ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
"Antonio Ruberti"
CONSIGLIO NAZIONALE DELLE RICERCHE

A. Pettorossi, M. Proietti, V. Senni

DECIDING FULL BRANCHING TIME LOGIC BY PROGRAM TRANSFORMATION

R. 09-04, 2009

Alberto Pettorossi – Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy, and Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy.
Email : pettorossi@info.uniroma2.it. URL : http://www.iasi.cnr.it/~adp.

Maurizio Proietti – Istituto di Analisi dei Sistemi ed Informatica del CNR, Viale Manzoni 30, I-00185 Roma, Italy. Email : maurizio.proietti@iasi.cnr.it.
URL : http://www.iasi.cnr.it/~proietti.

Valerio Senni – Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy.
Email : senni@info.uniroma2.it. URL : http://www.disp.uniroma2.it/users/senni.

ISSN: 1128–3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti", CNR

viale Manzoni 30, 00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: iasi@iasi.cnr.it
URL: http://www.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 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 system for monadic ω-programs. This proof system can be encoded as a logic program which always terminates if the evaluation is performed by tabled resolution. Our verification algorithm has essentially the same time complexity of the best algorithms known in the literature.

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 [5]. 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 [3] for a survey). Most of the known model checking algorithms for CTL* either combine model checking algorithms for CTL and LTL [3], or use techniques based on translations to automata on infinite trees [8].

In this paper we extend to CTL* a method proposed in [13] for LTL. We encode the satisfaction relation of a CTL* formula φ with respect to a Kripke structure $\mathcal{K}$ by means of a 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 on infinite lists 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* [16] 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 which defines unary predicates on infinite lists. This transformation is performed by applying unfold/fold transformation rules similar to those presented in [7, 19, 20] according to a strategy which is a variant of the strategy for the elimination of multiple occurrences of variables [14]. Similarly to [7, 19], 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 set of proof rules for monadic ω-programs which are sound and complete with respect to the perfect model semantics. Moreover, those rules can be encoded in a straightforward way as a logic program which always terminates if it is evaluated by using tabled resolution [2, 18].

Our verification method based on very general transformation techniques, has essentially the same time complexity as the algorithms based on the techniques presented in [3, 8].

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 transformation rules and the strategy which allows us to transform an ω-program into a monadic ω-program. In Section 3.2 we present the proof rules for monadic ω-programs and the encoding of those proof rules as clauses of a logic program, and in Section 3.3 we study the computational complexity of our verification algorithm. Finally, in Section 4 we discuss the 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 a formula *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 [3] 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) $\lambda : \Sigma \rightarrow \mathcal{P}(\mathit{Elem})$ 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 2.1 (CTL* Formulas) Given a set *Elem* of elementary properties, a CTL* formula φ is either a *state formula* φ_s or a *path formula* φ_p defined as follows:

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

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

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

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

Definition 2.2 (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 $\mathcal{K}, \pi \models \varphi$ is inductively defined as follows:

$\mathcal{K}, \pi \models d$	iff	$\pi = [a_0, a_1, \ldots]$ and $d \in \lambda(a_0)$
$\mathcal{K}, \pi \models \neg\varphi$	iff	$\mathcal{K}, \pi \not\models \varphi$
$\mathcal{K}, \pi \models \varphi_1 \wedge \varphi_2$	iff	$\mathcal{K}, \pi \models \varphi_1$ and $\mathcal{K}, \pi \models \varphi_2$
$\mathcal{K}, \pi \models \mathsf{E}\, \varphi$	iff	$\pi = [a_0, a_1, \ldots]$ and there exists a computation path π' from a_0 such that $\mathcal{K}, \pi' \models \varphi$
$\mathcal{K}, \pi \models \mathsf{X}\, \varphi$	iff	$\mathcal{K}, \pi_1 \models \varphi$
$\mathcal{K}, \pi \models \varphi_1\, \mathsf{U}\, \varphi_2$	iff	there exists $i \geq 0$ such that $\mathcal{K}, \pi_i \models \varphi_2$ and, for all $0 \leq j < i$, $\mathcal{K}, \pi_j \models \varphi_1$.

Given a *state formula* φ, we say that $\mathcal{K}$ is a *model* of φ, written $\mathcal{K} \models \varphi$, 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 $\mathcal{K}, \pi \models \varphi$ holds.

The above definition of the satisfaction relation for CTL* formulas is a shorter, yet equivalent, version of the usual definition given in the literature [3].

In order to encode the satisfaction relation for CTL* formulas as a logic program, in the next section we will introduce 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 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 [11] 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 $\mathit{Fun} - (\Sigma \cup \{[\_|\_]\})$, 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 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 *$\omega$-program* is a logic program constructed as usual (see, for instance, [11]) 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'.

If f is a term or a formula, then by $\mathit{vars}(f)$ 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 $\mathit{Fun} - (\Sigma \cup \{[\_|\_]\})$ 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) 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 $a \in \Sigma$ 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 $\mathit{Fun} - (\Sigma \cup \{[\_|\_]\})$, and (iv) to every n-ary predicate $p \in \mathit{Pred}$ 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 (*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) which we denote $M(P)$ [1, 16] (in Example 1 below we present the construction of the perfect model of an ω-program).

Definition 2.3 (Monadic ω-Programs) A *monadic* ω-clause is of the form:

$p_0([s|X_0]) \leftarrow p_1(X_1) \wedge \ldots \wedge p_k(X_k) \wedge \neg p_{k+1}(X_{k+1}) \wedge \ldots \wedge \neg p_m(X_m)$

where: (i) $0 \leq k \leq m$, (ii) for $i = 0, \ldots, m$, $p_i \in \mathit{Pred}$, (iii) s is a constant of type `state`, (iv) X_0 is a variable of type `ilist`, and (v) $X_1, \ldots, X_k, X_{k+1}, \ldots, X_m$ are *distinct* variables of type `ilist`. A *monadic ω-program* is a stratified, finite set of monadic ω-clauses.

