Deciding Full Branching Time Logic by Program Transformation

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 method based on logic program transformation, for verifying Computation Tree Logic (CTL*) properties of finite state reactive systems. The finite state systems and the CTL* properties we want to verify, are encoded as logic programs on infinite lists. Our verification method consists of two steps. In the first step we transform the logic program that encodes the given system and the given property, into a *monadic ω-program*, that is, a stratified program defining nullary or unary predicates on infinite lists. This transformation is performed by applying unfold/fold rules that preserve the perfect model of the initial program. In the second step we verify the property of interest by using a proof method for monadic ω-programs.

1 Introduction

The branching time temporal logic CTL* is among the most popular temporal logics that have been proposed for verifying properties of reactive systems [4]. A finite state reactive system, such as a protocol, a concurrent system, or a digital circuit, is formally specified as a Kripke structure and the property to be verified is specified as a CTL* formula. Thus, the problem of checking whether or not a reactive system satisfies a given property is reduced to the problem of checking whether or not a Kripke structure is a model of a CTL* formula.

There is a vast literature on the problem of model checking for the CTL* logic and, in particular, its two fragments: (i) the Computational Tree Logic CTL, and (ii) the Linear-time Temporal Logic LTL (see [2] for a survey). Most of the known model checking algorithms for CTL* either combine model checking algorithms for CTL and LTL [2], or use techniques based on translations to automata on infinite trees [6].

In this paper we extend to CTL* a method proposed in [11] for LTL. We encode the satisfaction relation of a CTL* formula φ with respect to a Kripke structure $\mathcal{K}$ by means of a locally stratified logic program $P_{\mathcal{K},\varphi}$. The program $P_{\mathcal{K},\varphi}$ belongs to a class of programs, called ω-programs, which define predicates on infinite lists. Predicates of this type are needed because the definition of the satisfaction relation is based on the infinite computation paths of $\mathcal{K}$. The semantics of $P_{\mathcal{K},\varphi}$ is provided by its unique *perfect model* [12] which for ω-programs 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 *monadic* ω-program, that is, a stratified program that defines nullary or unary predicates on infinite lists. This transformation is performed by applying unfold/fold transformation rules similar to those presented in [5,14,15] according to a strategy which is a variant of the *specialization strategy* presented in [5]. Similarly to [5,14], the use of those unfold/fold rules guarantees the preservation of the perfect model of $P_{\mathcal{K},\varphi}$.

In the second step of our verification method we apply a proof method for monadic ω-programs which is sound and complete with respect to the perfect model semantics.

The paper is structured as follows. In Section 2 we introduce the class of ω-programs and we show how to encode the satisfaction relation for any given Kripke structure and CTL* formula as an ω-program. In Section 3 we present our verification method. In particular, in Section 3.1 we present the specialization strategy for transforming an ω-program into a monadic ω-program and in Section 3.2 we present the proof method for monadic ω-programs. Finally, in Section 4 we discuss related work in the area of model checking and logic programming.

2 Encoding CTL* Model Checking as a Logic Program

In this section we describe a method which, given a Kripke structure $\mathcal{K}$ and a CTL* *state formula* φ, allows us to construct a logic program $P_{\mathcal{K},\varphi}$ and to define a nullary 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 $\mathcal{K} \models \varphi$ holds, also called the problem of model checking φ with respect to $\mathcal{K}$, is reduced to the problem of testing whether or not $M(P_{\mathcal{K},\varphi}) \models \mathit{prop}$ holds.

Now we briefly recall the definition of the temporal logic CTL* (see [2] for more details). A Kripke structure is a 4-tuple $\langle \Sigma, s_0, \rho, \lambda \rangle$, where: (i) $\Sigma = \{s_0, \ldots, s_h\}$ is a finite set of *states*, (ii) $s_0 \in \Sigma$ is the *initial state*, (iii) $\rho \subseteq \Sigma \times \Sigma$ is a total *transition relation*, and (iv) $\lambda : \Sigma \to \mathcal{P}(\mathit{Elem})$ is a total function that assigns to every state $s \in \Sigma$ a subset $\lambda(s)$ of the set *Elem* of *elementary properties*. A *computation path* of $\mathcal{K}$ from a state s is an infinite list $[a_0, a_1, \ldots]$ of states such that $a_0 = s$ and, for every $i \geq 0$, $(a_i, a_{i+1}) \in \rho$. Given an infinite list $\pi = [a_0, a_1, \ldots]$ of states, by π_j, for any $j \geq 0$, we denote the infinite list which is the suffix $[a_j, a_{j+1}, \ldots]$ of π.

Definition 1 (CTL* Formulas). Given a set *Elem* of elementary properties, a CTL* formula φ is either a *path formula* φ_p or a *state formula* φ_s defined as follows:

(*path formulas*) $\quad \varphi_p ::= \varphi_s \mid \neg\varphi_p \mid \varphi_p \wedge \varphi_p \mid \mathsf{X}\, \varphi_p \mid \varphi_p\, \mathsf{U}\, \varphi_p$
(*state formulas*) $\quad \varphi_s ::= d \mid \neg\varphi_s \mid \varphi_s \wedge \varphi_s \mid \mathsf{E}\, \varphi_p$

where $d \in \mathit{Elem}$.

As the following definition formally specifies, (i) $\mathsf{X}\,\varphi$ holds on a computation path π if φ holds in the second state of π, (ii) $\varphi_1\,\mathsf{U}\,\varphi_2$ holds on a computation path π if φ_2 holds in a state s of π and φ_1 holds in every state preceding s in π, and (iii) $\mathsf{E}\,\varphi$ holds in a state s if there exists a computation path starting from s on which φ holds.

Definition 2 (Satisfaction Relation for CTL*). Let $\mathcal{K} = \langle \Sigma, s_0, \rho, \lambda \rangle$ be a Kripke structure. For any CTL* formula φ and infinite list $\pi \in \Sigma^\omega$, the relation $\mathcal{K}, \pi \vDash \varphi$ is inductively defined as follows:

$\mathcal{K}, \pi \vDash d$ iff $\pi = [a_0, a_1, \ldots]$ and $d \in \lambda(a_0)$
$\mathcal{K}, \pi \vDash \neg\,\varphi$ iff $\mathcal{K}, \pi \nvDash \varphi$
$\mathcal{K}, \pi \vDash \varphi_1 \wedge \varphi_2$ iff $\mathcal{K}, \pi \vDash \varphi_1$ and $\mathcal{K}, \pi \vDash \varphi_2$
$\mathcal{K}, \pi \vDash \mathsf{X}\,\varphi$ iff $\mathcal{K}, \pi_1 \vDash \varphi$
$\mathcal{K}, \pi \vDash \varphi_1\,\mathsf{U}\,\varphi_2$ iff there exists $i \geq 0$ such that $\mathcal{K}, \pi_i \vDash \varphi_2$ and, for all $0 \leq j < i$, $\mathcal{K}, \pi_j \vDash \varphi_1$
$\mathcal{K}, \pi \vDash \mathsf{E}\,\varphi$ iff $\pi = [a_0, a_1, \ldots]$ and there exists a computation path π' from a_0 such that $\mathcal{K}, \pi' \vDash \varphi$.

Given a *state formula* φ, we say that $\mathcal{K}$ is a *model* of φ, written $\mathcal{K} \vDash \varphi$, iff there exists an infinite list $\pi \in \Sigma^\omega$ such that the first state of π is the initial state s_0 of $\mathcal{K}$ and $\mathcal{K}, \pi \vDash \varphi$ holds.

