Transformational Verification of Linear Temporal Logic

Alberto Pettorossi[1], Maurizio Proietti[2], and Valerio Senni[1]

[1] DISP, University of Rome Tor Vergata, Via del Politecnico 1, I-00133 Rome, Italy
{pettorossi,senni}@disp.uniroma2.it
[2] IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy
proietti@iasi.cnr.it

Abstract. We present a new method for verifying Linear Temporal Logic (LTL) properties of finite state reactive systems based on logic programming and program transformation. We encode a finite state system and an LTL property which we want to verify as a logic program on infinite lists. Then we apply a verification method consisting of two steps. In the first step we transform the logic program that encodes the given system and the given property into a new program belonging to the class of the so-called *linear monadic ω-programs* (which are stratified, linear recursive programs defining nullary predicates or unary predicates on infinite lists). This transformation is performed by applying rules that preserve correctness. In the second step we verify the property of interest by using suitable proof rules for linear monadic ω-programs. These proof rules can be encoded as a logic program which always terminates, if evaluated by using tabled resolution. Although our method uses standard program transformation techniques, the computational complexity of the derived verification algorithm is essentially the same as the one of the Lichtenstein-Pnueli algorithm [9], which uses sophisticated *ad-hoc* techniques.

1 Introduction

Model checking is a very successful technique for verifying finite state reactive systems, such as protocols, concurrent systems, and digital circuits [5]. In model checking the reactive system is formally specified as a Kripke structure and the properties to be verified are specified as formulas of a suitable temporal logic. Among the most popular logics that have been proposed are the temporal logic CTL* and its two fragments: (i) the Computational Tree Logic CTL, and (ii) the Linear-time Temporal Logic LTL (see [5] for a comprehensive account). In this paper we will focus our attention on the logic LTL.

The logic LTL has been shown to be decidable for finite state systems and several algorithms for LTL model checking have been proposed. These algorithms use sophisticated *ad-hoc* techniques based on tableaux [9], symbolic representations using BDDs [2,4], translations to Büchi automata [21], and translations to alternating automata [11].

In this paper we propose a method which is based on very general techniques developed in the field of logic programming. We encode the satisfaction relation of an LTL formula φ with respect to a Kripke structure $\mathcal{K}$ by means of

a stratified logic program $P_{\mathcal{K},\varphi}$. Program $P_{\mathcal{K},\varphi}$ belongs to a class of programs, called ω-programs, which define predicates on infinite lists. Such predicates are needed because the definition of the satisfaction relation of φ is based on the computation paths of $\mathcal{K}$, which are infinite lists of states. The semantics of $P_{\mathcal{K},\varphi}$ is the *perfect model* $M(P_{\mathcal{K},\varphi})$ [15], which is defined in terms of a non-Herbrand interpretation for infinite lists.

Our verification method consists of two steps. In the first step we transform the program $P_{\mathcal{K},\varphi}$ into a *linear monadic ω-program*, that is, a stratified, linear recursive program which defines nullary or unary predicates on infinite lists. This transformation is performed by applying unfold/fold transformation rules [8,20] according to a strategy which is a variant of the strategy for the elimination of multiple occurrences of variables described in [13]. The use of those unfold/fold rules according to the given strategy guarantees the preservation of the perfect model of $P_{\mathcal{K},\varphi}$ [8,17,18].

In the second step of our verification method we use suitable proof rules for linear monadic ω-programs. Those proof rules are sound and complete, that is, any given quantified literal L is true in the perfect model of a linear monadic ω-program P iff we can prove that $M(P) \models L$ holds by using those proof rules. Moreover, those rules can be encoded in a straightforward way as a logic program which always terminates if we evaluate it by using tabled resolution [3,19]. As a consequence of this termination property, we get the decidability of the problem of verifying whether or not a quantified literal is true in the perfect model of a linear monadic ω-program.

We will prove that our verification method based on very general transformation techniques, has essentially the same time complexity of the algorithms based on *ad-hoc* techniques presented in [2,4,9,21].

The paper is structured as follows. In Section 2 we introduce the class of ω-programs and we present the encoding as an ω-program of the satisfaction relation for any given Kripke structure and LTL formula. In Section 3 we present our verification method. In particular, in Sections 3.1 and 3.2 we present the transformation rules and the strategy which allow us to transform an ω-program $P_{\mathcal{K},\varphi}$ into a linear monadic ω-program, in Section 3.3 we present the proof rules for linear monadic ω-programs and the encoding of those proof rules as logic programs, and in Section 3.4 we discuss the computational complexity of the verification technique. Finally, in Section 4 we discuss related work in the area of model checking and logic programming.

2 Encoding LTL Model Checking as a Logic Program

In this section we describe a method which, given a Kripke structure $\mathcal{K}$ and an LTL formula φ, allows us to construct a logic program $P_{\mathcal{K},\varphi}$ that defines a predicate *prop* such that φ is true in $\mathcal{K}$, written $\mathcal{K} \models \varphi$, iff *prop* is true in the perfect model of $P_{\mathcal{K},\varphi}$, written $M(P_{\mathcal{K},\varphi}) \models \mathit{prop}$. Thus, the problem of checking whether or not $\mathcal{K} \models \varphi$, also called the problem of model checking φ with respect to $\mathcal{K}$, is reduced to the problem of checking whether or not $M(P_{\mathcal{K},\varphi}) \models \mathit{prop}$.

For a detailed definition of the logic LTL we refer to [5] (see also Appendix A). Throughout the paper we will consider a Kripke structure $\mathcal{K}$ defined as a 4-tuple $\langle \Sigma, I, \rho, \lambda \rangle$, where: (i) $\Sigma = \{s_1, \ldots, s_h\}$ is a finite set of *states*, (ii) $I \subseteq \Sigma$ is a set of *initial states*, (iii) $\rho \subseteq \Sigma \times \Sigma$ is a total *transition relation*, and (iv) $\lambda : \Sigma \rightarrow \mathcal{P}(Elem)$ is a total function that assigns to a state $s \in \Sigma$ a subset $\lambda(s)$ of the set *Elem* of *elementary properties*. A *computation path* of $\mathcal{K}$ is an *infinite* list $[a_0\, a_1 \ldots]$ of states such that $a_0 \in I$ and, for every $i \geq 0$, $(a_i, a_{i+1}) \in \rho$. LTL formulas are constructed by using the elementary properties in *Elem*, the logical connectives $\neg$ and $\wedge$, and the temporal operators X (*next time*) and U (*until*).

Since the definition of the satisfaction relation $\mathcal{K} \vDash \varphi$ refers to the computation paths of the Kripke structure $\mathcal{K}$ and these paths are infinite lists, in order to encode that relation as a predicate defined by the program $P_{\mathcal{K},\varphi}$ we need an extended form of logic programs where predicates have arguments that denote infinite lists. To this purpose in the following section we introduce the class of *ω-programs*.

2.1 Syntax and Semantics of ω-Programs

Let us consider a first order language $\mathcal{L}_\omega$ given by a set *Var* of variables, a set *Fun* of function symbols, and a set *Pred* of predicate symbols. We assume that *Fun* includes: (i) the set Σ of states of the Kripke structure, each state being a constant of $\mathcal{L}_\omega$, (ii) the set *Elem* of the elementary properties of the Kripke structure, each elementary property being a constant of $\mathcal{L}_\omega$, and (iii) the binary function symbol $[\_|\_]$, denoting the constructor of infinite lists. Thus, $[H|T]$ is an infinite list whose head is H and whose tail is the infinite list T.