Note that in Definition 2.3 the predicate symbols $p_0, p_1, \ldots, p_m$ are *not* necessarily distinct and X_0 may be one of the variables in $\{X_1, \ldots, X_m\}$.

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 $\ell : \mathit{Pred} \rightarrow \mathbb{N}$ such that $\ell(p) = 2$, $\ell(q) = 1$, and $\ell(r) = 0$. The value of the predicate of an atom under ℓ is called the *level* of that atom. The perfect model $M(P)$ is constructed by increasing levels of ground atoms. We start from ground atoms of level 0, that is, 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 ground atoms of level 1, that is, 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 ground atoms of level 2, that is, 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* 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* that encodes the satisfaction relation for CTL* formulas, in the sense that for all computation paths π and CTL* formulas ψ, we have that $M(P_{\mathcal{K},\varphi}) \models sat(\pi,\psi)$ iff $\mathcal{K},\pi \models \psi$, and (iii) the unary predicate *prop* that encodes the property φ to be verified, in the sense that $prop(\pi)$ holds iff the first state of π is the initial state s_0 of $\mathcal{K}$ and $\mathcal{K},\pi \models \varphi$.

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

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

Clause 1 of Definition 2.4 states that the property φ holds for an infinite list whose first state is s_0. Clauses 2–4 stipulate 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 3 and 4 stipulate that $notpath(X)$ holds iff there exist two consecutive elements a_i and a_{i+1} on X such that $(a_i, a_{i+1}) \notin \rho$. Clauses 5–11 define the satisfaction relation $sat(X, \varphi)$ for any infinite list X and CTL* formula φ. The definition is given by cases on the structure of 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 (here, for any CTL* formula ψ, we denote by $|\psi|$ the number of occurrences of function symbols in ψ): for all states $a, a_1, a_2 \in \Sigma$, for all infinite lists $\pi \in \Sigma^\omega$, for all elementary properties $d \in Elem$, and for all CTL* formulas ψ, (i) $\sigma(prop(\pi)) = |\varphi| + 1$, (ii) $\sigma(path(\pi)) = 2$, (iii) $\sigma(notpath(\pi)) = 1$, (iv) $\sigma(tr(a_1, a_2)) = \sigma(elem(d, a)) = 0$, (v) $\sigma(sat(\pi, \psi)) = |\psi| + 1$, and (vi) for every ground atom A, $\sigma(\neg A) = \sigma(A) + 1$.

Example 2. Let us consider the set $Elem = \{a, b, tt\}$ of elementary properties, where tt is the elementary property which holds in all states, and 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 $\varphi = \mathsf{E}\neg(tt\,\mathsf{U}\neg a) \wedge \mathsf{E}\neg(tt\,\mathsf{U}\neg(tt\,\mathsf{U}\,b))$, which is usually abbreviated as $\mathsf{EG}\,a \wedge \mathsf{EGF}\,b$, where: (i) $\mathsf{F}\,\psi$ (*eventually* ψ) stands for $tt\,\mathsf{U}\,\psi$, and (ii) $\mathsf{G}\,\psi$ (*always* ψ) stands for $\neg\mathsf{F}\neg\psi$. The encoding program $P_{\mathcal{K},\varphi}$ is as follows:

$prop(X) \leftarrow sat([s_0|X], and(e(not(u(tt, not(a)))), e(not(u(tt, not(u(tt, b)))))))$
$path(X) \leftarrow \neg\, notpath(X)$
$notpath([S_1, S_2|X]) \leftarrow \neg\, tr(S_1, S_2)$
$notpath([S|X]) \leftarrow notpath(X)$
$tr(s_0, s_0) \leftarrow \quad tr(s_0, s_1) \leftarrow \quad tr(s_1, s_1) \leftarrow \quad tr(s_1, s_2) \leftarrow \quad tr(s_2, s_1) \leftarrow$
$elem(a, s_0) \leftarrow \; elem(b, s_1) \leftarrow \; elem(a, s_2) \leftarrow \; elem(tt, S) \leftarrow$

together with clauses 5–11 of Definition 2.4 defining the predicate *sat*.

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 2.2), 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 2.4) $M(P_{\mathcal{K},\varphi}) \models \exists X\, prop(X)$. Now, if we denote the statement $\exists X\, prop(X)$ by *Prop*, the correctness of $P_{\mathcal{K},\varphi}$ is stated by the following theorem.

Theorem 2.5 (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 2.4 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 set of proof rules 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*, *instantiation*, *positive* and *negative unfolding*, *clause deletion*, *positive* and *negative folding* (see, for instance, [7, 19, 20]). These rules are applied according to a transformation strategy which is a variant of the one for eliminating multiple occurrences of variables [14].

Our strategy starts off from the clause γ_1: $prop(X) \leftarrow sat([s_0|X], \varphi)$ in $P_{\mathcal{K},\varphi}$ (see clause 1 in Definition 2.4) and iteratively applies two procedures: (i) the *unfold* procedure, and (ii) the *define-fold* procedure. At each iteration, the set *InDefs* of clauses, which is initialized to $\{\gamma_1\}$, is transformed into a set *Es* 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. Our strategy terminates when no new auxiliary clauses are introduced.

The Transformation 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 $M(T) \models \mathit{Prop}$.

