the critical region). It takes between d_m and d_M instants for a train to cross region R, and between h_m and h_M instants to cross the region I. Also, there are at least μ instants between the arrival of a train and of the next one. The bar can be in one of four different positions: *open*, *closed*, *moving up* and *moving down*. The commands to open and close the bar are *go(up)* and *go(down)*, respectively. It takes γ instants for the bar to switch its position from open to down or vice versa. The following relation holds among the temporal constants: $d_M \geq d_m > 0 \wedge h_M \geq h_m > 0 \wedge \mu > d_M + h_M \wedge d_m > \gamma$. The following specification is the LTLB version: axioms are named according to their role in the overall system description, with axioms T1-T9 describing the train movement, axioms B1-B3 illustrating the bar behavior, and axioms C1-C2 accounting for the control strategy.

(T1) $\text{EnterR} \rightarrow G_\mu \neg \text{EnterR}$

(T2) $\text{EnterI} \rightarrow G_\mu \neg \text{EnterI}$

(T3) $\text{ExitI} \rightarrow G_\mu \neg \text{ExitI}$

(T4) $\text{EnterR} \rightarrow X^{d_m}(\text{EnterI} \vee F_{<1+d_M-d_m}\text{EnterI})$

(T5) $\text{EnterI} \rightarrow X^{h_m}(\text{ExitI} \vee F_{<1+h_M-h_m}\text{ExitI})$

(T6) $\text{EnterI} \rightarrow Y^{d_m}(\text{EnterR} \vee O_{<1+d_M-d_m}\text{EnterR})$

(T7) $\text{ExitI} \rightarrow Y^{h_m}(\text{EnterI} \vee O_{<1+h_M-h_m}\text{EnterI})$

(T8) $\text{InR} \leftrightarrow (\text{EnterR} \vee O_{<1+d_M}\text{EnterR}) \wedge (\neg\text{EnterI S EnterR})$

(T9) $\text{InI} \leftrightarrow (\text{EnterI} \vee O_{<1+h_M}\text{EnterI}) \wedge (\neg\text{ExitI S EnterI})$

(M1) $\text{Y closed} \wedge \text{go(up)} \rightarrow \text{mvUp} \wedge G_\gamma\,\text{mvUp} \wedge X^\gamma((\text{open U go(down)}) \vee \text{G open})$

(M2) $\text{Y open} \wedge \text{go(down)} \rightarrow \text{mvDown} \wedge G_\gamma\,\text{mvDown} \wedge X^\gamma((\text{closed U go(up)}) \vee \text{G closed})$

(M3) $H\neg\text{go(down)} \wedge \neg\text{go(down)} \rightarrow \text{open}$

(C1) $\text{go(down)} \leftrightarrow Y^{d_m-\gamma}\,\text{EnterR}$

(C2) $\text{go(up)} \leftrightarrow \text{ExitI}$

The original specification was written in TRIO [MMPSS96]. Its goal was twofold: to provide a formal definition of the RC system (including the environment and the control system under design), and to prove the safety property that, whenever the train is inside the railway crossing, the bar is always down, i.e., the *Prop* formula: *G(inI → closed)*.

We note that the verification we carried out on the RC example did not consider, as it is customary in conventional model-checking, an operational model of the system under design (e.g., a state-transition system coded into a Promela program) but was performed exclusively on the LTLB formulas listed above, and thus it was quite similar in nature to an overall system analysis performed by proving desired properties. In fact, if we call *Spec* the formula G ($\wedge_i$ Ti $\wedge_i$ Bi $\wedge_i$ Ci) we proved, through model checking, the validity of the implication *Spec* $\rightarrow$ *Prop*, which intuitively asserts that, given the characteristics of the environment (axioms for train movement and bar behavior), any system implemented according to the bar control strategy formalized by axioms C1 and C2 would ensure the safety property expressed by formula Prop.

The following table summarizes the results of the verification of RC for different values of the constants. The verification was performed using a PC equipped with a Pentium 4 processor @ 2GHz, 256 MB of RAM, and every computation took less than 2 minutes. The experiments were run for different values of time constants in the original specification (namely, d_M, d_m, h_M, h_m and γ), with time and space complexity increasing with the values.