The above definition of the satisfaction relation for CTL* formulas is a shorter, yet equivalent, version of the usual definition one can find in the literature [2].

In order to encode the satisfaction relation for CTL* formulas as a logic program, we will introduce in the next section a class of logic programs, called *ω-programs*. In this class the arguments of predicates may denote infinite lists.

2.1 Syntax and Semantics of ω-Programs

Let us consider a Kripke structure $\mathcal{K}$. Let us also 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 the states of $\mathcal{K}$, each state being a constant of $\mathcal{L}_\omega$, (ii) the set *Elem* of the elementary properties of $\mathcal{K}$, each elementary property being a constant of $\mathcal{L}_\omega$, and (iii) the binary function symbol $[\_|\_]$ which is the constructor of infinite lists. Thus, for instance, $[H|T]$ is the infinite list whose head is H and whose tail is the infinite list T.

We assume that $\mathcal{L}_\omega$ is a typed language [9] with the following 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 $\mathit{Fun} - (\Sigma \cup \{[\_|\_]\})$, 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 arity 0 and 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 $\tau_1, \ldots, \tau_n \in \{\texttt{fterm}, \texttt{state}, \texttt{ilist}\}$. An *ω-program* is a logic program constructed as usual (see, for instance, [9]) from symbols in the

typed language $\mathcal{L}_\omega$. In what follows, for reasons of simplicity, we will feel free to say 'program', instead of 'ω-program'.

Given a term or a formula t, by $vars(t)$ we denote the set of variables occurring in t. The same notation will be used for sets of terms and sets of formulas. The *existential closure* of a formula φ, denoted $\exists(\varphi)$, is the formula $\exists X_1 \ldots \exists X_n\, \varphi$ where $\{X_1, \ldots, X_n\}$ is the set of the *free variables* occurring in φ. The *universal closure* of a formula φ, denoted $\forall(\varphi)$, is defined in a similar way by using $\forall$, instead of $\exists$. Note that if $vars(\varphi) = \emptyset$, then $\exists(\varphi)$ is φ itself.

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 the infinite lists of states. An ω-interpretation I is an interpretation such that: (i) I assigns to the types `fterm`, `state`, and `ilist`, respectively, the sets HU, Σ, and Σ^ω, (ii) I assigns to the function symbol $[\_|\_]$ the function $[\_|\_]_I$ such that, for any state $a \in \Sigma$ and infinite list $[a_1, a_2, \ldots] \in \Sigma^\omega$, $[a|[a_1, a_2, \ldots]]_I$ is the infinite list $[a, a_1, a_2, \ldots]$, (iii) I is an Herbrand interpretation for all function symbols in $Fun - (\Sigma \cup \{[\_|\_]\})$, and (iv) I assigns to every n-ary predicate $p \in Pred$ of type $\tau_1 \times \ldots \times \tau_n$ 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, according to the case where 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 $I \models \forall(\gamma)$.

A *level mapping* is a function $\ell : Pred \to \mathbb{N}$. A level mapping is extended to literals as follows: for any literal L having predicate p, if L is a positive literal, then $\ell(L) = \ell(p)$ and, if L is a negative literal then $\ell(L) = \ell(p) + 1$. An ω-clause γ of the form $H \leftarrow L_1 \wedge \ldots \wedge L_m$ is *stratified* w.r.t. ℓ if, for $i = 1, \ldots, m$, $\ell(H) \geq \ell(L_i)$. An ω-program P is *stratified* if there exists a level mapping ℓ such that all clauses of P are stratified w.r.t. ℓ.

A *valuation* is a function $v : Var \to HU \cup \Sigma \cup \Sigma^\omega$ such that: (i) if X has type `fterm` then $v(X) \in HU$, (ii) if X has type `state` then $v(X) \in \Sigma$, and (iii) if X has type `ilist` then $v(X) \in \Sigma^\omega$. For any term t, literal L, and clause γ, we define $v(t)$, $v(L)$, and $v(\gamma)$, by induction on the structure of t, L, and γ, respectively. We will say that $v(t)$, $v(L)$, and $v(\gamma)$, is 'a term', 'a literal', and 'a clause', respectively, also when they are infinite structures.

We extend the notion of *Herbrand base* [9] to the case of ω-programs by introducing the set $\mathcal{B}_\omega$ defined as follows:

$$\mathcal{B}_\omega = \{p(v(X_1), \ldots, v(X_n)) \mid p \text{ is an } n\text{-ary predicate symbol and } v \text{ is a valuation}\}$$

Thus, any ω-interpretation can be identified with a subset of $\mathcal{B}_\omega$.

A *local stratification* is a function $\sigma : \mathcal{B}_\omega \to W$, where W is the set of countable ordinals. Given $A \in \mathcal{B}_\omega$, we define $\sigma(\neg A) = \sigma(A) + 1$. Given a clause γ of the form $H \leftarrow L_1 \wedge \ldots \wedge L_m$ in an ω-program P and a local stratification σ, we say that γ is *locally stratified* w.r.t. σ if for $i = 1, \ldots, m$, for every valuation v, $\sigma(v(H)) \geq \sigma(v(L_i))$. An ω-program P is *locally stratified w.r.t. σ*, or σ is a *local stratification for P*, if every clause in P is locally stratified w.r.t. σ. An

ω-program P is *locally stratified* if there exists a local stratification σ such that P is *locally stratified w.r.t.* σ.

Clearly, every stratified ω-program is a locally stratified ω-program. Similarly to the case of logic programs, for every locally stratified ω-program P (and, hence, for every stratified ω-program P), we can construct a unique *perfect* ω-*model* (or *perfect model*, for short) denoted by $M(P)$ [1,12] (an instance of this construction is presented in Example 1).

Definition 3 (Monadic ω-Programs). A *monadic* ω-clause is an ω-clause of the form $A_0 \leftarrow L_1 \wedge \ldots \wedge L_m$, with $m \geq 0$, such that: (i) A_0 is an atom of the form p_0 or $q_0([s|X_0])$, where q_0 is a predicate of type `ilist` and $s \in \Sigma$, (ii) for $i = 1, \ldots, m$, L_i is either an atom A_i or a negated atom $\neg A_i$, where A_i is of the form p_i or $q_i(X_i)$, and q_i is a predicate of type `ilist`, and (iii) there exists a level mapping ℓ such that, for $i = 1, \ldots, m$, if L_i is an atom and $vars(A_0) \not\supseteq vars(L_i)$, then $\ell(A_0) > \ell(L_i)$ else $\ell(A_0) \geq \ell(L_i)$. A *monadic* ω*-program* is a finite set of monadic ω-clauses.

Note that in Definition 3 the predicate symbols $p_0, q_0, \ldots, p_m, q_m$ and the variables $X_0, \ldots, X_m$ are *not* necessarily distinct. Condition (iii) ensures that a monadic ω-program is stratified. This condition, which is actually stronger than stratification, is also needed for guaranteeing the completeness of the proof method for monadic ω-programs (see Section 3.2).