We assume that $\mathcal{L}_\omega$ is a typed language [10] with three basic types: (i) `fterm`, which is the type of finite terms, (ii) `state`, which is the type of states, and (iii) `ilist`, which is the type of infinite lists of states. Every function symbol in $Fun - (\Sigma \cup \{[\_|\_]\})$, with arity $n\,(\geq 0)$, has type $\texttt{fterm} \times \cdots \times \texttt{fterm} \rightarrow \texttt{fterm}$, where `fterm` occurs n times to the left of $\rightarrow$. Every function symbol in Σ has type `state`. The function symbol $[\_|\_]$ has type $\texttt{state} \times \texttt{ilist} \rightarrow \texttt{ilist}$. A predicate symbol of arity $n\,(\geq 0)$ in *Pred* has type of the form $\tau_1 \times \cdots \times \tau_n$, where $\tau_1, \ldots, \tau_n \in \{\texttt{fterm}, \texttt{state}, \texttt{ilist}\}$. ω-programs are logic programs constructed as usual from symbols in the typed language $\mathcal{L}_\omega$ [10]. In what follows, for reasons of simplicity, we will feel free to say 'programs', instead of 'ω-programs'.

The *definition* of a predicate p in a program P is the set of all clauses of P whose head predicate is p. If f is a term or a formula, then by $vars(f)$ we denote the set of variables occurring in f. By $\forall(\varphi)$ and $\exists(\varphi)$ we denote, respectively, the *universal closure* and the *existential closure* of the formula φ.

An interpretation for our typed language $\mathcal{L}_\omega$, called ω-interpretation, is given as follows. Let HU be the Herbrand universe constructed from the set $Fun - (\Sigma \cup \{[\_|\_]\})$ of function symbols and let Σ^ω be the set of infinite lists of states. An ω-interpretation I is an interpretation such that: (i) to the types `fterm`, `state`, and `ilist`, I assigns the sets HU, Σ, and Σ^ω, respectively, (ii) to the function symbol $[\_|\_]$, I assigns the function $[\_|\_]_I$ such that, for any state $s \in \Sigma$ and infinite list $[s_1, s_2, \ldots] \in \Sigma^\omega$, $[s|[s_1, s_2, \ldots]]_I$ is the infinite list $[s, s_1, s_2, \ldots]$,

(iii) I is an Herbrand interpretation for all function symbols in $Fun - (\Sigma \cup \{[\_|\_]\})$, and (iv) to every n-ary predicate $p \in Pred$ of type $\tau_1 \times \ldots \times \tau_n$, I assigns a relation on $D_1 \times \cdots \times D_n$, where, for $i = 1, \ldots, n$, D_i is either HU or Σ or Σ^ω, if τ_i is either `fterm` or `state` or `ilist`, respectively. We say that an ω-interpretation I is an *ω-model* of a program P iff for every clause $\gamma \in P$ we have that $I \models \forall(\gamma)$. Similarly to the case of logic programs, we can define the (*locally*) *stratified* ω-programs and we have that every (locally) stratified ω-program P has a unique *perfect ω-model* (or *perfect model*, for short) denoted by $M(P)$ [1,15].

Definition 1 (Linear Monadic ω-Programs). A *linear monadic ω-clause* is a clause of one of the following forms:

$p_0 \leftarrow$	$p_0 \leftarrow q_0$	$p_0 \leftarrow \neg q_0$
$p_0 \leftarrow q_1(L)$	$p_0 \leftarrow \neg q_1(L)$	
$p_1([s\|L]) \leftarrow$	$p_1([s\|L]) \leftarrow q_1(L)$	$p_1([s\|L]) \leftarrow \neg q_1(L)$

where: (i) s is a constant of type `state` and (ii) L a variable of type `ilist`. A *linear monadic ω-program* is a stratified, finite set of linear monadic ω-clauses.

Example 1. The following set of clauses is an example of ω-program:

$p([a\|L]) \leftarrow p(L)$	$p([a\|L]) \leftarrow \neg q(L)$	$q([a\|L]) \leftarrow q(L)$
$p([b\|L]) \leftarrow p(L)$	$p([b\|L]) \leftarrow \neg q(L)$	$q([b\|L]) \leftarrow$

Every infinite list in $\{a, b\}^\omega$ that satisfies predicate p is a list in $(a + b)^+ a^\omega$. Indeed, $q(L)$ holds for every infinite list L which has an occurrence of b.

2.2 Encoding the LTL Satisfaction Relation as an ω-Program

Given a Kripke structure $\mathcal{K}$ and an LTL formula φ, we introduce a locally stratified ω-program $P_{\mathcal{K},\varphi}$ which defines, among others, the following three predicates: (i) the unary predicate *path*, such that $path(\pi)$ holds iff π is an infinite list representing a computation path of $\mathcal{K}$, (ii) the binary predicate *sat*, which encodes the satisfaction relation for LTL formulas in the sense that for all paths π and LTL formulas ψ, we have that $\mathcal{K}, \pi \models \psi$ iff $M(P_{\mathcal{K},\varphi}) \models sat(\pi, \psi)$, and (iii) the nullary predicate *prop*, which holds iff φ holds on all computation paths of $\mathcal{K}$, that is, $M(P_{\mathcal{K},\varphi}) \models prop$ iff $\mathcal{K} \models \varphi$.

In the terms that encode LTL formulas, such as the second argument of the predicate *sat*, we will use the function symbols x and u standing for the operator symbols X and U, respectively.

Definition 2 (Encoding Program). Given a Kripke structure $\mathcal{K} = \langle \Sigma, I, \rho, \lambda \rangle$ and an LTL formula φ, the *encoding program* $P_{\mathcal{K},\varphi}$ is the following ω-program:

1. $prop \leftarrow \neg\, p_1$
2. $p_1 \leftarrow p_2(P)$
3. $p_2(P) \leftarrow path(P) \wedge sat(P, \neg\varphi)$
4. $path([X|P]) \leftarrow initial(X) \wedge \neg\, nopath([X|P])$

5.1 $nopath([s_1|P]) \leftarrow q_1(P) \qquad q_1([s_{11}|P]) \leftarrow \quad \ldots \quad q_1([s_{1\,k_1}|P]) \leftarrow$

$\vdots$

5.m $nopath([s_m|P]) \leftarrow q_m(P) \qquad q_m([s_{m1}|P]) \leftarrow \quad \ldots \quad q_m([s_{m\,k_m}|P]) \leftarrow$

6. $nopath([X|P]) \leftarrow nopath(P)$
7. $sat([X|P], E) \leftarrow elem(E, X)$
8. $sat(P, \neg F) \leftarrow \neg\, sat(P, F)$
9. $sat(P, F_1 \wedge F_2) \leftarrow sat(P, F_1) \wedge sat(P, F_2)$
10. $sat([X|P], x(F)) \leftarrow sat(P, F)$
11. $sat(P, u(F_1, F_2)) \leftarrow sat(P, F_2)$
12. $sat([X|P], u(F_1, F_2)) \leftarrow sat([X|P], F_1) \wedge sat(P, u(F_1, F_2))$

together with the clauses defining the predicates *initial* and *elem*, where:
(1) $initial(s)$ holds iff $s \in I$, for every state $s \in \Sigma$;
(2) $elem(e, s)$ holds iff $e \in \lambda(s)$, for every property $e \in Elem$ and state $s \in \Sigma$;
(3) $nopath(P)$ holds if P is an infinite list $[a_0, a_1, \ldots]$ of states which is *not* a computation path in $\mathcal{K}$, that is, for some $i \geq 0$, we have that $(a_i, a_{i+1}) \notin \rho$; and
(4) clauses 5.1–5.m are obtained as follows. Let $\{s_1, \ldots, s_m\}$ be the subset of Σ such that, for $i = 1, \ldots, m$, there exists $s \in \Sigma$ for which $(s_i, s) \notin \rho$. For $i = 1, \ldots, m$, the clauses 5.i are:

5.i $nopath([s_i|P]) \leftarrow q_i(P)$ and for all $s \in \Sigma$ such that $(s_i, s) \notin \rho$, $q_i([s|P]) \leftarrow$.