$T := \emptyset$; $\quad \mathit{Defs} := \{\mathit{prop}(X) \leftarrow \mathit{sat}([s_0|X], \varphi)\}$; $\quad \mathit{InDefs} := \mathit{Defs}$;
while $\mathit{InDefs} \neq \emptyset$ **do**
  $\mathit{unfold}(\mathit{InDefs}, \mathit{Ds})$;
  $\mathit{define\text{-}fold}(\mathit{Ds}, \mathit{Defs}, \mathit{NewDefs}, \mathit{Es})$;
  $T := T \cup \mathit{Es}$; $\quad \mathit{Defs} := \mathit{Defs} \cup \mathit{NewDefs}$; $\quad \mathit{InDefs} := \mathit{NewDefs}$
od;

Let us now introduce some notions needed for presenting the *unfold* procedure and the *define-fold* procedure. A conjunction of literals $L_1 \wedge \ldots \wedge L_m$, with $m \geq 1$, is called a *block* if, for some variable X of type `ilist`, $\mathit{vars}(L_1) = \cdots = \mathit{vars}(L_m) = \{X\}$. A block is said to be *positive* if it contains at least one positive literal. Otherwise, it is said to be *negative*. Two blocks B_1 and B_2 are *disjoint* if $\mathit{vars}(B_1) \cap \mathit{vars}(B_2) = \emptyset$

A *definition clause* is a (non-monadic) ω-clause of the form $p(X) \leftarrow B$, where B is a positive block and $\mathit{vars}(B) = \{X\}$. A *quasi-monadic* clause is a clause of the form $p([s|X]) \leftarrow B_1 \wedge \ldots \wedge B_l$, where $l \geq 0$ and $B_1, \ldots, B_l$ are pairwise disjoint blocks. Thus, a monadic ω-clause is a quasi-monadic clause where every block consists of exactly one literal.

The *unfold* procedure takes as input a set *InDefs* of definition clauses and returns as output a set *Ds* of quasi-monadic clauses. (Note that, in particular, clause $\mathit{prop}(X) \leftarrow \mathit{sat}([s_0|X], \varphi)$ is a definition clause.) The *unfold* procedure starts off by instantiating every variable of type `ilist` occurring in *InDefs* by a term of the form $[s|Y]$, where s is a state in Σ and Y is a new variable of type `ilist`, thereby deriving a new set *Cs* of clauses. Then, the *unfold* procedure repeatedly applies as long as possible the positive and negative unfolding rules starting from the set *Cs* of instantiated clauses. Finally, the *unfold* procedure deletes clauses that are subsumed by other clauses. By reasoning on the structure of the program $P_{\mathcal{K},\varphi}$ one can prove that the output of the *unfold* procedure is a set *Ds* of quasi-monadic clauses.

The *unfold* Procedure.
Input: A set *InDefs* of definition clauses. *Output*: A set *Ds* of quasi-monadic clauses.

(*Instantiation*)
let Y be a new variable of type `ilist` and let $s_0, \ldots, s_h$ be the states of $\mathcal{K}$

$\mathit{Cs} := \{\, C\{X/[s_0|Y]\}, \ldots, C\{X/[s_h|Y] \mid C \in \mathit{InDefs} \text{ and } \mathit{vars}(C) = \{X\}\}$

(*Unfolding*)
while there exists a clause C in *Cs* **do**

(*Case* 1. *Positive Unfolding*)
if (i) C is of the form $p([s|Y]) \leftarrow G_1 \wedge A \wedge G_2$, where A is a positive literal, and $K_1 \leftarrow B_1, \ldots, K_m \leftarrow B_m$ 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
(ii) for $i = 1, \ldots, m$, $A = K_i\vartheta_i$ (that is, A is an instance of K_i)

then $\mathit{Cs} := (\mathit{Cs} - \{C\}) \cup \{p([s|Y]) \leftarrow G_1 \wedge B_1\vartheta_1 \wedge G_2, \ldots, p([s|Y]) \leftarrow G_1 \wedge B_m\vartheta_m \wedge G_2\}$

(*Case* 2. *Negative Unfolding*)

elseif (i) C is of the form $p([s|Y]) \leftarrow G_1 \wedge \neg A \wedge G_2$ and $K_1 \leftarrow B_1, \ldots, K_m \leftarrow B_m$ are *all* clauses in $P_{\mathcal{K},\varphi}$ such that A is unifiable with $K_1, \ldots, K_m$ with most general unifiers $\vartheta_1, \ldots, \vartheta_m$,
(ii) for $i = 1, \ldots, m$, $A = K_i\vartheta_i$ (that is, A is an instance of K_i), and
(iii) for $i = 1, \ldots, m$, $vars(B_i) \subseteq vars(K_i)$

then from $G_1 \wedge \neg(B_1\vartheta_1 \vee \ldots \vee B_m\vartheta_m) \wedge G_2$ 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;
$Cs := (Cs - \{C\}) \cup \{p([s|Y]) \leftarrow Q_1, \ldots, p([s|Y]) \leftarrow Q_r\}$

(*Case* 3. *No Unfolding*)

else $Cs := (Cs - \{C\})$; $Ds := Ds \cup \{C\}$

od;

(*Subsumption*)

while there exists a clause C_1 in Ds of the form $p([s|Y]) \leftarrow G_1$ and a variant of a clause C_2 in $Ds - \{C_1\}$ of the form $p([s|Y]) \leftarrow G_1 \wedge G_2$ **do** $Ds := Ds - \{C_2\}$ **od**

The *define-fold* procedure transforms every quasi-monadic clause $p([s|X]) \leftarrow B_1 \wedge \ldots \wedge B_l$ in Ds into a monadic ω-clause by applying, for $i = 1, \ldots, l$, to each block B_i the positive or negative folding rule as follows. If B_i is a positive block, then the *define-unfold* procedure introduces a new definition clause N of the form $q_i(Y_i) \leftarrow B_i$, where q_i is a fresh new predicate symbol and $vars(B_i) = \{Y_i\}$, unless that clause N already belongs to *Defs* (modulo the name of the head predicate). Clause N is added to the set *Defs* of definition clauses which can be used for subsequent folding steps. Clause N is also added to the set *InDefs* of definition clauses. (*InDefs* will be processed in a subsequent execution of the body of the while loop of the strategy.)