Example 1. Let r, q, and p be predicates of type `ilist`. The following set of clauses is a monadic ω-program P (and, thus, also an ω-program):

$$
\begin{array}{lll}
p([a|X]) \leftarrow p(X) & q([a|X]) \leftarrow q(X) & r([a|X]) \leftarrow r(X) \\
p([b|X]) \leftarrow \neg\, q(X) & q([a|X]) \leftarrow \neg\, r(X) & r([b|X]) \leftarrow \\
 & q([b|X]) \leftarrow q(X) &
\end{array}
$$

Program P is stratified by the level mapping $\ell : Pred \rightarrow \mathbb{N}$ such that $\ell(p) = 2$, $\ell(q) = 1$, and $\ell(r) = 0$. The perfect model $M(P)$ is constructed starting from the ground atoms of level 0 and going up, level-by-level, as we now indicate. We start from the ground atoms of level 0, that is, the ground atoms with predicate r. For all $w \in \{a, b\}^\omega$, $r(w) \in M(P)$ iff $w \in a^*b(a + b)^\omega$. Thus, $r(w) \notin M(P)$ iff $w \in a^\omega$, that is, $\neg\, r(w)$ holds in $M(P)$ iff $w \in a^\omega$. Then we consider the ground atoms of level 1, that is, the ground atoms with predicate q. For all $w \in \{a, b\}^\omega$, $q(w) \in M(P)$ iff $w \in (a+b)^*a^\omega$ (that is, w has finitely many occurrences of b). Thus, $\neg\, q(w)$ holds in $M(P)$ iff $w \in (a^*b)^\omega$ (that is, w has infinitely many occurrences of b). Finally, we consider the ground atoms of level 2, that is, the ground atoms with predicate p. For all $w \in \{a, b\}^\omega$, $p(w) \in M(P)$ iff $w \in (a^*b)(a^*b)^\omega$, that is, $p(w) \in M(P)$ iff $w \in (a^*b)^\omega$.

2.2 Encoding the CTL* Satisfaction Relation as an ω-Program

Given a Kripke structure $\mathcal{K}$ and a CTL* state 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*(π) holds iff π is an infinite list

representing a computation path of $\mathcal{K}$, (ii) the binary predicate *sat* that encodes the satisfaction relation for CTL* formulas, in the sense that for all computation paths π and CTL* formulas ψ, we have that $M(P_{\mathcal{K},\varphi}) \vDash sat(\pi, \psi)$ iff $\mathcal{K}, \pi \vDash \psi$, and (iii) the nullary predicate *prop* that encodes the property φ to be verified, in the sense that *prop* holds iff there exists an infinite list π whose first element is the initial state s_0 of $\mathcal{K}$ and $\mathcal{K}, \pi \vDash \varphi$.

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

Definition 4 (Encoding Program). Given a Kripke structure $\mathcal{K} = \langle \Sigma, s_0, \rho, \lambda \rangle$ and a CTL* formula φ, the *encoding program* $P_{\mathcal{K},\varphi}$ is the following ω-program:

1. $prop \leftarrow sat([s_0|X], \varphi)$
2. $sat([S|X], F) \leftarrow elem(F, S)$
3. $sat(X, not(F)) \leftarrow \neg\, sat(X, F)$
4. $sat(X, and(F_1, F_2)) \leftarrow sat(X, F_1) \wedge sat(X, F_2)$
5. $sat([S|X], x(F)) \leftarrow sat(X, F)$
6. $sat(X, u(F_1, F_2)) \leftarrow sat(X, F_2)$
7. $sat([S|X], u(F_1, F_2)) \leftarrow sat([S|X], F_1) \wedge sat(X, u(F_1, F_2))$
8. $sat([S|X], e(F)) \leftarrow exists\_sat(S, F)$
9. $exists\_sat(S, F) \leftarrow path([S|Y]) \wedge sat([S|Y], F)$
10. $path(X) \leftarrow \neg\, notpath(X)$
11. $notpath([S_1, S_2|X]) \leftarrow \neg\, tr(S_1, S_2)$
12. $notpath([S|X]) \leftarrow notpath(X)$

together with the clauses defining the predicates *tr* and *elem*, where:
(1) for all states $s_1, s_2 \in \Sigma$, $tr(s_1, s_2)$ holds iff $(s_1, s_2) \in \rho$, and
(2) for every property $d \in Elem$ and state $s \in \Sigma$, $elem(d, s)$ holds iff $d \in \lambda(s)$.

Clause 1 of Definition 4 asserts that the property φ holds for an infinite list of states whose first element is s_0. Clauses 2–9 define the satisfaction relation $sat(X, \varphi)$ for any infinite list X and CTL* formula φ. The definition of $sat(X, \varphi)$ is by structural induction on φ. Clauses 10–12 establish that $path(X)$ holds iff for every pair (a_i, a_{i+1}) of consecutive elements on the infinite list X, we have that $(a_i, a_{i+1}) \in \rho$. Indeed, clauses 11 and 12 establish that $notpath(X)$ holds iff in the list X there exist two consecutive elements a_i and a_{i+1} such that $(a_i, a_{i+1}) \notin \rho$.

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 (in what follows, for any CTL* formula χ, we will denote by $|\chi|$ the number of occurrences of function symbols in r): for all states $a \in \Sigma$, for all infinite lists $\pi \in \Sigma^\omega$, and for all CTL* formulas ψ, (i) $\sigma(prop) = |\varphi| + 1$, where $prop \leftarrow sat([s_0|X], \varphi)$, (ii) $\sigma(sat(\pi, \psi)) = |\psi| + 1$, (iii) $\sigma(exists\_sat(a, \psi)) = |\psi| + 2$, (iv) $\sigma(path(\pi)) = 2$, (v) $\sigma(notpath(\pi)) = 1$, (vi) for every ground atom A, $\sigma(\neg A) = \sigma(A) + 1$, and (vii) in all other cases σ returns 0.

Example 2. Let us consider: (i) the set $Elem = \{a, b, tt\}$ of elementary properties, where tt is the elementary property which holds in every state, and

(ii) the Kripke structure $\mathcal{K} = \langle\{s_0, s_1, s_2\}, s_0, \rho, \lambda\rangle$, where ρ is the transition relation $\{(s_0, s_0), (s_0, s_1), (s_1, s_1), (s_1, s_2), (s_2, s_1)\}$ and λ is the function such that $\lambda(s_0) = \{a\}$, $\lambda(s_1) = \{b\}$, and $\lambda(s_2) = \{a\}$. Let us also consider the formula $\varphi = \mathsf{E}\,(a\,\mathsf{U}\,\neg\,\mathsf{E}\,(tt\,\mathsf{U}\,\neg\,(tt\,\mathsf{U}\,b)))$, which can be abbreviated as $\mathsf{E}\,(a\,\mathsf{U}\,\mathsf{AGF}\,b)$, where: (i) for every state formula ψ, $\mathsf{F}\,\psi$ (read '*eventually* ψ') stands for $tt\,\mathsf{U}\,\psi$, and $\mathsf{G}\,\psi$ (read '*always* ψ') stands for $\neg\,\mathsf{F}\neg\,\psi$, and (ii) for every path formula ψ, $\mathsf{A}\,\psi$ (read '*for all computation paths* ψ') stands for $\neg\,\mathsf{E}\,\neg\,\psi$. The encoding program $P_{\mathcal{K},\varphi}$ is as follows:

$prop \leftarrow sat([s_0|X], e(u(a, not(e(u(tt, not(u(tt, b))))))))$
$tr(s_0, s_0)\leftarrow$ $tr(s_0, s_1)\leftarrow$ $tr(s_1, s_1)\leftarrow$ $tr(s_1, s_2)\leftarrow$ $tr(s_2, s_1)\leftarrow$
$elem(a, s_0)\leftarrow$ $elem(b, s_1)\leftarrow$ $elem(a, s_2)\leftarrow$ $elem(tt, S)\leftarrow$

together with clauses 2–12 of Definition 4 defining the predicates *sat*, *path*, and *notpath*.