μ	d_M	d_m	h_M	h_m	γ	Depth	Mem (KB)	States	Transitions
10	5	4	4	3	2	4663	28449	467581	468931
15	7	4	7	3	2	7325	108228	1693890	1697940
20	7	5	12	5	3	10839	173666	2441910	2446810
20	10	8	9	5	3	10601	135569	1882000	1885830
25	15	12	9	7	6	15375	228334	2796640	2801330
25	15	12	9	7	9	> 230 MB			
30	12	10	12	10	3	14093	175710	2315270	2319460

As a comparative experimentation, we ran LTL2BA and Wring on the same specification with m = 7, $h_M = d_M = 3$, $h_m = d_m = 2$, and $\gamma = 1$. LTL2BA crashed after a memory overflow, while Wring was still running after three days and was therefore aborted. Hence, even though we did not implement yet an actual LTL-to-Spin translator, this shows that on large specifications other available approaches are infeasible, while our approach may tackle the problem. This is mainly due to the translation of the alternating automaton into Promela by using statitically-defined, synchronous parallel processes for conjunctive states, as already explored in [MPSS03]: experiments show that careful exploitation of parallelism may actually lead to improvement of performance.

We have also pursued an alternative approach, by eliminating past operators in the RC specification, deriving equivalent LTL formulae. The resulting specification was examined by SPIN, showing that future operators not only make the specification longer and less readable than the original one, but also they may adversely affect the analysis. In fact, SPIN was not able to check the LTL version, running out of memory even for small values of the time constants d_M, d_m, h_M, h_m. This is a confirmation of the validity of our approach, since the simultaneous presence of both past and future operators, far from hampering verification, actually simplified it (because of the deterministic nature of the past).

5 Conclusions

In this paper we presented an approach to model checking LTL+past formulae with SPIN. The idea is to start from formulae in separate normal form, since specifications occurring in practice are usually in this form or they can be easily made separate.

The model is based on a combination of a Büchi deterministic automaton for the past components of the formula and of a one-way alternating automaton for the future component. Experiments with a temporal logic specification of the Railroad Crossing problem show that model and satisfiability checking are possible also for nontrivial specifications. Past formulae also appear much easier to check than their, often

cumbersome, translation into future formulae. Future work will deal with an implementation of an algorithm translating separate form LTL+past formulae into SPIN, following the approach developed in this paper. Notice that other approaches for translating LTL into SPIN often rely on sophisticated algorithms and optimizations, whose running time is at least linear in the size of the resulting automaton. In our case, instead, the translation algorithm appears to be straighforward, since no optimization is explicitly performed on the number of states, and to work in time linear in the size of the LTL formulae.

References

[CKS81] A. Chandra, D. Kozen, and L. Stockmeyer, Alternation. Journal of the Association for Computing Machinery 28, 1 (January 1981), 114-133.

[Eme90] E. Allen Emerson. Temporal and Modal Logic. Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics 1990, J. van Leeuwen, ed., North-Holland Pub. Co./MIT Press, Pages 995-1072.

[BC03] M. Benedetti, A, Cimatti: Bounded Model Checking for Past LTL. TACAS 2003: 18-33

[Gab89] D. Gabbay. The declarative past and imperative future. In H. Barringer, editor, Proceedings of the Colloquium on Temporal Logic and Specifications, vol. 398 of Lecture Notes in Computer Science, pages 409-448. Springer-Verlag, 1989.

[GM01] A. Gargantini, A.Morzenti, Automated Deductive Requirements Analysis of Critical Systems, ACM TOSEM - Transactions On Software Engineering and Methodologies, Vol. 10, no. 3, July 2001, pp. 225-307.

[GMM90] C.Ghezzi, D.Mandrioli, A.Morzenti, TRIO, a logic language for executable specifications of real-time systems, The Journal of Systems and Software, Elsevier Science Publishing, vol.12, no.2, May 1990.

[GO01] P. Gastin, D. Oddoux, Fast LTL to Büchi Automata Translation, Proceedings of CAV'01, Lecture Notes in Computer Science 2102, p. 53-65, 2001.

[GO03] P. Gastin, D. Oddoux, LTL with past and two-way very-weak alternating automata, Proceedings of MFCS'03, Lecture Notes in Computer Science 2747, p. 439-448, 2003.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the Temporal Analysis of Fairness, 7th ASM Symposium on Principles of Programming Languages (POPL’80), January 1980.

[HM96] C. Heitmeyer and D. Mandrioli, editors. Formal Methods for Real-Time Computing, volume 5 of Trends in Software. Wiley, 1996.