Otherwise, if B_i is a negative block of the form $\neg A_1 \wedge \ldots \wedge \neg A_m$, the *define-fold* procedure introduces m new definition clauses N_1: $r_i(Y_i) \leftarrow A_1$, ..., N_m: $r_i(Y_i) \leftarrow A_m$, where r_i is a fresh new predicate symbol and $vars(B_i) = \{Y_i\}$, unless those clauses already belong to *Defs* (modulo the name of the head predicate). The definition clauses $N_1, \ldots, N_m$ are added to *Defs* and to *InDefs*.

Finally, we obtain a monadic ω-clause $p([s|X]) \leftarrow M_1 \wedge \ldots \wedge M_l$ as follows. For $i = 1, \ldots, l$, if B_i is a positive block then the we apply the positive folding rule to B_i using clause N and, thus, the literal M_i is $q_i(Y_i)$. Otherwise, if B_i is a negative block, we apply the negative folding rule to B_i using clauses $N_1, \ldots, N_m$ and, thus, the literal M_i is $\neg\, r_i(Y_i)$. (Note that X and Y_i may be identical.)

The *define-fold* Procedure.

Input: (i) A set *Ds* of quasi-monadic clauses and (ii) a set *Defs* of definition clauses; *Output*: (i) A set *NewDefs* of definition clauses and (ii) a set *Es* of monadic ω-clauses.

$NewDefs := \emptyset$; $Es := \emptyset$;
for each clause $D \in Ds$ **do**
if D is a monadic ω-clause **then** $Es := Es \cup \{D\}$ **else**
let D be of the form $p([s|X]) \leftarrow B_1 \wedge \ldots \wedge B_l$, where $B_1, \ldots, B_l$ are pairwise disjoint blocks.
for $i = 1, \ldots, l$ **do**
let $B_i = L_1 \wedge \ldots \wedge L_m$ and $vars(L_1) = \ldots = vars(L_m) = \{Y_i\}$

(*Case* 1. *Positive Define-Fold*)

if for some $j \in \{1, \ldots, m\}$, L_j is a *positive* literal

then if there exists a clause N of the form $q_i(Y_i) \leftarrow L_1 \wedge \ldots \wedge L_m$ such that $N \in \mathit{Defs} \cup \mathit{NewDefs}$ and the predicate symbol q_i does not occur in $(\mathit{Defs} \cup \mathit{NewDefs}) - \{N\}$

then $M_i := q_i(Y_i)$

else $\mathit{NewDefs} := \mathit{NewDefs} \cup \{r_i(Y_i) \leftarrow L_1 \wedge \ldots \wedge L_m\}$, where r_i is a fresh new predicate symbol;
$M_i := r_i(Y_i)$

(*Case* 2. *Negative Define-Fold*)

else for $j = 1, \ldots, m$, let L_j be a *negative* literal $\neg A_j$

if there exists a set Ns of clauses of the form $\{q_i(Y_i) \leftarrow A_1, \ldots, q_i(Y_i) \leftarrow A_m\}$ such that $Ns \subseteq \mathit{Defs} \cup \mathit{NewDefs}$ and q_i does not occur in $(\mathit{Defs} \cup \mathit{NewDefs}) - Ns$

then $M_i := \neg\, q_i(Y_i)$

else $\mathit{NewDefs} := \mathit{NewDefs} \cup \{r_i(Y_i) \leftarrow A_1, \ldots, r_i(Y_i) \leftarrow A_m\}$, where r_i is a fresh new predicate symbol;
$M_i := \neg\, r_i(Y_i)$

od;

$\mathit{Es} := \mathit{Es} \cup \{H \leftarrow M_1 \wedge \ldots \wedge M_l\}$

od

The transformation 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 $M(P_{\mathcal{K},\varphi}) \models \mathit{Prop}$ iff $M(T) \models \mathit{Prop}$. This correctness result can be proved similarly to [7, 19]. Note that the instantiation rule we use in the *unfold* procedure is not present in [7, 19], but its application can be viewed as an unfolding of an additional atom $\mathit{ilist}(X)$ defined by the clauses: $\mathit{ilist}([s_0|Y]) \leftarrow$, $\ldots$, $\mathit{ilist}([s_h|Y]) \leftarrow$.

Our transformation strategy terminates for every input program $P_{\mathcal{K},\varphi}$. The proof of termination of the *unfold* procedure is based on 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 3) or the term occurring in the body is a proper subterm of the term occurring in the head (see clause 4). (4) For each clause in $P_{\mathcal{K},\varphi}$ whose head is of the form $\mathit{sat}(l_1, \psi_1)$ and for each literal of the form $\mathit{sat}(l_2, \psi_2)$ 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) The applicability conditions given in the *unfold* procedure (see Point (ii) of Case 1 and Case 2) do not allow the unfolding of a clause C if this unfolding instantiates a variable in C.

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

Finally, the proof of termination of the while loop of the transformation strategy follows from the fact that only a finite number of new clauses can be introduced by the *define-fold* procedure. Indeed, every new clause is of the form $\mathit{newp}(X) \leftarrow L_1 \wedge \ldots \wedge L_m$, where for $i = 1, \ldots, m$, either (a) L_i is an atom A_i or a negated atom $\neg A_i$, where A_i belongs to the set $\{\mathit{notpath}(X)\} \cup \{\mathit{notpath}([s|X]) \mid s \in \Sigma\} \cup \{\mathit{sat}(X, \psi) \mid \psi$ is a subformula of $\varphi\}$, or (b) L_i belongs to the set $\{\neg\, \mathit{sat}([s|X], e(\psi)) \mid s \in \Sigma$ and ψ is a subformula of $\varphi\}$.