Since $\mathcal{K} \models \varphi$ holds iff there exists an infinite list $\pi \in \Sigma^\omega$ such that the first state of π is the initial state s_0 of $\mathcal{K}$ and $\mathcal{K}, \pi \models \varphi$ holds (see Definition 2), we have that the correctness of $P_{\mathcal{K},\varphi}$ can be expressed by stating that $\mathcal{K} \models \varphi$ holds iff $M(P_{\mathcal{K},\varphi}) \models \exists X\, sat([s_0|X], \varphi)$ iff (by clause 1 of Definition 4) $M(P_{\mathcal{K},\varphi}) \models prop$. The correctness of $P_{\mathcal{K},\varphi}$ is stated in the following theorem.

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

3 Transformational CTL* 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 state formula φ, $M(P_{\mathcal{K},\varphi}) \models prop$ holds, where $P_{\mathcal{K},\varphi}$ is constructed as indicated in Definition 4 above. Our technique consists of two steps. In the first step we transform the ω-program $P_{\mathcal{K},\varphi}$ into a *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 method for monadic ω-programs.

3.1 Transformation to Monadic ω-Programs

The first step of our model checking technique is realized by applying specialized versions of the following transformation rules: *definition introduction* and *elimination*, *instantiation*, *positive* and *negative unfolding*, *clause deletion*, *positive* and *negative folding* (see, for instance, [5,14,15]). These rules are applied according to a strategy which is a variant of the specialization strategy presented in [5].

Our specialization strategy starts off from the clause γ_1: $prop \leftarrow sat([s_0|X], \varphi)$ in $P_{\mathcal{K},\varphi}$ (see clause 1 in Definition 4) and a set of clauses, called *InDefs* which is initialized to $\{\gamma_1\}$. Then, our strategy iteratively applies two procedures: (i) the

instantiate-unfold procedure, and (ii) the *define-fold* procedure. At each iteration, the set *InDefs* is transformed into a set *Ds* of monadic ω-clauses, at the expense of possibly introducing some auxiliary, non-monadic clauses which are stored in the set *NewDefs*. These auxiliary clauses are given as input to a subsequent iteration of the strategy. The strategy terminates when no new auxiliary clauses are introduced and, when this happens, in a final step we apply the definition elimination rule by keeping only the clauses whose head predicate is either *prop* or a predicate on which *prop* depends.

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

$Q := P_{\mathcal{K},\varphi}$; $\quad \mathit{InDefs} := \{\mathit{prop} \leftarrow \mathit{sat}([s_0|X], \varphi)\}$; $\quad \mathit{Defs} := \mathit{InDefs}$;
while $\mathit{InDefs} \neq \emptyset$ **do**
  instantiate-unfold$(Q, \mathit{InDefs}, \mathit{Cs})$;
  define-fold$(\mathit{Cs}, \mathit{Defs}, \mathit{NewDefs}, \mathit{Ds})$;
  $Q := (Q - \mathit{InDefs}) \cup \mathit{NewDefs} \cup \mathit{Ds}$;
  $\mathit{InDefs} := \mathit{NewDefs}$; $\quad \mathit{Defs} := \mathit{Defs} \cup \mathit{NewDefs}$
od;
$T := \{\gamma \mid \gamma \in Q$ and the head predicate of γ is either *prop* or a predicate on which *prop* depends$\}$.

Let us now introduce two notions which are needed for presenting the *instantiate-unfold* and the *define-fold* procedures. A *definition clause* is a non-monadic ω-clause of the form $H \leftarrow A$ where: (1) H is an atom of the form p or $q(X)$, where q is a predicate of type `ilist`, (2) A is an atom, and (3) $\mathit{vars}(H) = \mathit{vars}(A)$. A *quasi-monadic clause* is an ω-clause of the form $H \leftarrow L_1 \wedge \ldots \wedge L_k$, with $k \geq 0$, such that: (i) H is an atom of the form p or $q([s|X])$, where p is a predicate of type `ilist` and $s \in \Sigma$, and (ii) for $i = 1, \ldots, k$, there exists a variable Y (possibly equal to X) of type `ilist` such that $\mathit{vars}(L_i) \subseteq \{Y\}$.

The *instantiate-unfold* procedure transforms a given set *InDefs* of definition clauses into a set *Cs* of quasi-monadic clauses by: (1) instantiating each clause in *InDefs*, (2) applying the positive (or negative) unfolding rule to clauses of the form $p([s|X]) \leftarrow B_L \wedge L \wedge B_R$, whenever L is a positive literal (or a negative literal, respectively), and (3) deleting subsumed clauses.

Given a clause δ, a variable X, and a term t, we denote by $\delta\{X/t\}$ the clause δ with every occurrence of X replaced by t.

The *instantiate-unfold* **Procedure.**
Input: An ω-program Q and a set $\mathit{InDefs} \subseteq Q$ of definition clauses.
Output: A set *Cs* of quasi-monadic clauses.

(*Instantiation*)
Let Y be a new variable of type `ilist` and let Σ be the set of states of $\mathcal{K}$;
$S := \{\, \delta\{X/[s|Y]\} \mid \delta \in \mathit{InDefs}$ and $\mathit{vars}(\delta) = \{X\}$ and $s \in \Sigma\} \cup$
$\{\, \delta \mid \delta \in \mathit{InDefs}$ and $\mathit{vars}(\delta) = \emptyset\}$;

$Cs := \emptyset$;

(*Unfolding*)
while there exists a clause γ in S **do**

(*Case* 1. *Positive Unfolding*)
if (i) γ is of the form $H \leftarrow B_L \wedge A \wedge B_R$, where A is an atom,
(ii) $K_1 \leftarrow B_1, \ldots, K_m \leftarrow B_m$ are *all* clauses in $P_{\mathcal{K},\varphi}$ such that A is unifiable with $K_1, \ldots, K_m$ with most general unifiers $\vartheta_1, \ldots, \vartheta_m$, and
(iii) for $i = 1, \ldots, m$, $A = K_i\vartheta_i$ (that is, A is an instance of K_i)
then $S := (S - \{\gamma\}) \cup \{H \leftarrow B_L \wedge B_1\vartheta_1 \wedge B_R, \ldots, H \leftarrow B_L \wedge B_m\vartheta_m \wedge B_R\}$

(*Case* 2. *Negative Unfolding*)
elseif (i) γ is of the form $H \leftarrow B_L \wedge \neg A \wedge B_R$, where A is an atom,
(ii) $K_1 \leftarrow B_1, \ldots, K_m \leftarrow B_m$ are *all* clauses in $P_{\mathcal{K},\varphi}$ such that A is unifiable with $K_1, \ldots, K_m$ with most general unifiers $\vartheta_1, \ldots, \vartheta_m$,
(iii) for $i = 1, \ldots, m$, $A = K_i\vartheta_i$ (that is, A is an instance of K_i), and
(iv) for $i = 1, \ldots, m$, $vars(B_i) \subseteq vars(K_i)$
then from $B_L \wedge \neg(B_1\vartheta_1 \vee \ldots \vee B_m\vartheta_m) \wedge B_R$ 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;
$S := (S - \{\gamma\}) \cup \{H \leftarrow Q_1, \ldots, H \leftarrow Q_r\}$

(*Case* 3. *No Unfolding*)
else $S := S - \{\gamma\}$; $Cs := Cs \cup \{\gamma\}$ **fi**
od;

(*Subsumption*)
while there exists a unit clause γ_1 in Cs of the form $H \leftarrow$ and a variant of a clause γ_2 in $Cs - \{\gamma_1\}$ of the form $H \leftarrow B$ **do** $Cs := Cs - \{\gamma_2\}$ **od**

The *define-fold* procedure transforms the quasi-monadic ω-clauses of Cs into monadic ω-clauses by applying the definition introduction rule and the (positive or negative) folding rule. In particular, for any given quasi-monadic clause γ: $H \leftarrow L_1 \wedge \ldots \wedge L_k$ in Cs and for $i = 1, \ldots, m$, the *define-fold* procedure performs the following steps.