Clauses 1, 2, and 3 of the above Definition 2 stipulate that *prop* holds iff the formula $\forall P\, (path(P) \rightarrow sat(P, \varphi))$ holds. Since the universal quantifier and the implication connective cannot appear in the body of a clause, we have defined the predicate *prop* starting from the equivalent formula $\neg\, \exists P\, (path(P) \wedge sat(P, \neg\, \varphi))$. Indeed, (i) $p_2(P)$ holds iff $path(P) \wedge sat(P, \neg\, \varphi)$, (ii) p_1 holds iff $\exists P\, p_2(P)$ holds, and (iii) *prop* holds iff $\neg\, p_1$ holds.

Clauses 4–6 stipulate that $path(P)$ holds iff for every pair (a_i, a_{i+1}) of consecutive elements on the infinite list P we have that $(a_i, a_{i+1}) \in \rho$. Similarly to the encoding of the predicate *prop* above, we have defined the predicate *path* by using existential quantification and negation, instead of universal quantification and implication. Indeed, clauses 5.1–5.m and 6 stipulate that $nopath(P)$ holds iff there exist two consecutive elements a_i and a_{i+1} of the list P such that $(a_i, a_{i+1}) \notin \rho$. Clause 6 is required for allowing the two consecutive elements a_i and a_{i+1} to occur at any position in P.

Clauses 7–12 define the satisfaction relation $sat(P, \varphi)$ by cases, according to the structure of the formula φ.

The program $P_{\mathcal{K},\varphi}$ is locally stratified w.r.t. the stratification function σ from ground literals to natural numbers defined as follows (where, for an LTL formula ψ, we denote by $|\psi|$ the number of occurrences of function symbols in ψ): σ always returns 0, except that, for all infinite lists $\pi \in \Sigma^\omega$ and LTL formulas ψ, (i) $\sigma(prop) = |\varphi| + 2$, (ii) $\sigma(p_1) = \sigma(p_2(\pi)) = |\varphi| + 1$, (iii) $\sigma(path(\pi)) = 1$, (iv) $\sigma(sat(\pi, \psi)) = |\psi| + 1$, and (v) for every ground atom A, $\sigma(\neg A) = \sigma(A) + 1$.

Example 2. Let us consider the set $Elem = \{true, a, b\}$ of elementary properties and the Kripke structure $\mathcal{K} = \langle \Sigma, I, \rho, \lambda \rangle$, where: (i) Σ is $\{s_1, s_2\}$, (ii) $I = \Sigma$, (iii) ρ is the transition relation $\{(s_1, s_2), (s_2, s_1), (s_2, s_2)\}$, and (iv) λ is the function such that $\lambda(s_1) = \{a\}$ and $\lambda(s_2) = \{b\}$. Let us also consider the formula $\varphi = \neg\, (true\, \mathsf{U}\, (a \wedge \neg\, (true\, \mathsf{U}\, b)))$ (that is, $\varphi = \mathsf{G}(a \rightarrow \mathsf{F}b)$). The encoding program $P_{\mathcal{K},\varphi}$, where we wrote $u(true,\ a \wedge \neg\, u(true, b))$, instead of $\neg\varphi$, is as follows:

1. $prop \leftarrow \neg p_1$
2. $p_1 \leftarrow p_2(P)$
3. $p_2(P) \leftarrow path(P) \ \wedge \ sat(P, u(true, (a \wedge \neg\, u(true, b))))$
4. $path([X|P]) \leftarrow initial(X) \wedge \neg\, nopath([X|P])$
5. $nopath([s_1|P]) \leftarrow q_1(P) \qquad\qquad q_1([s_1|P]) \leftarrow$
6. $nopath([X|P]) \leftarrow nopath(P)$

together with clauses 7–12 of Definition 2 defining the predicate *sat*, and the following clauses defining the predicates *initial* and *elem*:

$initial(s_1) \leftarrow \quad initial(s_2) \leftarrow \quad elem(a, s_1) \leftarrow \quad elem(b, s_2) \leftarrow \quad elem(true, X) \leftarrow$

Theorem 1 (Correctness of the Encoding Program). *Let $P_{\mathcal{K},\varphi}$ be the encoding program for a Kripke structure $\mathcal{K}$ and an LTL formula φ. Then, $\mathcal{K} \models \varphi$ iff $M(P_{\mathcal{K},\varphi}) \models prop$.*

3 Transformational LTL Model Checking

In this section we present a technique based on program transformation for checking whether or not, for any given structure $\mathcal{K}$ and LTL formula φ, $M(P_{\mathcal{K},\varphi}) \models prop$ holds, where $P_{\mathcal{K},\varphi}$ is constructed as indicated in Definition 2 above. Our technique consists of two steps. In the first step we transform the ω-program $P_{\mathcal{K},\varphi}$ into a *linear monadic* ω-program T such that $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$. In the second step we check whether or not $M(T) \models prop$ holds by using a proof system for linear monadic ω-programs.

3.1 Unfold/Fold Transformation Rules

Now we introduce the transformation rules that will be used for transforming ω-programs into linear monadic ω-programs. These rules are specialized versions of the familiar unfold/fold rules (see, for instance, [8,20]).

A *transformation sequence* is a sequence $P_0, \ldots, P_n$ of ω-programs, where for $0 \leq k \leq n-1$, program P_{k+1} is derived from program P_k by the application of a transformation rule as indicated below. In what follows we assume that $\Sigma = \{s_1, \ldots, s_h\}$ is the set of states of the given Kripke structure $\mathcal{K}$.

R1. Definition Introduction. Let us consider the following $m\,(\geq 1)$ clauses:

$\delta_1 \colon newp(X_1, \ldots, X_r) \leftarrow B_1, \quad \ldots, \quad \delta_m \colon newp(X_1, \ldots, X_r) \leftarrow B_m,$

such that: (i) *newp* is a predicate symbol that does not occur in $\{P_0, \ldots, P_k\}$, (ii) $X_1, \ldots, X_r$ are distinct variables, (iii) for $i = 1, \ldots, m$, $vars(B_i) = \{X_1, \ldots, X_r\}$, and (iv) every predicate symbol occurring in $\{B_1, \ldots, B_m\}$ also occurs in P_0. By *definition introduction* (or *definition*, for short) from program P_k we derive the program $P_{k+1} = P_k \cup \{\delta_1, \ldots, \delta_m\}$.

For $0 \leq k \leq n$, we denote by $Defs_k$ the set of clauses introduced by using rule R1 during the transformation sequence $P_0, \ldots, P_n$. In particular, $Defs_0 = \emptyset$.

R2. Definition Elimination. By *definition elimination* w.r.t. a predicate symbol p, from program P_k we derive the new program $P_{k+1} = \{\gamma \in P_k \mid$ the head

predicate of γ is either p or a predicate on which p depends} (see [1] for the definition of the dependency relation).

R3. Instantiation. Let $\gamma\colon H \leftarrow B$ be a clause in program P_k, L be a variable of type `ilist` in γ, and M be a variable of type `ilist` not occurring in γ. By *instantiation* of L in clause γ we derive the clauses:

$\gamma_1\colon (H \leftarrow B)\{L/[s_1|M]\}, \quad \ldots, \quad \gamma_h\colon (H \leftarrow B)\{L/[s_h|M]\}$

(recall that $\{s_1, \ldots, s_h\}$ is the set Σ of states of the given Kripke structure $\mathcal{K}$) and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\gamma_1, \ldots, \gamma_h\}$.