Theorem 3.1 (Correctness and Termination of the Transformation Strategy) *Let $P_{\mathcal{K},\varphi}$ be the encoding program for a Kripke structure $\mathcal{K}$ and a state formula φ. The transformation strategy terminates for the input program $P_{\mathcal{K},\varphi}$ and returns the 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 transformation strategy starts off from the sets $T = \emptyset$ and $Defs = InDefs = \{\gamma_1\}$, where γ_1 is the following definition clause:

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

In the first execution of the loop body of our strategy we apply the *unfold* procedure to the set *InDefs*. We get the set $Ds = \{\gamma_2, \gamma_3, \gamma_4\}$ of quasi-monadic clauses, where:

γ_2: $prop([s_0|X]) \leftarrow \neg\, notpath([s_0|Y]) \wedge \neg\, sat(Y, u(tt, not(a))) \wedge$
$\neg\, notpath([s_0|Z]) \wedge sat(Z, u(tt, b)) \wedge \neg\, sat(Z, u(tt, not(u(tt, b))))$

γ_3: $prop([s_1|X]) \leftarrow \neg\, notpath([s_0|Y]) \wedge \neg\, sat(Y, u(tt, not(a))) \wedge$
$\neg\, notpath([s_0|Z]) \wedge sat(Z, u(tt, b)) \wedge \neg\, sat(Z, u(tt, not(u(tt, b))))$

γ_4: $prop([s_2|X]) \leftarrow \neg\, notpath([s_0|Y]) \wedge \neg\, sat(Y, u(tt, not(a))) \wedge$
$\neg\, notpath([s_0|Z]) \wedge sat(Z, u(tt, b)) \wedge \neg\, sat(Z, u(tt, not(u(tt, b))))$

Then, by applying the *define-fold* procedure, we get the set $NewDefs = \{\gamma_5, \gamma_6, \gamma_7\}$ of definition clauses and the set $Es = \{\gamma'_2, \gamma'_3, \gamma'_4\}$ of monadic ω-clauses (note that the body of each clause in *Ds* is partitioned into two blocks), where:

γ_5: $p_1(X) \leftarrow notpath([s_0|X])$
γ_6: $p_1(X) \leftarrow sat(X, u(tt, not(a)))$
γ_7: $p_2(X) \leftarrow \neg\, notpath([s_0|X]) \wedge sat(X, u(tt, b)) \wedge \neg\, sat(X, u(tt, not(u(tt, b))))$

γ'_2: $prop([s_0|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
γ'_3: $prop([s_1|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
γ'_4: $prop([s_2|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$

Thus, at the end of the first execution of the body of the while loop of our strategy, we get: $T = \{\gamma'_2, \gamma'_3, \gamma'_4\}$, $Defs = \{\gamma_1\} \cup \{\gamma_5, \gamma_6, \gamma_7\}$, and $InDefs = \{\gamma_5, \gamma_6, \gamma_7\}$. Since $InDefs \neq \emptyset$ we execute again the body the while loop. After a few more executions we get the following monadic ω-program T:

$prop([s_0|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
$prop([s_1|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
$prop([s_2|X]) \leftarrow \neg\, p_1(Y) \wedge p_2(Z)$
$p_1([s_0|X]) \leftarrow p_{10}(X)$
$p_1([s_0|X]) \leftarrow p_{11}(X)$
$p_1([s_1|X]) \leftarrow$
$p_1([s_2|X]) \leftarrow$
$p_2([s_0|X]) \leftarrow p_2(X)$
$p_2([s_1|X]) \leftarrow \neg\, p_3(X)$
$p_2([s_1|X]) \leftarrow p_4(X)$
$p_3([s_0|X]) \leftarrow$
$p_3([s_1|X]) \leftarrow p_9(X)$
$p_3([s_1|X]) \leftarrow p_7(X)$
$p_3([s_2|X]) \leftarrow p_5(X)$
$p_3([s_2|X]) \leftarrow \neg\, p_6(X)$
$p_3([s_2|X]) \leftarrow p_7(X)$
$p_4([s_1|X]) \leftarrow \neg\, p_3(X)$
$p_4([s_1|X]) \leftarrow p_4(X)$
$p_4([s_2|X]) \leftarrow p_8(X)$
$p_5([s_0|X]) \leftarrow$
$p_5([s_1|X]) \leftarrow p_9(X)$
$p_5([s_2|X]) \leftarrow$
$p_6([s_0|X]) \leftarrow p_6(X)$
$p_6([s_1|X]) \leftarrow$
$p_6([s_2|X]) \leftarrow p_6(X)$
$p_7([s_0|X]) \leftarrow \neg\, p_6(X)$
$p_7([s_0|X]) \leftarrow p_7(X)$
$p_7([s_1|X]) \leftarrow p_7(X)$
$p_7([s_2|X]) \leftarrow \neg\, p_6(X)$
$p_7([s_2|X]) \leftarrow p_7(X)$
$p_8([s_1|X]) \leftarrow \neg\, p_3(X)$
$p_8([s_1|X]) \leftarrow p_4(X)$
$p_9([s_0|X]) \leftarrow$
$p_9([s_1|X]) \leftarrow p_9(X)$
$p_9([s_2|X]) \leftarrow p_5(X)$
$p_{10}([s_0|X]) \leftarrow p_{10}(X)$
$p_{10}([s_1|X]) \leftarrow p_9(X)$
$p_{10}([s_2|X]) \leftarrow$
$p_{11}([s_0|X]) \leftarrow p_{11}(X)$
$p_{11}([s_1|X]) \leftarrow$
$p_{11}([s_2|X]) \leftarrow p_{11}(X)$