Let L_i be either the positive literal A_i or the negative literal $\neg A_i$. We consider the following two cases. Case (1): If in $Defs \cup NewDefs$ there is a clause δ_i of the form $K_i \leftarrow A_i$, then γ is folded using δ_i, that is, the occurrence of L_i in the body of γ is replaced either (i) by K_i, if $L_i = A_i$ (positive folding), or (ii) by $\neg K_i$, if $L_i = \neg A_i$ (negative folding). Case (2): Otherwise, if in $Defs \cup NewDefs$ there is no clause of the form $K_i \leftarrow A_i$, then the definition clause δ_i: $K_i \leftarrow A_i$, where K_i has a new predicate symbol $newp_i$, is added to $NewDefs$ (by applying the definition introduction rule). Then, clause γ is folded using the newly introduced clause δ_i as in Case (1).

The clause $H \leftarrow M_1 \wedge \ldots \wedge M_k$ derived by folding γ using clauses $\delta_1, \ldots, \delta_k$ is a monadic ω-clause. Indeed, we have that: (1) H is either of the form p or of the form $q([s|X])$ (because γ is quasi-monadic), (2) for $i = 1, \ldots, k$, M_i is either the atom K_i or the negated atom $\neg K_i$, where K_i is either of the form $newp_i$ or of the

form $newp_i(Y)$ (this follows from the definition of δ_i and the fact that γ is quasi-monadic), and (3) Condition (iii) of Definition 3 holds by defining ℓ as follows: let σ be the stratification function for the encoding program $P_{\mathcal{K},\varphi}$ (see Section 2.2), (i) $\ell(prop) = \sigma(prop) = |\varphi| + 1$, and (ii) for every predicate $newp_i$ that occurs in the head of a clause $K_i \leftarrow A_i$ introduced during any execution of the *define-fold* procedure, $\ell(newp_i) = \sigma(A'_i)$, where A'_i is any ground instance of A_i. For example, if we introduce the definition clause $newp_i(X) \leftarrow sat(X, e(u(a,b)))$, then we define $\ell(newp_i) = \sigma(sat(\pi, e(u(a,b)))) = |e(u(a,b)))| + 1 = 5$, where π is any infinite list. Note that ℓ does not depend on the particular instance of A_i, because the value of σ is independent of the infinite list which (possibly) occurs as an argument of A_i.

The *define-fold* Procedure.
Input: (i) A set *Cs* of quasi-monadic clauses and (ii) a set *Defs* of definition clauses;
Output: (i) A set *NewDefs* of definition clauses, and (ii) a set *Ds* of monadic ω-clauses.

NewDefs := $\emptyset$; *Ds* := $\emptyset$;

for each clause γ in *Cs* **do**
  let the clause γ be of the form $H \leftarrow L_1 \wedge \ldots \wedge L_k$;
  for $i = 1, \ldots, k$ **do**
    let L_i be either A_i or $\neg A_i$, for some atom A_i;
(*Definition Introduction*)
    if a clause δ with body A_i has a variant in *Defs* $\cup$ *NewDefs*
    then take K_i to be the head of δ
    else take K_i to be: (i) $newp_i(Y)$, if $vars(A_i) = \{Y\}$, and (ii) $newp_i$, if $vars(A_i) = \emptyset$, where $newp_i$ is a new predicate symbol;
      NewDefs := *NewDefs* $\cup \{K_i \leftarrow A_i\}$ **fi**;
(*Positive or Negative Folding*)
    if L_i is A_i **then** $M_i := K_i$ **else** $M_i := \neg K_i$ **fi**
  od; *Ds* := *Ds* $\cup \{H \leftarrow M_1 \wedge \ldots \wedge M_k\}$
od

The specialization strategy, which from the initial program $P_{\mathcal{K},\varphi}$ produces the final program T, is correct w.r.t. the perfect model semantics, in the sense that $M(P_{\mathcal{K},\varphi}) \models prop$ iff $M(T) \models prop$. This correctness result can be proved similarly to [5,14]. Note that the instantiation rule that we use in the *unfold* procedure, is not present in [5,14], but its application can be viewed as an unfolding of an additional atom $ilist(X)$ defined by the clauses: $ilist([s_0|Y]) \leftarrow$, $\ldots$, $ilist([s_h|Y]) \leftarrow$, where $\Sigma = \{s_0, \ldots, s_h\}$ is the set of states of $\mathcal{K}$.

Our specialization strategy terminates for every input program $P_{\mathcal{K},\varphi}$ because: (i) both the *instantiate-unfold* and *define-fold* procedures terminate, and (ii) the while loop of the strategy terminates.

The termination of the *instantiate-unfold* procedure is a consequence from the following properties. (1) The Instantiation and Subsumption steps terminate.

(2) The predicates *path*, *tr*, and *elem* do not depend on themselves in program $P_{\mathcal{K},\varphi}$. (3) For each clause in $P_{\mathcal{K},\varphi}$ defining the predicate *notpath*, either the predicate of the body literal does not depend on *notpath* (see clause 11) or the term occurring in the body is a proper subterm of the term occurring in the head (see clause 12). (4) For each clause in $P_{\mathcal{K},\varphi}$ whose head is of the form $sat(l_1, \psi_1)$ and for each literal of the form $sat(l_2, \psi_2)$ occurring (positively or negatively) in the body of that clause, *either* ψ_2 is a proper subterm of ψ_1 *or* $\psi_1 = \psi_2$ and l_2 is a proper subterm of l_1. (5) For each state s and formula ψ, the literal $esists\_sat(s, \psi)$ depends on itself through a call to the predicate *sat* (see clauses 8 and 9) and by consuming at least one operator e in the formula ψ. (6) The applicability conditions given in the *instantiate-unfold* procedure (see Point (iii) of Case 1 and Case 2) do not allow the unfolding of a clause γ if this unfolding instantiates a variable in γ.

The termination of the *define-fold* procedure is straightforward.

Finally, the proof of termination of the while loop of the specialization strategy follows from the fact that only a finite number of definition clauses can be introduced by the *define-fold* procedure. Indeed, every definition clause is of the form $H \leftarrow A$, where: (i) A is an atom in the finite set $\Delta = \{notpath([s|X]) \mid s \in \Sigma\} \cup \{exists\_sat(s, \psi) \mid s \in \Sigma$ and ψ is a subformula of $\varphi\} \cup \{sat(X, \psi) \mid \psi$ is a subformula of $\varphi\}$, and (ii) for any $A \in \Delta$ the *define-fold* procedure introduces at most one definition clause.