R4. Positive Unfolding. Let $\gamma\colon H \leftarrow G_L \wedge A \wedge G_R$ be a clause in program P_k and let P'_k be a variant of P_k without variables in common with γ. Let

$\gamma_1\colon K_1 \leftarrow B_1, \quad \ldots, \quad \gamma_m\colon K_m \leftarrow B_m$, for $m \geq 0$,

be all clauses of program P'_k such that A is unifiable with $K_1,\ldots,K_m$ with most general unifiers $\vartheta_1,\ldots,\vartheta_m$, respectively. By *unfolding clause γ w.r.t. the positive literal A* we derive the clauses

$\eta_1\colon (H \leftarrow G_L \wedge B_1 \wedge G_R)\vartheta_1, \quad \ldots, \quad \eta_m\colon (H \leftarrow G_L \wedge B_m \wedge G_R)\vartheta_m$

and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_m\}$.

R5. Negative Unfolding. Let $\gamma\colon H \leftarrow G_L \wedge \neg A \wedge G_R$ be a clause in program P_k and let P'_k be a variant of P_k without variables in common with γ. Let

$\gamma_1\colon K_1 \leftarrow B_1, \quad \ldots, \quad \gamma_m\colon K_m \leftarrow B_m$, for $m \geq 0$,

be all clauses of program P'_k such that A is unifiable with $K_1,\ldots,K_m$ with most general unifiers $\vartheta_1,\ldots,\vartheta_m$, respectively. Assume that: (i) for $j = 1, \ldots, m$, $A = K_j\vartheta_j$, that is, A is an instance of K_j, (ii) for $j = 1, \ldots, m$, $vars(K_j) = vars(B_j)$, and (iii) from $G_L \wedge \neg(B_1\vartheta_1 \vee \ldots \vee B_m\vartheta_m) \wedge G_R$ we get an equivalent disjunction $Q_1 \vee \ldots \vee Q_r$ of conjunctions of literals, with $r \geq 0$, by first pushing $\neg$ inside and then pushing $\vee$ outside. By *unfolding clause γ w.r.t. the negative literal $\neg A$* we derive the clauses

$\eta_1\colon H \leftarrow Q_1, \quad \ldots, \quad \eta_r\colon H \leftarrow Q_r,$

and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_r\}$.

R6. Positive Folding. Let γ be a clause in P_k and let $Defs'_k$ be a variant of $Defs_k$ without variables in common with γ. Suppose that there exists a predicate in $Defs'_k$ whose definition consists of the clause

$\delta\colon K \leftarrow B$, where $vars(K) = vars(B)$.

Suppose that there exists a substitution ϑ such that γ is of the form $H \leftarrow B\vartheta$. By *folding* clause γ using clause δ we derive the clause $\eta\colon H \leftarrow K\vartheta$ and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

R7. Negative Folding. Let γ be a clause in P_k and let $Defs'_k$ be a variant of $Defs_k$ without variables in common with γ. Suppose that there exists a predicate in $Defs'_k$ whose definition consists of the clauses

$\delta_1\colon K \leftarrow A_1, \quad \ldots, \quad \delta_m\colon K \leftarrow A_m$

where, for $j = 1, \ldots, m$, A_j is an atom and $vars(K) = vars(A_j)$. Suppose that there exists a substitution ϑ such that γ is of the form $H \leftarrow \neg A_1\vartheta \wedge \ldots \wedge \neg A_m\vartheta$. By *folding clause γ using clauses $\delta_1, \ldots, \delta_m$* we derive the clause $\eta\colon H \leftarrow \neg K\vartheta$ and from program P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

3.2 The Transformation Strategy

Now we present a transformation strategy which terminates for every input program $P_{\mathcal{K},\varphi}$ and produces a linear monadic ω-program T such that $M(P_{\mathcal{K},\varphi}) \models \mathit{prop}$ iff $M(T) \models \mathit{prop}$.

Our strategy makes use of the transformation rules presented in Section 3.1 and it is a variant of the transformation strategy for eliminating the so-called *unnecessary variables* presented in [13]. The strategy starts off from the clause $p_2(P) \leftarrow \mathit{path}(P) \wedge \mathit{sat}(P, \neg\varphi)$ (see Definition 2) and iterates a sequence of applications of the three procedures: *instantiate*, *unfold*, and *define-fold*. At each iteration, the set *InDefs* of input clauses is transformed into a set *Es* of linear monadic ω-clauses, by possibly introducing some auxiliary (not linear monadic) clauses *NewDefs*. These auxiliary clauses are given in input to a subsequent iteration of the strategy until no more auxiliary clauses are introduced. Thus, our strategy terminates when all clauses are transformed into linear monadic ω-clauses without the need for new auxiliary clauses. As a final step of our strategy, we apply the definition elimination rule and we keep only the clauses for *prop* and for the predicates on which *prop* depends.

The Transformation Strategy.
Input: An ω-program $P_{\mathcal{K},\varphi}$, for a Kripke structure $\mathcal{K}$ and an LTL formula φ.
Output: A linear monadic ω-program T such that $M(P_{\mathcal{K},\varphi}) \models \mathit{prop}$ iff $M(T) \models \mathit{prop}$.

$T := P_{\mathcal{K},\varphi}$; $\quad \mathit{Defs} := \{p_2(P) \leftarrow \mathit{path}(P) \wedge \mathit{sat}(P, \neg\varphi)\}$; $\quad \mathit{InDefs} := \mathit{Defs}$;
while $\mathit{InDefs} \neq \emptyset$ **do**
  $\mathit{instantiate}(\mathit{InDefs}, \mathit{Cs})$; $\; T := (T - \mathit{InDefs}) \cup \mathit{Cs}$;
  $\mathit{unfold}(\mathit{Cs}, \mathit{Ds})$; $\; T := (T - \mathit{Cs}) \cup \mathit{Ds}$;
  $\mathit{define\text{-}fold}(\mathit{Ds}, \mathit{Defs}, \mathit{NewDefs}, \mathit{Es})$; $\; T := (T - \mathit{Ds}) \cup \mathit{NewDefs} \cup \mathit{Es}$;
  $\mathit{Defs} := \mathit{Defs} \cup \mathit{NewDefs}$; $\; \mathit{InDefs} := \mathit{NewDefs}$
od;
$T := \{\gamma \in T \mid$ the head predicate of γ is either *prop* or a predicate on which *prop* depends$\}$.

Now we describe the three procedures *instantiate*, *unfold*, and *define-fold* that are used in the transformation strategy. When describing these procedures we will rely on the fact that during the application of the transformation strategy we derive clauses with occurrences of a single variable of type `ilist`.

The *instantiate* procedure consists in applying rule R3 to each clause B in the set *InDefs*.

The *instantiate* **Procedure.**
Input: A set *InDefs* of clauses. *Output*: A set *Cs* of clauses.

$\mathit{Cs} := \bigcup_{B \in \mathit{InDefs}} \{C \mid C$ is derived by instantiation of a variable L of type `ilist` occurring in $B\}$.

The *unfold* procedure applies rules R4 and R5 to a set of clauses of the form $H \leftarrow G_1 \wedge L \wedge G_2$. The procedure applies positive unfolding if L is a positive literal, and negative unfolding if L is a negative literal.

The *unfold* Procedure.
Input: A set Cs of clauses in program T. *Output*: A set Ds of clauses.