S1. $\dfrac{}{P \vdash \mathit{true}}$ S2. $\dfrac{P \nvdash F}{P \vdash \neg F}$

S3. $\dfrac{P \vdash F}{P \vdash \exists(F)}$ if $\mathit{closed\_literal}(F)$

S4. $\dfrac{P \vdash \exists(F)}{P \vdash \exists(p(X))}$ if there exists $p([s|Y]) \leftarrow F \in P$ for some $s \in \{s_0, \ldots, s_h\}$

S5. $\dfrac{P \vdash \neg\forall(p(X))}{P \vdash \exists(\neg p(X))}$ S6. $\dfrac{P \vdash \exists(F_1) \quad P \vdash \exists(F_2)}{P \vdash \exists(F_1 \wedge F_2)}$

S7. $\dfrac{P \vdash F}{P \vdash \forall(F)}$ if $\mathit{closed\_literal}(F)$

S8. $\dfrac{P \vdash \forall(F_0) \quad \ldots \quad P \vdash \forall(F_h)}{P \vdash \forall(p(X))}$ if $\{p([s_0|Y]) \leftarrow F_0, \ldots, p([s_h|Y]) \leftarrow F_h\} \subseteq P$

S9. $\dfrac{P \vdash \neg\exists(p(X))}{P \vdash \forall(\neg p(X))}$ S10. $\dfrac{P \vdash \forall(F_1) \quad P \vdash \forall(F_2)}{P \vdash \forall(F_1 \wedge F_2)}$

Figure 1: Proof system for monadic ω-programs. $\Sigma = \{s_0, \ldots, s_h\}$ is the set of states of the Kripke structure $\mathcal{K}$. For any formula F, $\mathit{closed\_literal}(F)$ holds iff F is either the formula *true* or the formula $\exists(L)$, where L is a literal.

3.2. A Proof System for Monadic ω-Programs.

Now we present a proof system for checking whether or not $M(P) \models F$ holds for any monadic ω-program P and any quantified literal F. Thus, in particular, we will be able to check whether or not $M(T) \models \mathit{Prop}$ holds for the monadic ω-program T derived by the transformation strategy presented in Section 3.1 and the quantified literal *Prop* which encodes the state formula to be verified (recall that *Prop* stands for $\exists X\, \mathit{prop}(X)$).

Every monadic ω-program P used by the proof system is first rewritten as follows. (i) Every clause of the form $H \leftarrow$ is rewritten as $H \leftarrow \mathit{true}$ (so that no clause in P has an empty body), and (ii) for every clause of the form $H \leftarrow G$, each literal L occurring in G such that $\mathit{vars}(H) \cap \mathit{vars}(L) = \emptyset$ is replaced by its existential closure $\exists(L)$. (Recall that in the body of a monadic ω-clause two distinct literals do not share any variable.) For instance, the clause $p([s|X]) \leftarrow q(X) \wedge p(Y)$ is rewritten as $p([s|X]) \leftarrow q(X) \wedge \exists Y\, p(Y)$.

The proof rules presented in Figure 1 define a relation $\vdash$ such that $P \vdash F$ iff $M(P) \models F$, for every monadic ω-program P and closed formula F, which is of one of the following forms: *true*, $\forall(F_1 \wedge \ldots \wedge F_k)$, $\exists(F_1 \wedge \ldots \wedge F_k)$, where $k \geq 1$ and for $i = 1, \ldots k$, F_i is either a literal or the existential closure of a literal.

Note that the proof rule S2 is the *negation as failure* rule, that is, the negated judgement '$P \nvdash F$' should be interpreted as '$P \vdash F$ cannot be proved by using the proof rules S1–S10'. This interpretation of $P \nvdash F$ as (finite or infinite) failure of $P \vdash F$ is meaningful because P is stratified and, thus, also the instances of the proof rules are stratified. The stratification of these instances is induced by the existence of a mapping μ from formulas to natural numbers such that, for every rule instance with conclusion $P \vdash F_1$, (i) if $P \vdash F_2$ occurs as a premise, then $\mu(F_1) \geq \mu(F_2)$, and (ii) if $P \nvdash F_3$ occurs as a premise, then $\mu(F_1) > \mu(F_3)$. Thus, when

constructing a proof of $P \vdash F$ it is never required to show that $P \nvdash F$, that is, it is never required to show that no proof of $P \vdash F$ can be constructed.

Note also that the proof rule S6 is sound because, as already mentioned, the literals in the body of a monadic ω-clause do not share any variable and, therefore, the existential quantifier distributes over the conjunction of those literals.

By induction on the strata of the monadic ω-program P we can show that the proof rules of Figure 1 are sound and complete for proving that a quantified literal F is true in the perfect model of P, as stated by the following theorem.