Theorem 2 (Correctness and Termination of the Specialization Strategy). *Let $P_{\mathcal{K},\varphi}$ be the encoding program for a Kripke structure $\mathcal{K}$ and a state formula φ. The specialization strategy terminates for the input program $P_{\mathcal{K},\varphi}$ and returns an output program T such that: (i) T is a 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 specialization strategy starts off from the sets $Q = P_{\mathcal{K},\varphi}$ and $InDefs = Defs = \{\gamma_1\}$, where γ_1 is the following definition clause (that is, clause 1 of $P_{\mathcal{K},\varphi}$):

γ_1: $prop \leftarrow sat([s_0|X], e(u(a, not(e(u(tt, not(u(tt, b))))))))$

In the first execution of the loop body of our strategy we apply the *instantiate-unfold* procedure to the set *InDefs*. We get the set $Cs = \{\gamma_2, \gamma_3\}$ of quasi-monadic clauses, where:

γ_2: $prop \leftarrow \neg\, notpath([s_0|X]) \wedge sat(X, u(a, not(e(u(tt, not(u(tt, b))))))))$
γ_3: $prop \leftarrow \neg\, notpath([s_0|X]) \wedge \neg\, exists\_sat(s_0, u(tt, not(u(tt, b)))))$

Then, by applying the *define-fold* procedure, we get the set $NewDefs = \{\gamma_4, \gamma_5, \gamma_6\}$ of definition clauses and the set $Ds = \{\gamma'_2, \gamma'_3\}$ of monadic ω-clauses, where:

γ_4: $p_1(X) \leftarrow notpath([s_0|X])$
γ_5: $p_2(X) \leftarrow sat(X, u(a, not(e(u(tt, not(u(tt, b))))))))$
γ_6: $p_3 \leftarrow exists\_sat(s_0, u(tt, not(u(tt, b)))))$
γ'_2: $prop \leftarrow \neg\, p_1(X) \wedge p_2(X)$
γ'_3: $prop \leftarrow \neg\, p_1(X) \wedge \neg\, p_3$

At the end of the first execution of the body of the while loop of our strategy, we get: $Q = (P_{\mathcal{K},\varphi} - \{\gamma_1\}) \cup \{\gamma'_2, \gamma'_3\}$, $\mathit{InDefs} = \{\gamma_4, \gamma_5, \gamma_6\}$, and $\mathit{Defs} = \{\gamma_1\} \cup \{\gamma_4, \gamma_5, \gamma_6\}$. Since $\mathit{InDefs} \neq \emptyset$ the execution of the while loop continues. After a few more executions of the loop body, the *define-fold* procedure does not introduce any new clause in *NewDefs*. Thus, we get $\mathit{InDefs} = \emptyset$ and we derive the final program Q. By keeping every clause in Q whose head predicate is either *prop* or a predicate on which *prop* depends, we get the following monadic ω-program T:

$\mathit{prop} \leftarrow \neg\, p_1(X) \wedge p_2(X)$
$\mathit{prop} \leftarrow \neg\, p_1(X) \wedge \neg\, p_3$
$p_1([s_0|X]) \leftarrow p_1(X)$
$p_1([s_1|X]) \leftarrow p_4(X)$
$p_1([s_2|X]) \leftarrow$
$p_2([s_0|X]) \leftarrow \neg\, p_3$
$p_2([s_0|X]) \leftarrow p_2(X)$
$p_2([s_1|X]) \leftarrow \neg\, p_5$
$p_2([s_2|X]) \leftarrow \neg\, p_6$
$p_2([s_2|X]) \leftarrow p_2(X)$
$p_3 \leftarrow \neg\, p_1(X) \wedge \neg\, p_7(X)$
$p_3 \leftarrow \neg\, p_1(X) \wedge p_8(X)$
$p_4([s_0|X]) \leftarrow$
$p_4([s_1|X]) \leftarrow p_4(X)$
$p_4([s_2|X]) \leftarrow p_9(X)$
$p_5 \leftarrow \neg\, p_4(X) \wedge p_8(X)$
$p_6 \leftarrow \neg\, p_9(X) \wedge \neg\, p_7(X)$
$p_6 \leftarrow \neg\, p_9(X) \wedge p_8(X)$
$p_7([s_0|X]) \leftarrow p_7(X)$
$p_7([s_1|X]) \leftarrow$
$p_7([s_2|X]) \leftarrow p_7(X)$
$p_8([s_0|X]) \leftarrow \neg\, p_7(X)$
$p_8([s_0|X]) \leftarrow p_8(X)$
$p_8([s_1|X]) \leftarrow p_8(X)$
$p_8([s_2|X]) \leftarrow \neg\, p_7(X)$
$p_8([s_2|X]) \leftarrow p_8(X)$
$p_9([s_0|X]) \leftarrow$
$p_9([s_1|X]) \leftarrow p_4(X)$
$p_9([s_2|X]) \leftarrow$

3.2 A Proof Method for Monadic ω-Programs.

In this section we present the second step of our model checking technique. In particular, we present a method for checking whether or not $M(P) \models F$ holds, for any monadic ω-program P and any formula F which is either of the form p or of the form $\exists\, X(L_1 \wedge \ldots \wedge L_n)$, with $n \geq 1$, where, for $i = 1, \ldots, n$, L_i is either a positive literal $q_i(X)$ or a negative literal $\neg q_i(X)$. In what follows the set of the formulas F of this form will be denoted by $\mathcal{F}$. In particular, our method allows us to check whether or not $M(T) \models \mathit{prop}$ holds for the monadic ω-program T that we derive by the specialization strategy presented in Section 3.1.

First, we introduce the notion of a *derivation tree* and, then, the notion of a *proof* of a formula F in $\mathcal{F}$ w.r.t. a monadic ω-program P. Every node of a derivation tree has: (i) a *depth* which is the number of its ancestor nodes (in particular, the root has depth 0), and (ii) a *label* which is either

(1) the empty conjunction *true*, or
(2) the empty disjunction *false*, or
(3) a literal of the form: either p, or $\neg\, p$, or $q(X)$, or $\neg\, q(X)$, or
(4) a formula of the form: either $\exists X\, (L_1 \wedge \ldots \wedge L_n)$ or $\neg\, \exists X\, (L_1 \wedge \ldots \wedge L_n)$, with $n \geq 1$, where, for $i = 1, \ldots, n$, L_i is either $q_i(X)$ or $\neg\, q_i(X)$.

We denote by $\mathcal{L}$ the set of formulas of the forms (3) and (4). Let us also introduce the following notation: (i) for any atom A, $\overline{A}$ denotes $\neg A$ and $\overline{\neg A}$ denotes A, and (ii) for any formula B, $\overline{\exists X\, B}$ denotes $\neg \exists X\, B$.

In order to construct a derivation tree of a formula in $\mathcal{F}$ w.r.t. a given monadic ω-program P, we begin by rewriting the program P as follows. (Recall that in the body of a monadic ω-clause at most one variable occurs in a literal and two distinct literals may have a variable in common.) For every clause $H \leftarrow B$ in P

and for every variable Y in $vars(B) - vars(H)$, we replace the literals $L_1, \ldots, L_m$ of B such that $vars(L_1) = \ldots = vars(L_m) = \{Y\}$ by the formula $\exists Y\,(L_1 \wedge \ldots \wedge L_m)$. Thus, every clause in P is rewritten as $H \leftarrow F_1 \wedge \ldots \wedge F_k$, where, for $i = 1, \ldots, k$, F_i is a formula in $\mathcal{L}$.