$Ds := Cs$;
while there exists a clause D in Ds of the form $H \leftarrow G_1 \wedge L \wedge G_2$, where L is either an atom A or a negated atom $\neg A$ and for all clauses $K \leftarrow B$ in T *either* A and K are not unifiable *or* A is an instance of K **do**
$Ds := (Ds - \{D\}) \cup \{U \mid U$ is a clause derived by unfolding D w.r.t. $L\}$
od

The *define-unfold* procedure applies the positive and negative folding rules R6 and R7. The procedure transforms every clause in Ds, which has been obtained by unfolding, into a linear monadic ω-clause. Each clause D in Ds is of the form $p([s|X]) \leftarrow L_1 \wedge \ldots \wedge L_m$. If among $L_1, \ldots, L_m$ there is at least one positive literal, then the procedure introduces a new clause N of the form $newp(X) \leftarrow L_1 \wedge \ldots \wedge L_m$, unless (a variant of) such a clause N was added to *Defs* in a previous transformation step. Then D is folded using N (by applying rule R6), thereby deriving a linear monadic ω-clause of the form $p([s|X]) \leftarrow newp(X)$. The new clause N is added to the set *Defs* of clauses that can be used for subsequent folding steps and to the set *InDefs* of clauses to be processed in a subsequent iteration of the strategy.

If $L_1, \ldots, L_m$ are all negative literals of the form $\neg A_1, \ldots, \neg A_m$, respectively, then the procedure introduces m new clauses N_1: $newp(X) \leftarrow A_1$, ..., N_m: $newp(X) \leftarrow A_m$, unless (variants of) such clauses were added to *Defs* in a previous transformation step. Then D is folded using $N_1, \ldots, N_m$ (by applying rule R7), thereby deriving a linear monadic ω-clause of the form $p([s|X]) \leftarrow \neg newp(X)$. The new clauses $N_1, \ldots, N_m$ are added to the sets *Defs* and *InDefs*.

The *define-fold* Procedure.
Input: (i) A set Ds of clauses of the form $p([s|X]) \leftarrow G_1$, where $vars(G_1) \subseteq \{X\}$, and (ii) a set *Defs* of clauses;
Output: (i) A set *NewDefs* of clauses of the form $newp(X) \leftarrow G_3$, where $\{X\} = vars(G_3)$, and (ii) a set Es of linear monadic ω-clauses.

$NewDefs := \emptyset$; $Es := \emptyset$;
for each clause $D \in Ds$ of the form $p([s|X]) \leftarrow L_1 \wedge \ldots \wedge L_m$ **do**
if D is a linear monadic ω-clause **then** $Es := Es \cup \{D\}$ **else**
(*Case* 1. *Positive Define-Fold*)
if for some $i \in \{1, \ldots, m\}$, L_i is a positive literal
then **if** there exists no clause N of the form $newp(X) \leftarrow L_1 \wedge \ldots \wedge L_m$ such that: (i) (a variant of) N belongs to $Defs \cup NewDefs$ and (ii) *newp* does not occur in $(Defs \cup NewDefs) - \{N\}$
then $NewDefs := NewDefs \cup \{newp(X) \leftarrow L_1 \wedge \ldots \wedge L_m\}$, where

newp is a predicate not occurring in $Defs \cup NewDefs$ **fi**;
$Es := Es - \{D\} \cup \{p([s|X]) \leftarrow newp(X)\}$ **fi**;
(*Case* 2. *Negative Define-Fold*)
if for all $i \in \{1, \ldots, m\}$, L_i is a negative literal $\neg A_i$
then **if** there exists no set *Ns*: $\{newp(X) \leftarrow A_1, \ldots, newp(X) \leftarrow A_m\}$ of clauses such that: (i) $Ns \subseteq Defs \cup NewDefs$ (modulo variants) and (ii) *newp* does not occur in $(Defs \cup NewDefs) - Ns$
then $NewDefs := NewDefs \cup \{newp(X) \leftarrow A_1, \ldots, newp(X) \leftarrow A_m\}$, where *newp* is a predicate not occurring in $Defs \cup NewDefs$ **fi**;
$Es := Es - \{D\} \cup \{p([s|X]) \leftarrow \neg newp(X)\}$ **fi**
od

The correctness of our transformation strategy can be proved by showing that when the transformation rules R1–R7 are applied according to the strategy, the perfect model of $P_{\mathcal{K},\varphi}$ is preserved (see [8,17,18] for analogous proofs).

The termination of the transformation strategy follows from the termination of the procedures *instantiate*, *unfold*, and *define-fold*, and from the fact that the while loop can be executed only a finite number of times. Indeed, (i) the strategy terminates when no new clauses are introduced by the *define-fold* procedure and, thus, $InDefs = \emptyset$, (ii) only a finite number of new clauses can be introduced by the *define-fold* procedure, because every new clause is of the form $newp(X) \leftarrow L_1 \wedge \ldots \wedge L_m$, where for $i = 1, \ldots, n$, L_i is either an atom A_i or a negated atom $\neg A_i$ and A_i belongs to the set $\{path(X), initial(X), nopath(X), q_1(X), \ldots, q_m(X)\} \cup \{sat(X, \psi) \mid \psi \text{ is a subformula of } \varphi\}$.

Theorem 2 (Correctness and Termination of the Transformation Strategy). *Let $P_{\mathcal{K},\varphi}$ be the encoding program for a Kripke structure $\mathcal{K}$ and an LTL formula φ. The transformation strategy terminates for the input program $P_{\mathcal{K},\varphi}$ and returns an output program T such that: (i) T is a linear monadic ω-program and (ii) $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$.*

Example 3. Let us consider program $P_{\mathcal{K},\varphi}$ of Example 2. Our transformation strategy starts off from the sets of clauses $T = P_{\mathcal{K},\varphi}$ and $Defs = InDefs = \{3\}$, where:

3. $p_2(P) \leftarrow path(P) \wedge sat(P, u(true, (a \wedge \neg\, u(true, b))))$

The first execution of the loop body of our strategy applies the *instantiate* procedure to the set *InDefs*. We get the set of clauses $Cs = \{3.1, 3.2\}$, where:

3.1. $p_2([s_1|P]) \leftarrow path([s_1|P]) \wedge sat([s_1|P], u(true, (a \wedge \neg\, u(true, b))))$
3.2. $p_2([s_2|P]) \leftarrow path([s_2|P]) \wedge sat([s_2|P], u(true, (a \wedge \neg\, u(true, b))))$

Then, by applying the *unfold* procedure to the set *Cs* we get the set $Ds = \{3.3, 3.4, 3.5\}$, where:

3.3. $p_2([s_1|P]) \leftarrow \neg\, q_1(P) \wedge \neg\, nopath(P) \wedge \neg\, sat(P, u(true, b))$
3.4. $p_2([s_1|P]) \leftarrow \neg\, q_1(P) \wedge \neg\, nopath(P) \wedge sat(P, u(true, (a \wedge \neg\, u(true, b))))$
3.5. $p_2([s_2|P]) \leftarrow \neg\, nopath(P) \wedge sat(P, u(true, (a \wedge \neg\, u(true, b))))$

Finally, by applying the *define-fold* procedure, we get the sets $NewDefs = \{13, 14, 15, 16, 17\}$ and $Es = \{3.6, 3.7, 3.8\}$, where:

13. $p_3(P) \leftarrow q_1(P)$
14. $p_3(P) \leftarrow nopath(P)$
15. $p_3(P) \leftarrow sat(P, u(true, b))$
16. $p_4(P) \leftarrow \neg\, q_1(P) \wedge \neg\, nopath(P) \wedge sat(P, u(true, (a \wedge \neg\, u(true, b))))$
17. $p_5(P) \leftarrow \neg\, nopath(P) \wedge sat(P, u(true, (a \wedge \neg\, u(true, b))))$

3.6. $p_2([s_1|P]) \leftarrow \neg\, p_3(P)$
3.7. $p_2([s_1|P]) \leftarrow p_4(P)$
3.8. $p_2([s_2|P]) \leftarrow p_5(P)$

Thus, at the end of the first iteration of our strategy we get:

$T = (P_{\mathcal{K},\varphi} - \{3\}) \cup \{13, 14, 15, 16, 17\} \cup \{3.6, 3.7, 3.8\}$
$Defs = \{3\} \cup \{13, 14, 15, 16, 17\}$
$InDefs = \{13, 14, 15, 16, 17\}$

Since $InDefs \neq \emptyset$ we perform a second execution of the loop body of our strategy. After a few more executions, and a final application of the definition elimination rule, we get the following linear monadic ω-program T:

$prop \leftarrow \neg\, p_1$	$p_3([s_2|P]) \leftarrow p_8(P)$	$p_6([s_1|P]) \leftarrow p_6(P)$
$p_1 \leftarrow p_2(P)$	$p_3([s_1|P]) \leftarrow p_6(P)$	$p_6([s_2|P]) \leftarrow$
$p_2([s_1|P]) \leftarrow \neg\, p_3(P)$	$p_3([s_2|P]) \leftarrow$	$p_6([s_2|P]) \leftarrow p_6(P)$
$p_2([s_1|P]) \leftarrow p_4(P)$	$p_3([s_2|P]) \leftarrow p_6(P)$	$p_7([s_1|P]) \leftarrow$
$p_2([s_2|P]) \leftarrow p_5(P)$	$p_4([s_2|P]) \leftarrow p_5(P)$	$p_8([s_1|P]) \leftarrow p_7(P)$
$p_3([s_1|P]) \leftarrow$	$p_5([s_1|P]) \leftarrow \neg\, p_3(P)$	$p_8([s_1|P]) \leftarrow p_8(P)$
$p_3([s_1|P]) \leftarrow p_7(P)$	$p_5([s_1|P]) \leftarrow p_4(P)$	$p_8([s_2|P]) \leftarrow p_8(P)$
$p_3([s_1|P]) \leftarrow p_8(P)$	$p_5([s_2|P]) \leftarrow p_5(P)$	

3.3 A Proof System for Linear Monadic ω-Programs.

Now we present a proof system for checking whether a quantified literal is true or not in the perfect model of a linear monadic ω-program.

A *closed literal* is a statement of one of the following forms: p, $\neg p$, $\forall\,(L)$, $\exists\,(L)$, where p is a nullary predicate and L is either an atom $q(X)$ or a negated atom $\neg q(X)$, and $\forall(L)$ and $\exists(L)$ denote the universal and existential closure of L, respectively. The proof rules in Figure 1 define a provability relation $\vdash$, such that for every linear monadic ω-program P and closed literal L, $P \vdash L$ iff $M(P) \models L$ (see Theorem 3 below). When applying these proof rules we assume that: (i) the set Σ of states is $\{s_1, \ldots, s_h\}$, (ii) each clause in P of the form $H\leftarrow$ is written as $H \leftarrow true$ (so that no clause in P has an empty body) and (iii) the formulas $\exists(true)$ and $\forall(true)$, which may appear in the premise of rules S5 and S6, are identified with $true$.

Note that the proof rules S7, S8, and S9 have negative premises. The negated judgement '$P \nvdash L$' should be interpreted as '$P \vdash L$ cannot be proved by using the proof rules S1–S9' and in this case we say that '$P \vdash L$ has a disproof'.

S1. $\dfrac{}{P \vdash \mathit{true}}$ S2. $\dfrac{}{P \vdash p}$ if $p \leftarrow \mathit{true} \in P$

S3. $\dfrac{P \vdash L}{P \vdash p}$ if $p \leftarrow L \in P$ and $\mathit{vars}(L) = \emptyset$ S4. $\dfrac{P \vdash \exists(L)}{P \vdash p}$ if $p \leftarrow L \in P$ and $\mathit{vars}(L) \neq \emptyset$

S5. $\dfrac{P \vdash \exists(L)}{P \vdash \exists(p(Z))}$ if $p([s|Y]) \leftarrow L \in P$ for some $s \in \{s_1, \ldots, s_h\}$

S6. $\dfrac{P \vdash \forall(L_1) \quad \ldots \quad P \vdash \forall(L_h)}{P \vdash \forall(p(Z))}$ if $\{ p([s_1|Y]) \leftarrow L_1, \ldots, p([s_h|Y]) \leftarrow L_h \} \subseteq P$

S7. $\dfrac{P \nvdash p}{P \vdash \neg p}$ S8. $\dfrac{P \nvdash \forall(p(Z))}{P \vdash \exists(\neg p(Z))}$ S9. $\dfrac{P \nvdash \exists(p(Z))}{P \vdash \forall(\neg p(Z))}$

Fig. 1. Proof system for linear monadic ω-programs. $\Sigma = \{s_1, \ldots, s_h\}$ is the set of states of the Kripke structure $\mathcal{K}$.

The interpretation of $P \nvdash L$ as (finite or infinite) failure of $P \vdash L$ is meaningful because the program P is stratified and, thus, also the instances of the proof rules can be stratified. The stratification of these instances is induced by a well-founded ordering on closed literals such that, for every rule instance with conclusion $P \vdash L_1$, (i) if $P \vdash L_2$ occurs as a premise, then L_2 is not larger than L_1, and (ii) if $P \nvdash L_2$ occurs as a premise, then L_2 is strictly smaller than L_1. Thus, in order to construct a proof for $P \vdash L$, we are never required to show that $P \nvdash L$, that is, we are never required to show that no proof for $P \vdash L$ itself can be constructed.

The following theorem shows that the proof rules of Figure 1 are a sound and complete proof system for proving that a closed literal is true in the perfect model of a linear monadic ω-program.

Theorem 3. *For every linear monadic ω-program P and closed literal L, $P \vdash L$ iff $M(P) \models L$.*

The proof system for linear monadic ω-programs can be encoded as a logic program, which we call *Demo*. In *Demo* a closed literal L is represented by a ground term $\lceil L \rceil$ constructed as follows. Let v be a new constant symbol. (i) For any variable Z, $\lceil Z \rceil$ is v, (ii) for any list $[s|Z]$, where s is a state and Z is a variable, $\lceil [s|Z] \rceil$ is (s, v), (iii) for any nullary predicate p, $\lceil p \rceil$ is p, (iv) for any unary predicate q and term t, $\lceil q(t) \rceil$ is $(q, \lceil t \rceil)$, (v) for any atom A, $\lceil \neg A \rceil$ is $\mathit{not}(\lceil A \rceil)$, (vi) for any literal L, $\lceil \exists(L) \rceil$ is $e(\lceil L \rceil)$ and $\lceil \forall(L) \rceil$ is $a(\lceil L \rceil)$. An ω-program P is represented by the set $\lceil P \rceil$ of ground unit clauses of the form $\mathit{clause}(\lceil H \rceil, \lceil B \rceil) \leftarrow$ such that $H \leftarrow B$ is a clause of P.

In the clauses for *Demo*, which we list below, we also use the predicate $\mathit{nullary}(R)$ which holds iff R is a nullary predicate symbol, and $\mathit{emptyvars}(L)$ which holds iff L has no occurrences of the symbol v. The three clauses 1.1, 1.2, and 1.3 correspond to rule S1 (recall that in the proof system of Figure 1 we identify the three expressions *true*, $\exists(\mathit{true})$, and $\forall(\mathit{true})$).