Theorem 3.2. *Let P be a monadic ω-program and F a formula of the form $\exists(L)$ or $\forall(L)$, where L is a literal. Then, $P \vdash F$ iff $M(P) \models F$.*

The proof system for monadic ω-programs given in Figure 1 can be encoded as a logic program, called *Demo*. Given a monadic ω-program P and a closed literal F, the program *Demo* uses the (ground) representations $\lceil P \rceil$ and $\lceil F \rceil$ of P and F, respectively, which are constructed as follows. Let v be a new constant symbol. (i) For any variable X, $\lceil X \rceil$ is v, (ii) for any list $[s|X]$, where s is a state and X is a variable, $\lceil [s|X] \rceil$ is $[s|v]$, (iii) for any unary predicate q and term t, $\lceil q(t) \rceil$ is $(q, \lceil t \rceil)$, (iv) for any atom A, $\lceil \neg A \rceil$ is $not(\lceil A \rceil)$, (v) for any conjunction $F_1 \wedge F_2$, $\lceil F_1 \wedge F_2 \rceil$ is $and(\lceil F_1 \rceil, \lceil F_2 \rceil)$, (vi) for any F which is either a literal or a conjunction, $\lceil \exists(F) \rceil$ is $exists(\lceil F \rceil)$ and $\lceil \forall(F) \rceil$ is $all(\lceil F \rceil)$, (vii) for any clause C of the form $H \leftarrow$, $\lceil C \rceil$ is the unit clause $clause(\lceil H \rceil, true) \leftarrow$, (viii) for any clause C of the form $H \leftarrow F$, $\lceil C \rceil$ is the unit clause $clause(\lceil H \rceil, \lceil F \rceil) \leftarrow$, and, finally, (ix) for any monadic ω-program $P = \{C_1, \ldots, C_n\}$, $\lceil P \rceil$ is the set of ground unit clauses $\{\lceil C_1 \rceil, \ldots, \lceil C_n \rceil\}$.

D1. $demo(true) \leftarrow$
D2. $demo(not(F)) \leftarrow \neg\, demo(F)$
D3. $demo(exists(F)) \leftarrow closed\_literal(F) \wedge demo(F)$
D4. $demo(exists((R, v))) \leftarrow clause((R, [S|v]),\, F) \wedge demo(exists(F))$
D5. $demo(exists(not((R, v)))) \leftarrow demo(not(all((R, v))))$
D6. $demo(exists(and(F_1, F_2))) \leftarrow demo(exists(F_1)) \wedge demo(exists(F_2))$
D7. $demo(all(F)) \leftarrow closed\_literal(F) \wedge demo(F)$
D8. $demo(all((R, v))) \leftarrow clause((R, [s_0|v]),\, F_0) \wedge demo(all(F_0)) \wedge \ldots \wedge$
$clause((R, [s_h|v]),\, F_h) \wedge demo(all(F_h))$
D9. $demo(all(not((R, v)))) \leftarrow demo(not(exists((R, v))))$
D10. $demo(all(and(F_1, F_2))) \leftarrow demo(all(F_1)) \wedge demo(all(F_2))$

Since P is a stratified program, $Demo \cup \lceil P \rceil$ is a *weakly stratified* program [15] and, hence, it has a unique perfect model $M(Demo \cup \lceil P \rceil)$.

Theorem 3.3. *Let P be a monadic ω-program and F be a formula of the form $\exists(L)$ or $\forall(L)$, where L is a literal. Then, $P \vdash F$ iff $M(Demo \cup \lceil P \rceil) \models demo(\lceil F \rceil)$.*

Thus, by Theorems 3.2 and 3.3 for any monadic ω-program T derived from $P_{\mathcal{K},\varphi}$ using the transformation strategy described in Section 3.1, we can check whether or not $M(T) \models Prop$ holds (and, thus, whether or not $\mathcal{K} \models \varphi$ holds) by using any logic programming system which computes the perfect model of $Demo \cup \lceil T \rceil$. One can use, for instance, a system based on *tabled resolution* [2, 18] which guarantees the termination of the evaluation of the query $demo(\lceil Prop \rceil)$ and returns 'yes' iff $M(Demo \cup \lceil T \rceil) \models demo(\lceil Prop \rceil)$. Indeed, starting from $demo(\lceil Prop \rceil)$, we can only derive a finite set of queries of the form $demo(\lceil F \rceil)$, and the tabling mechanism ensures that each query is evaluated at most once.

Example 4. Let T be the monadic ω-program obtained in Example 3 as the output of our transformation strategy. In order to prove that $T \vdash \exists X\, prop(X)$, we compute the encoding $\lceil T \rceil$ of the program T and the encoding $exists((prop, v))$ of the property $\exists X\, prop(X)$. Then, we evaluate the query $demo(exists((prop, v)))$ w.r.t. the program $Demo \cup \lceil T \rceil$ by using a system based on tabled resolution and we obtain the answer 'yes'. Since $T \vdash \exists X\, prop(X)$, we have that $M(P_{\mathcal{K},\varphi}) \models Prop$ and, thus, we get that the formula φ holds in the Kripke structure $\mathcal{K}$ (see Example 2).

3.3. Complexity of the Verification Technique

We will measure the time complexity of our verification technique as: (i) the number of applications of transformation rules in Step 1 and (ii) the number of closed literals that are evaluated when executing the *Demo* program in Step 2.

Let us first consider Step 1 and let us measure the number of new predicate symbols, that is, the number of distinct blocks that can be generated during the *define-fold* procedure. Let $B = L_1 \wedge \ldots \wedge L_m$ be a block. Since the literals in B cannot be further unfolded, we have that, for $i = 1, \ldots, m$, either (a) L_i is the atom A_i or the negated atom $\neg A_i$, where A_i belongs to the set $\{notpath(X)\} \cup \{notpath([s|X]) \mid s \in \Sigma\} \cup \{sat(X, \psi) \mid \psi \text{ is a subformula of } \varphi\}$, or (b) L_i belongs to the set $\{\neg\, sat([s|X], e(\psi)) \mid s \in \Sigma \text{ and } \psi \text{ is a subformula of } \varphi\}$. We have also the following properties of B: (i) there is at most one atom in B of the form $notpath([s|X])$, and (ii) if in B there are two occurrences of terms of the form $[s_1|X]$ and $[s_2|X]$, then $s_1 = s_2$. By these properties the number of distinct blocks and, thus, the number of new predicate symbols is $\mathcal{O}(|\Sigma| \cdot 2^{|\varphi|})$. Moreover, the size of each block is $\mathcal{O}(|\varphi|)$, which is also the number of clauses introduced in the set *InDefs* for each new predicate symbol.