For instance, clause $q_0([s|X]) \leftarrow q_1(X) \wedge q_2(Y) \wedge p_1 \wedge \neg\, q_3(Y) \wedge p_2$ is rewritten as $q_0([s|X]) \leftarrow q_1(X) \wedge \exists Y\,(q_2(Y) \wedge \neg\, q_3(Y)) \wedge p_1 \wedge p_2$.

Definition 5 (Derivation Tree). Given a monadic ω-program P and a formula F in $\mathcal{F}$, a *derivation tree* of F w.r.t. P is a *finite* tree T constructed as follows:

1. the root node is labeled by F, and if F is of the form $\exists X\,(L_1 \wedge \ldots \wedge L_n)$ then the root node has n children labeled by $L_1, \ldots, L_n$, respectively,
2. if a non-root node N is labeled by: (i) *true*, or (ii) *false*, or (iii) $\exists X\, B$, or (iv) $\neg\, \exists X\, B$ (that is, N is not labeled by a literal), then N is a leaf,
3. for every integer $d \geq 0$, consider the nodes $N_1, \ldots, N_\ell$, with $\ell \geq 1$, of depth d:
   if there exists an integer c, with $0 \leq c < d$, such that for every literal L labeling a node of depth d, there exists a node of depth c labeled by L
   then the nodes $N_1, \ldots, N_\ell$ are leaves
   else choose a state $s \in \Sigma$ and, for $i = 1, \ldots, \ell$, if the node N_i is labeled by a literal L_i, then construct a child node of N_i with label F, for each formula F in the set $\mathcal{C}_i$ of formulas in $\mathcal{L} \cup \{true, false\}$ constructed from the state s, the literal L_i, and the program P, as we now indicate. There are two cases.
   Case (i): L_i is an atom $q(X)$ (or p). If in P there is no clause whose head is $q([s|X])$ (or p), then take $\mathcal{C}_i$ to be $\{false\}$. Otherwise, choose a clause $q([s|X]) \leftarrow F_1 \wedge \ldots \wedge F_k$ (or $p \leftarrow F_1 \wedge \ldots \wedge F_k$) in P, where, for $i = 1, \ldots, k$, $F_i \in \mathcal{L}$. If $k = 0$ then take $\mathcal{C}_i$ to be $\{true\}$, else take $\mathcal{C}_i$ to be $\{F_1, \ldots, F_k\}$.
   Case (ii): L_i is a negated atom $\neg q(X)$ (or $\neg p$). Let $q([s|X]) \leftarrow B_1, \ldots, q([s|X]) \leftarrow B_k$ (or $p \leftarrow B_1, \ldots, p \leftarrow B_k$) be all clauses in P whose head is $q([s|X])$ (or p). If $k = 0$ then take $\mathcal{C}_i$ to be $\{true\}$. If $k \geq 1$ and there exists i, with $1 \leq i \leq k$, such that B_i is the empty conjunction, then take $\mathcal{C}_i$ to be $\{false\}$. Otherwise, for $i = 1, \ldots, k$, choose a formula $F_i \in \mathcal{L}$ such that $B_i = G_1 \wedge F_i \wedge G_2$, where G_1 and G_2 are (possibly empty) conjunctions, and take $\mathcal{C}_i$ to be $\{\overline{F}_1, \ldots, \overline{F}_k\}$.

By construction, for any derivation tree T there exist: (i) an integer m which is the maximal depth of a node of T, and (ii) a *least* integer c, with $0 \leq c < m$, such that for every literal L labeling a node of depth m, there exists a node of depth c labeled by L. Now, we introduce a relation r_T between literals as follows. For any two literals L_1 and L_2, $r_T(L_1, L_2)$ holds iff: (i) there exists a node M of depth c in T whose label is L_1, (ii) there exists a node N of depth m in T whose label is L_2, and (iii) M is an ancestor of N in T. We denote by r_T^+ the transitive closure of r_T.

Proposition 1. *Let P be a monadic ω-program and F be a formula in $\mathcal{F}$. (i) Every derivation tree T of F w.r.t. P is minimal, in the sense that no proper subtree*

of T is itself a derivation tree of F w.r.t. P. (ii) There exists a finite number of derivation trees of F w.r.t. P.

Now we present the definitions of proof and refutation, which are based on the notion of derivation tree.

Definition 6 (Proof and Refutation). Let P be a monadic ω-program and F be a formula in $\mathcal{F}$. We say that F has a *proof* w.r.t. P iff there exists a derivation tree T of F w.r.t. P which satisfies the following conditions:

1. every leaf N of T is labeled by: either (i) *true*, or (ii) a literal of the form p, or $\neg p$, or $q(X)$, or $\neg q(X)$, or (iii) a formula of the form $\exists X\, B$ that has a proof w.r.t. P, or (iv) a formula of the form $\neg\, \exists X\, B$ such that $\exists X\, B$ has a refutation w.r.t. P,
2. for every positive literal L labeling a leaf of T, $r_T^+(L, L)$ does not hold.

We say that F has a *refutation* w.r.t. P iff no derivation tree of F w.r.t. P is a proof of F w.r.t. P.

By Proposition 1 it is decidable whether or not a there exists a proof of a formula in $\mathcal{F}$ w.r.t. a monadic ω-program. Moreover, by induction on the level of the predicates occurring in the monadic ω-program P, we can show that our proof method is sound and complete for showing that a formula in the set $\mathcal{F}$ is true in the perfect model of P. Thus, we have the following result.

Theorem 3. *Let P be a monadic ω-program and F a formula in $\mathcal{F}$. Then: (i) there is an algorithm to check whether or not F has a proof w.r.t. P, and (ii) F has a proof w.r.t. P iff $M(P) \models F$.*

Now we present an example of application of the second step of our transformational method for proving CTL* properties of the Kripke structures which encode reactive systems.

Example 4. Let us consider: (i) the monadic ω-program T, obtained as the output of our specialization strategy (see Example 3), and (ii) the formula *prop*, that encodes the CTL* property φ of the Kripke structure $\mathcal{K}$ introduced in Example 2. We can construct a proof for the formula *prop* w.r.t. T as shown by the various derivation trees depicted in Figure 1. As a consequence, we have that $M(P_{\mathcal{K},\varphi}) \models \mathit{prop}$ holds and, thus, the formula φ holds in the Kripke structure $\mathcal{K}$.