[Hol97] G. Holzmann, The Model Checker SPIN, IEEE Transactions on Software Engineering, Vol. 23, 5, May 1997

[Kam68] H. Kamp, Tense Logic and the Theory of Linear Order, PhD Thesis, University of California, 1968.

[KMMP93] Y. Kesten, Z. Manna, H. McGuire and A. Pnueli. A Decision Algorithm for Full Propositional Temporal Logic. In 5th Conference on Computer Aided Verification, Lecture Notes in Computer Science 697, Springer-Verlag, pp. 97-109, 1993.

[Koy90] R. Koymans, Specifying real-time properties with metric temporal logic, Real-Time Systems, 2(4):255--299, 1990.

[KV97] O. Kupferman, M. Vardi, Weak Alternating Automata Are Not That Weak, Proceedings of the Fifth Israel Symposium on Theory of Computing and Systems, ISTCS'97, 1997.

[KPV01] O. Kupferman, N. Piterman, M. Vardi, Extended Temporal Logic Revisited, CONCUR'01, 2001.

[LMS02] F. Laroussinie, N. Markey, and Ph. Schnoebelen, Temporal Logic with Forgettable Past, 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02), July 22-25, 2002, Copenhagen, Denmark, pp. 383-392. IEEE Comp. Soc. Press, 2002.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In R. Parikh, editor, Proceedings of the Conference on Logic of Programs, vol. 193 of LNCS, pages 196-218, Brooklyn, NY, June 1985. Springer.

[Mar02] N. Markey. Past is for free: on the complexity of verifying linear temporal properties with past. In Proc. 9th Int. Workshop on Expressiveness in Concurrency (EXPRESS'2002), Brno, Czech Republic, Aug. 2002, volume 68.2 of Electronic Notes in Theor. Comp. Sci. Elsevier Science, 2002

[MMG92] Morzenti, A., Mandrioli, D., and Ghezzi, C., A Model Parametric Real-Time Logic, ACM Trans. on Programming Languages and Systems 14, 4 (October 1992), 521-573.

[MMM95] D.Mandrioli, S.Morasca, A.Morzenti, Generating Test Cases for Real-Time Systems from Logic Specifications, ACM Trans. On Computer Systems, Vol. 13, No. 4, November 1995. pp.365-398.

[MMPSS96] D. Mandrioli, A. Morzenti, M. Pezzè, P. San Pietro, S. Silva, A Petri Net and Logic Approach to the Specification and Verification of Real Time Systems, in [HM96].

[MPSS03] A. Morzenti, M. Pradella, P. San Pietro, P. Spoletini, Model-checking TRIO specifications in SPIN, to appear in FM 2003: the 12th International FME Symposium, Pisa, Italy -September 8-14, 2003.

[MS94] A. Morzenti, P. San Pietro, Object-Oriented Logic Specifications of Time Critical Systems, ACM Trans. on Softw. Engin. and Meth., vol.3, n.1, Jan.1994, pp. 56-98.

[Pnu77] A. Pnueli. The temporal logic of programs. In 18th FOCS, 46--57, 1977.

[PP03] D. Perrin and J.-E. Pin, Infinite Words (Automata, Semigroups, Logic and Games), Elsevier Publ. (to appear).

[Sch02] P. Schnoebelen, The complexity of Temporal Logic Model Checking, in Advances in Modal Logic, vol. 4(Proc. 4th Int. Workshop), AiML'2002, Toulouse, France, Sep.-Oct. 2002, World Scientific, 2003.

[SB00] F. Somenzi and R. Bloem, Efficient Büchi automata from LTL Formulae, CAV'00, pp.248-263, 2000

[SMM00] P.San Pietro, A.Morzenti, S.Morasca, Generation of Execution Sequences for Modular Time-Critical Systems, IEEE Trans. on Software Engineering, vol. 26 n. 2, Feb. 2000, pp. 128-149, IEEE, New York.

[Tho97] W. Thomas: Automata Theory on Trees and Partial Orders. TAPSOFT, 1997.

[Var94] M. Vardi, An automata-theoretic approach to linear temporal logic, Banff'94, 1994.

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

[Wol87] P. Wolper. On the relation of programs and computations to models of temporal logic. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, Temporal Logic in Specification, pages 75–123, Altrincham, UK, 1987. Springer-Verlag.