1.1 $demo(true) \leftarrow$
1.2 $demo(e(true)) \leftarrow$
1.3 $demo(a(true)) \leftarrow$
2. $demo(R) \leftarrow nullary(R) \wedge clause(R, true)$
3. $demo(R) \leftarrow nullary(R) \wedge clause(R, L) \wedge emptyvars(L) \wedge demo(L)$
4. $demo(R) \leftarrow nullary(R) \wedge clause(R, L) \wedge \neg emptyvars(L) \wedge demo(e(L))$
5. $demo(e((R, v))) \leftarrow clause((R, (S, v)), L) \wedge demo(e(L))$
6. $demo(a((R, v))) \leftarrow clause((R, (s_1, v)), L_1) \wedge demo(a(L_1)) \wedge \ldots \wedge clause((R, (s_h, v)), L_h) \wedge demo(a(L_h))$
7. $demo(not(R)) \leftarrow nullary(R) \wedge \neg demo(R)$
8. $demo(e(not((R, v)))) \leftarrow \neg\, demo(a((R, v)))$
9. $demo(a(not((R, v)))) \leftarrow \neg\, demo(e((R, v)))$

Since P is a stratified program, $Demo \cup \lceil P \rceil$ is a *weakly stratified* program [14] and, hence, it has a unique perfect model $M(Demo \cup \lceil P \rceil)$.

Theorem 4. *For every linear monadic ω-program P and closed literal L, $P \vdash L$ iff $M(Demo \cup \lceil P \rceil) \models demo(\lceil L \rceil)$.*

Thus, by Theorems 3 and 4 for any linear monadic ω-program P, we can check whether or not $M(P) \models L$ holds by using any logic programming system which computes the perfect model of $Demo \cup \lceil P \rceil$. One can use, for instance, a system based on *tabled resolution* [3,19] which guarantees the termination of any query of the form $demo(\lceil L \rceil)$ and returns 'yes' iff $demo(\lceil L \rceil)$ belongs to $M(Demo \cup \lceil P \rceil)$. Indeed, starting from $demo(\lceil L \rceil)$, we can only derive a finite set of queries of the form $demo(\lceil M \rceil)$, and the tabling mechanism ensures that each query is evaluated at most once.

Example 4. Let T be the linear monadic ω-program that is the output of our transformation strategy, as illustrated in Example 3. By using our proof system we obtain the proof in Figure 2 which shows that $T \vdash prop$. Thus, $M(P_{\mathcal{K},\varphi}) \models prop$ and, therefore, $\mathsf{G}(a \rightarrow \mathsf{F}\, a)$ holds in $\mathcal{K}$ (see Example 2).

3.4 Complexity of the Verification Technique

The complexity of our verification technique will be measured in terms of: (i) the number of applications of transformation rules for generating the linear monadic ω-program T from $P_{\mathcal{K},\varphi}$ (Step 1), and (ii) the number of closed literals that are checked by the proof system during the execution of the *Demo* program (Step 2).

Let us consider Step 1. In the body of the clauses defining each new predicate introduced by the *define-fold* procedure, there is at most one occurrence of an atom in $\{q_1(X), \ldots, q_m(X)\}$ and all other occurrences of atoms are taken from the set $\{path(X), nopath(X)\} \cup \{sat(X, \psi) \mid \psi \text{ is a subformula of } \varphi\}$. Thus, the number of new predicate symbols that can be introduced is $\mathcal{O}((m + 1) \cdot 2^{|\varphi|})$. The number of clauses introduced for each new predicate symbol is $\mathcal{O}(|\varphi|)$.

For each new clause, our transformation strategy performs one execution of the loop body, which starts off by applying the *instantiate* procedure and

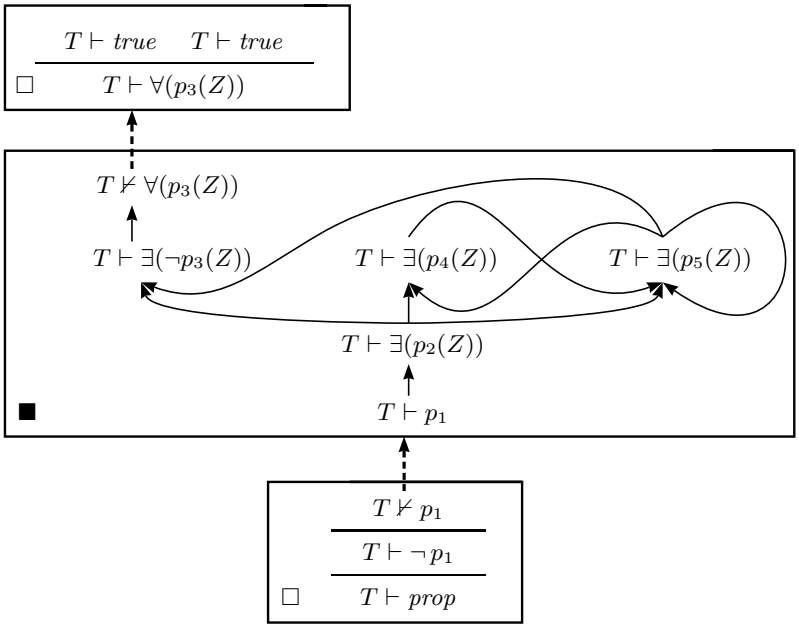

Fig. 2. Proof of $T \vdash prop$ (see Example 4). A rectangle marked by □ (or ■) shows the proof (or a disproof, respectively) of its root judgement at the bottom of the rectangle. In the rectangle marked by ■ there is a solid arrow from judgement A to judgement B iff there exists an instance of a proof rule with conclusion A and premise B. A dashed uparrow from a negated judgement $T \nvdash \varphi$ to a judgement $T \vdash \varphi$ indicates that in order to show that $T \nvdash \varphi$ holds (or does not hold), we provide a disproof (or a proof, respectively) of $T \vdash \varphi$.

generates $\mathcal{O}(|\Sigma|)$ clauses. Then, it continues by applying the *unfold* procedure. The number of possible unfolding steps for each instantiated clause is $\mathcal{O}(|\varphi|)$. Thus, the total number of unfolding steps for each new clause is $\mathcal{O}(|\Sigma| \cdot |\varphi|)$, which is also the number of clauses generated by the instantiation and unfolding procedures. Finally, the *define-fold* procedure performs at most one folding step and definition step per clause. Considering all possible predicate symbols and recalling that the number of clauses introduced for each new predicate is $\mathcal{O}(|\varphi|)$, we get that the total number of transformation rule applications during Step 1 is $\mathcal{O}(|\Sigma| \cdot (m+1) \cdot 2^{|\varphi| + 2log_2|\varphi|})$.

In Step 2, by using tabled resolution, the proof system checks every closed literal at most once. The proof of a closed literal requires $\mathcal{O}((m+1) \cdot 2^{|\varphi|})$ applications of proof rules. Therefore, we may conclude that the complexity of our algorithm is $\mathcal{O}(|\Sigma| \cdot (m+1) \cdot 2^{|\varphi| + 2log_2|\varphi|})$.

The complexity of the Lichtenstein-Pnueli algorithm is $\mathcal{O}((|\Sigma|+|\rho|) \cdot 2^{5|\varphi|})$ [9]. Since ρ is a total binary relation on Σ, we have that $|\Sigma| \leq |\rho| \leq |\Sigma|^2$. In the case where $|\rho| = |\Sigma|^2$ and, thus, $m = 0$, the complexity of the Lichtenstein-Pnueli algorithm is $\mathcal{O}(|\Sigma|^2 \cdot 2^{5|\varphi|})$ and the complexity of our algorithm is $\mathcal{O}(|\Sigma| \cdot 2^{|\varphi| + 2log_2|\varphi|})$. In the case where ρ is a function, we have that $|\rho| = m = |\Sigma|$. Thus, the complexity of the Lichtenstein-Pnueli algorithm is $\mathcal{O}(|\Sigma| \cdot 2^{5|\varphi|})$ and the complexity of our algorithm is $\mathcal{O}(|\Sigma|^2 \cdot 2^{|\varphi| + 2log_2|\varphi|})$. When $m = |\Sigma|$, it

may still be the case that $|\rho|$ is proportional to $|\Sigma|^2$ and, in this case, the Lichtenstein-Pnueli algorithm and our algorithm are both quadratic in the size of Σ. In the literature, $\mathcal{O}(2^{5|\varphi|})$ is often overestimated to $2^{\mathcal{O}(|\varphi|)}$ and, therefore, the Lichtenstein-Pnueli algorithm and our algorithm have essentially the same time complexity, that is, $\mathcal{O}(|\Sigma|^2) \cdot 2^{\mathcal{O}(|\varphi|)}$.