4 Related Work and Concluding Remarks

Various logic programming techniques and tools have been developed for model checking. In particular, tabled resolution has been shown to be quite effective for implementing a modal μ-calculus model checker for CCS value passing programs [13]. Techniques based on logic programming, constraint solving, abstract interpretation, and program transformation have been proposed for

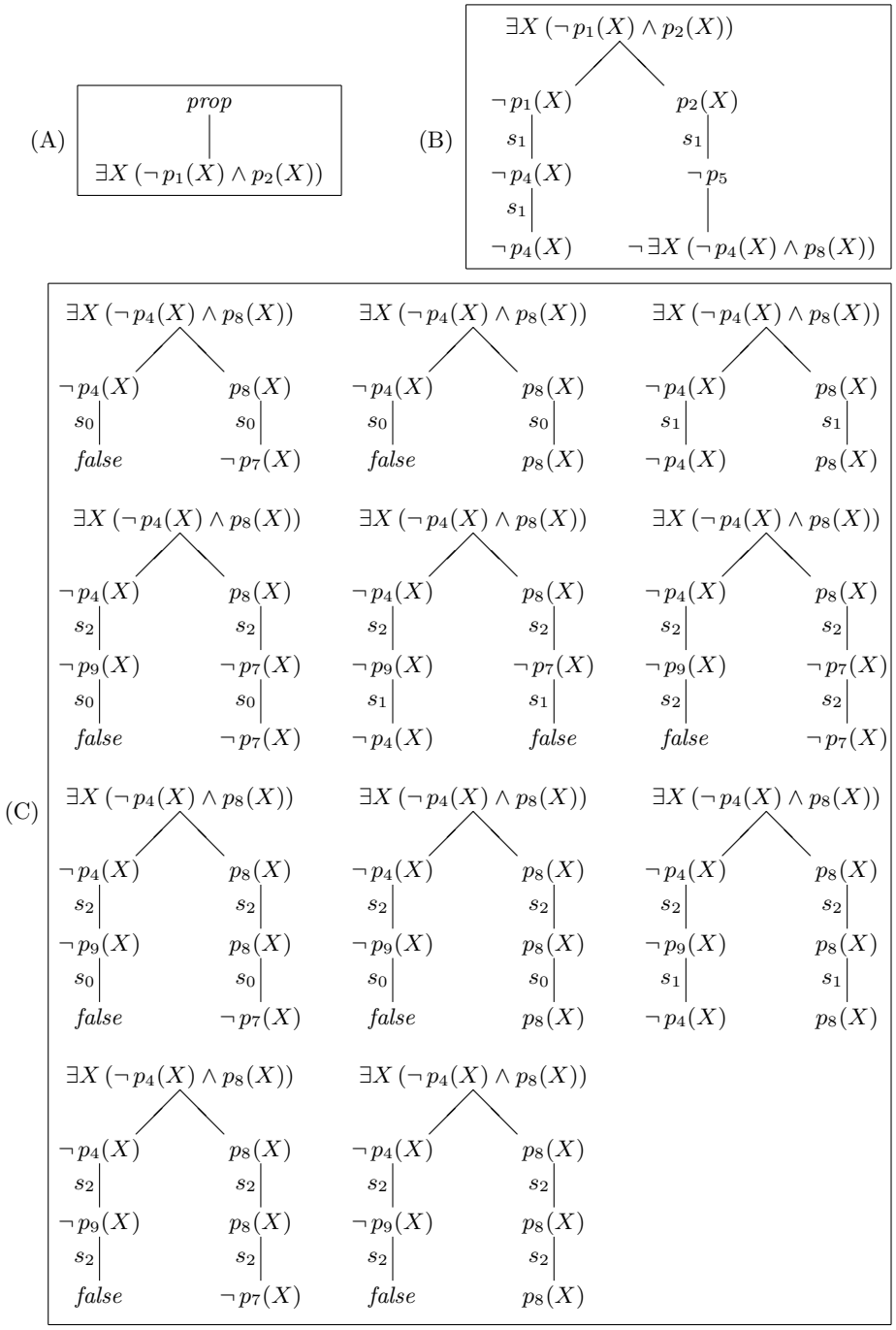

Fig. 1. The tree in (A) is a proof of *prop* w.r.t. T. The tree in (B) is a proof of $\exists X\,(\neg\, p_1(X) \wedge p_2(X))$ w.r.t. T. The 11 trees in (C) are all the derivation trees of $\exists X\,(\neg\, p_4(X) \wedge p_8(X))$ w.r.t. T and none of them is a proof. The labels of the arcs are the states $s \in \Sigma$ to be chosen according to Point (3) of Definition 5.

performing CTL model checking of finite and infinite state systems (see, for instance, [3,5,8,10]). In this paper we have extended to CTL* model checking the transformational approach which was proposed for LTL model checking in [11].

The main contributions of this work are the following. (i) We have proposed a method for specifying CTL* properties of reactive systems based on ω-programs, that is, logic programs acting on infinite lists. This method is a proper extension of the methods for specifying CTL or LTL properties, because CTL and LTL are fragments of CTL*. (ii) We have introduced the subclass of monadic ω-programs for which the satisfaction relation w.r.t. the perfect model is decidable. This subclass of programs properly extends the class of *linear monadic* ω-programs introduced in [11]. (iii) Finally, we have shown that we can transform, by applying semantics preserving unfold/fold rules, the logic programming specification of a CTL* property into a monadic ω-program.

Our transformation strategy can be viewed as a specialization of the Encoding Program (see Definition 4) w.r.t. a given Kripke structure $\mathcal{K}$ and a given CTL* formula φ. However, it should be noted that this program specialization could not be achieved by using partial deduction techniques (see [7] for a brief survey). Indeed, our specialization strategy performs instantiation and negative unfolding steps that cannot be realized by partial deduction.

Our two step verification approach bears some similarities with the automata-theoretic approach to CTL* model checking, where the specification of a finite state system and a CTL* formula are translated into alternating tree automata [6]. The automata-theoretic approach is quite appealing because many useful techniques are available in the field of automata theory. However, we believe that also our approach has its advantages because of the following reasons. (1) The specification of properties of reactive systems, together with the transformation of these specifications into monadic ω-programs, and the proofs of properties of monadic ω-programs, can all be done within the single framework of logic programming, while in the automata-theoretic approach one has to translate the temporal logic formalism into the distinct formalism of automata theory. (2) The translation of a specification into a monadic ω-program can be performed by using semantics preserving transformation rules, thereby avoiding the burden of proving the correctness of the translation by *ad-hoc* methods. (3) Finally, due its generality, we believe that our approach can be extended without much effort to the case of infinite state systems.

Issues that can be investigated in the future include: (i) the complexity of our verification method and, in particular, an efficient implementation of the proof method presented in Section 3.2, (ii) the relationship between monadic ω-programs and alternating tree automata, (iii) the applicability of our transformational approach to other logics, such as the monadic second order logic of successors, and (iv) the experimental evaluation of the efficiency of our transformational approach by considering various test cases and comparing its performance in practical examples w.r.t. that of other model checking techniques known in the literature.

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. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
3. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
4. E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.
5. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'01, Florence (Italy)*, Technical Report DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
6. O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, 2000.
7. M. Leuschel. Logic program specialisation. In J. Hatcliff and P. Thiemann (Eds.) T. Mogensen, editors, *Partial Evaluation - Practice and Theory*, Lecture Notes in Computer Science 1706, pages 155–188. Springer, 1998.
8. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation (LOPSTR '99), Venezia, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 2000.
9. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
10. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In J. W. Lloyd, editor, *Proceedings of the First International Conference on Computational Logic (CL 2000), London, UK, 24-28 July*, Lecture Notes in Artificial Intelligence 1861, pages 384–398. Springer-Verlag, 2000.
11. A. Pettorossi, M. Proietti, and V. Senni. Transformational verification of linear temporal logic. In *24th Italian Conference on Computational Logic June 24-26, 2009, Ferrara, Italy (CILC '09)*. http://www.ing.unife.it/eventi/cilc09.
12. 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.
13. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV '97)*, Lecture Notes in Computer Science 1254, pages 143–154. Springer-Verlag, 1997.
14. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
15. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming (ICLP'84)*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.