For each clause in *InDefs*, our transformation strategy performs one execution of the loop body, which starts off by applying the *unfold* procedure. This procedure applies the instantiation rule which generates $|\Sigma|$ clauses and, then, makes $\mathcal{O}(|\varphi|)$ unfolding steps for each instantiated clause. The total number of unfolding steps for each new predicate is, thus, $\mathcal{O}(|\Sigma| \cdot |\varphi|^2)$, which is also the number of the derived clauses. Since there are $\mathcal{O}(|\varphi|^2)$ clauses with same head, the number of subsumption steps is $\mathcal{O}(|\Sigma| \cdot |\varphi|^4)$ for each new predicate symbol. Thus, the total number of transformation rule applications in the *unfold* procedure is $\mathcal{O}(|\Sigma| \cdot |\varphi|^4)$. If we consider all new predicates symbols, we get $\mathcal{O}(|\Sigma|^2 \cdot |\varphi|^4 \cdot 2^{|\varphi|})$ applications of transformation rules.

The *define-fold* procedure performs at most one folding and one definition introduction for each block occurring in the body of a clause. The number of blocks in the body of a clause is $\mathcal{O}(|\varphi|)$ because each block has been introduced by unfolding an atom of the form $sat(X, e(\psi))$, where $e(\psi)$ is a subformula of φ. The number of folding steps is, thus, $\mathcal{O}(|\Sigma|^2 \cdot |\varphi|^5 \cdot 2^{|\varphi|})$, while the number of definition introductions is equal to the number of predicate symbols. Therefore, the total number of applications of transformation rules in Step 1 of our verification technique is $\mathcal{O}(|\Sigma|^2 \cdot |\varphi|^5 \cdot 2^{|\varphi|})$.

In Step 2, a logic programming system which uses tabled resolution evaluates every closed literal at most once. The proof of a closed literal requires $\mathcal{O}(|\Sigma|^2 \cdot 2^{|\varphi|})$ applications of proof rules. Therefore, we may conclude that the time complexity of our verification algorithm is $\mathcal{O}(|\Sigma|^2) \cdot 2^{\mathcal{O}(|\varphi|)}$.

In [8] an algorithm for CTL* model checking is provided whose time complexity is $\mathcal{O}(|\Sigma| + |\rho|) \cdot 2^{\mathcal{O}(|\varphi|)}$. Note that, since ρ is a total binary relation, we have $|\Sigma| \leq |\rho| \leq |\Sigma|^2$. Thus, in the case where $|\rho| = |\Sigma|^2$, the time complexity of our algorithm is the same of the best known algorithm for CTL* model checking. The case where the Kripke structure $\mathcal{K}$ represents a deterministic

transition system and, thus, $|\rho| = |\Sigma|$ is quite unfrequent in practice.

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 [17]. Techniques based on logic programming, constraint solving, abstract interpretation, and program transformation have been proposed for performing CTL model checking of finite and infinite state systems (see, for instance, [4, 6, 10, 12]). In this paper we have extended to CTL* model checking the transformational approach which was proposed for LTL model checking in [13].

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 truth in the perfect model is decidable. This subclass of programs properly extends the class of *linear recursive* ω-programs introduced in [13] and also the proof system presented here extends the one in [13]. (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. The transformation strategy presented in this paper is more elaborated than the one considered in [13] for the case of LTL properties. However, the worst case time complexity of the two strategies is the same.

Our transformation strategy can be understood as a specialization of the Encoding Program (see Definition 2.4) 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 [9] for a brief survey). Indeed, our transformation strategy performs folding 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 [8]. 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 features. (1) The specification of properties of reactive systems, the transformation of specifications into a monadic ω-programs, and the proofs of properties of monadic ω-programs are all expressed within the single framework of logic programming, while in the automata-theoretic approach one has to translate the temporal logic formalism into the automata-theoretic formalism. (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 easily be extended to the case of infinite state systems.

Issues that can be investigated in the future include: (i) the relationship between monadic ω-programs and alternating tree automata, (ii) the applicability of our transformational approach to other logics, such as the monadic second order logic of successors, and (iii) the experimental evaluation of the efficiency of our transformational approach by considering various test cases and comparing its performance in practice w.r.t. that of other model checking techniques.

16.

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] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1), 1996.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[4] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.

[5] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, 1986.

[6] 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.

[7] 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.

[8] O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *J. ACM*, 47(2):312–360, 2000.

[9] M. Leuschel. Logic program specialisation. In J. Hatcliff and P. Thiemann (Eds.) T. Mogensen, editors, *Partial Evaluation - Practice and Theory*, LNCS 1706, pages 155–188. Springer, 1998.

[10] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proc. of LOPSTR '99*, LNCS 1817, pages 63–82. Springer, 2000.

[11] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.

[12] U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. *Proc. of CL 2000*, LNAI 1861, pages 384–398. Springer, 2000.

[13] A. Pettorossi, M. Proietti, and V. Senni. Transformational verification of linear temporal logic. *24th Italian Conference on Computational Logic June 24-26, 2009, Ferrara, Italy (CILC '09)*. http://www.ing.unife.it/eventi/cilc09.

[14] 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.

[15] H. Przymusinska and T. C. Przymusinski. Weakly stratified logic programs. *Fundamenta Informaticae*, 13:51–65, 1990.

[16] 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.

[17] 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.

[18] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, and E. Johnson. The XSB System, Version 2.2., 2000.

[19] H. Seki, Unfold/Fold transformation of stratified programs. *Theoretical Computer Science*, 86, 1:107–139, 1991.

[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.