4 Related Work and Concluding Remarks

Various logic programming techniques and tools have been developed for model checking. For instance, tabled resolution has been shown to be quite effective for implementing a modal μ-calculus model checker for a CCS value passing language [16]. Techniques based on constraint logic programming, abstract interpretation, and program transformation have been proposed for performing CTL model checking of finite and infinite state systems (see, for instance, [6,7,12]).

The main novelties of this paper are the following: (i) we have proposed a method for specifying LTL properties of reactive systems based on ω-programs, that is, logic programs acting on infinite lists, (ii) we have also introduced the subclass of linear monadic ω-programs for which the truth in the perfect model is decidable and, finally, (iii) we have shown that we can transform, by applying semantics preserving unfold/fold rules, the logic programming specification of an LTL property into a linear monadic ω-program.

Our two step verification approach bears some similarity to the automata-theoretic approaches for LTL model checking, where the specification of a finite state system and an LTL formula are translated into nondeterministic Büchi automata [21] or alternating automata [11].

The automata-theoretic approach has the advantage that automata theory is very well studied and many results are available. However, we believe that also our approach has its advantages because of the following features. (1) The specification of the properties of the reactive systems, the transformation of the specification into a linear monadic ω-program, and the proof of the properties of a linear monadic ω-program can all be done within the single framework of logic programming, while in the automata-theoretic approach one has to use both the temporal logic formalism and the automata-theoretic formalism. (2) The translation of the specification into an ω-program can be performed by using semantics preserving transformation rules, thereby avoiding the burden of proving the correctness of the translation via *ad-hoc* methods.

Issues which can be investigated in future research include: (i) the relationships between linear monadic ω-programs, Büchi automata, and alternating automata, (ii) the strength of our transformational approach and its applicability to other logics, such as CTL* and the Monadic Second Order logic of successors, and (iii) the comparison of the efficiency of our approach w.r.t. that of other model checking techniques via experiments using practical examples.

References

1. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.

2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
3. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.
4. E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.
5. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.
6. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'01*, Technical Report DSSE-TR-2001-3, pages 85–96. Univ. Southampton, UK, 2001.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In *Program Development in Computational Logic*, LNCS 3049, pp. 292–340. Springer-Verlag, 2004.
9. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. *Proc. of POPL'85*, pp.97–107. ACM Press, 1985.
10. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
11. D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings of LICS '88*, pages 422–427. IEEE Press, 1988.
12. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. *Proc. of CL 2000*, LNAI 1861, pp. 384–398. Springer, 2000.
13. M. Proietti and A. Pettorossi. Unfolding-definition-folding, in this order, for avoiding unnecessary variables in logic programs. *Theoretical Computer Science*, 142(1):89–124, 1995.
14. H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.
15. T. C. Przymusinski. On the declarative semantics of stratified deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.
16. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. *Proceedings of CAV '97*, LNCS 1254, pages 143–154. Springer-Verlag, 1997.
17. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I.V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. *International Journal on Foundations of Computer Science*, 13(3):387–403, 2002.
18. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
19. K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, and E. Johnson. The XSB System, Version 2.2., 2000.
20. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proceedings of ICLP'84*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.
21. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). *Proc. LICS '86*, pp. 332–344. IEEE Press, 1986.

A Linear Temporal Logic

In this section we briefly recall the definition of the Linear Temporal Logic. We first introduce the notion of a Kripke structure, which models state transition systems, and then we define the semantics of LTL formulas with respect to Kripke structures. We refer the reader to [5] for further details.

Definition 3 (Kripke Structure). Let *Elem* be a set of symbols denoting *elementary properties*. A *Kripke Structure* $\mathcal{K}$ is a 4-tuple $\langle \Sigma, I, \rho, \lambda \rangle$ where:

1. Σ is a finite set of *states*;
2. $I \subseteq \Sigma$ is a set of *initial states*;
3. $\rho \subseteq \Sigma \times \Sigma$ is a total *transition relation*, that is, for every state $s \in \Sigma$ there exists a state $s' \in \Sigma$ such that $(s, s') \in \rho$;
4. $\lambda : \Sigma \to \mathcal{P}(Elem)$ is a total function that assigns to a state $s \in \Sigma$ a subset $\lambda(s)$ of *Elem* which is the subset of the elementary properties that hold in s.

A *computation path* π in $\mathcal{K}$ is an *infinite* list $[s_0, s_1, \ldots]$ of states such that $s_0 \in I$ and, for every $i \geq 0$, $(s_i, s_{i+1}) \in \rho$. We use π_i to denote the sequence obtained by taking the suffix of π which starts at state s_i.

LTL is a propositional temporal logic for expressing properties of the computation paths of a Kripke structure. LTL makes use of the temporal operators X (*next time*) and U (*until*). Other temporal operators, such as F (*eventually*) and G (*always*), can be defined in terms of X and U as follows: for every formula φ, $\mathsf{G}\varphi = \neg\, \mathsf{F} \neg\, \varphi$, and $\mathsf{F}\varphi = \mathsf{U}(\mathit{true}, \varphi)$.

Definition 4 (LTL Formulas). Given a set *Elem* of elementary properties, the syntax of the LTL formulas φ is as follows:

$$\varphi ::= e \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathsf{X}\varphi \mid \varphi_1 \mathsf{U}\, \varphi_2, \quad \text{where } e \in \mathit{Elem}.$$

Now we define the satisfaction relation $\mathcal{K}, \pi \vDash \varphi$, which tells us when an LTL formula φ holds on a computation path π of the Kripke structure $\mathcal{K}$. We assume that the structure $\mathcal{K}$ and the formula φ share the same set *Elem* of elementary properties.

Definition 5 (Satisfaction Relation for LTL). Given a Kripke structure $\mathcal{K} = \langle \Sigma, I, \rho, \lambda \rangle$, a computation path π of $\mathcal{K}$, and an LTL formula φ, we inductively define the relation $\mathcal{K}, \pi \vDash \varphi$ as follows:

$$\begin{array}{lll}
\mathcal{K}, \pi \vDash e & \text{iff} & \pi = s_0 s_1 \ldots \text{ and } e \in \lambda(s_0) \\
\mathcal{K}, \pi \vDash \neg\varphi & \text{iff} & \mathcal{K}, \pi \nvDash \varphi \\
\mathcal{K}, \pi \vDash \varphi_1 \wedge \varphi_2 & \text{iff} & \mathcal{K}, \pi \vDash \varphi_1 \text{ and } \mathcal{K}, \pi \vDash \varphi_2 \\
\mathcal{K}, \pi \vDash \mathsf{X}\, \varphi & \text{iff} & \mathcal{K}, \pi_1 \vDash \varphi \\
\mathcal{K}, \pi \vDash \varphi_1\, \mathsf{U}\, \varphi_2 & \text{iff} & \text{there exists } i \geq 0 \text{ such that } \mathcal{K}, \pi_i \vDash \varphi_2 \\
 & & \text{and, for all } 0 \leq j < i,\ \mathcal{K}, \pi_j \vDash \varphi_1
\end{array}$$

$\mathcal{K} \vDash \varphi$ iff for all computation paths π in $\mathcal{K}$ we have that $\mathcal{K}, \pi \vDash \varphi$.