A Transformation Strategy for Verifying Logic Programs on Infinite Lists

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
maurizio.proietti@iasi.cnr.it

Abstract. We consider an extension of the class of logic programs, called *ω-programs*, that can be used to define predicates over *infinite lists*. The ω-programs allow us to specify properties of the infinite behaviour of reactive systems and, in general, properties of infinite sequences of events. The semantics of ω-programs is an extension of the perfect model semantics. We present a general methodology based on an extension of the unfold/fold transformation rules which can be used for verifying properties of ω-programs. Then we propose a strategy for the mechanical application of those rules and we demonstrate the power of that strategy by proving some properties of ω-regular languages and Büchi automata.

1 Introduction

The problem of specifying and verifying properties of *reactive systems*, such as communication protocols and concurrent systems, has received much attention over the past fifty years or so. The main peculiarity of reactive systems is that they perform nonterminating computations and, in order to specify and verify the properties of these computations, various formalisms dealing with infinite sequences of events have been proposed. Among these we would like to mention: (i) ω-languages [26], (ii) Büchi automata and other classes of finite automata on infinite sequences [28], and (iii) various temporal and modal logics (see [4] for a brief overview of these logics).

Also logic programming has been proposed as a formalism for specifying computations over infinite structures, such as infinite lists or infinite trees (see, for instance, [5,13,14,25]). One advantage of using logic programming languages is that they are general purpose languages and, together with a model-theoretic semantics, they also have an operational semantics. Thus, logic programs over infinite structures can be used for *specifying* infinite computations and, in fact, providing executable specifications for them. However, very few techniques which use logic programs over infinite structures have been proposed in the literature for *verifying* properties of infinite computations. We are aware only of a recent work which is based on coinductive logic programming [10], that is, a logic programming language whose semantics is defined in terms of greatest models.

In this paper we want to develop a methodology based on the familiar unfold/fold transformation rules [3,27] for reasoning about infinite structures and verifying properties of programs over such structures. In order to do so, we do not introduce a new programming language and, instead, we consider the simple extension of logic programming which was proposed in [18]. In that paper we introduced the class of ω-programs, which are logic programs that act on infinite lists (as well as finite terms) and admit the *perfect model* semantics (see [2] for a survey on negation in logic programming).

As indicated in [18], one can extend to ω-programs the classical unfold/fold transformation rules for locally stratified programs [8,16,22,23,24]. In this paper, by applying those extended rules, we will adapt to ω-programs the transformation-based methodology for verifying properties of programs which was first presented in [16].

That verification methodology consists of two steps. In the first step, starting from an ω-program which represents the reactive system under consideration and the property to be verified, we apply a strategy, called *TransfM*, for guiding the application of the transformation rules and we derive an ω-program which is *monadic* (see Definition 1). In the second step we apply a decision procedure for monadic ω-programs and we decide whether or not the given property holds.

We will demonstrate the power of our verification methodology through some examples. In particular, we will prove: (i) the containment between languages denoted by ω-regular expressions, and (ii) the non-emptiness of languages recognized by Büchi automata.

The paper is structured as follows. In Section 2 we introduce the class of ω-programs and we define the perfect model semantics for locally stratified ω-programs. In Section 3 we present the transformation rules and we establish their correctness, that is, the fact that they preserve the perfect model semantics. In Section 4 we present the transformation-based verification method, we introduce our transformation strategy *TransfM*, and in Section 5 we see it in action in some examples. Finally, in Section 6 we discuss related work in the area of program transformation and program verification.

2 Programs on Infinite Lists

Let us consider a first order language $\mathcal{L}_\omega$ given by a set *Var* of variables, a set *Fun* of function symbols, and a set *Pred* of predicate symbols. We assume that *Fun* includes: (i) a *finite*, *non-empty* set Σ of constants, (ii) the constructor $[\![\_|\_]\!]$ of the infinite lists of elements of Σ, and (iii) at least one constant not in Σ. Thus, $[\![s|t]\!]$ is an infinite list whose head is $s \in \Sigma$ and whose tail is the infinite list t. Let Σ^ω denote the set of the infinite lists of elements of Σ.

We assume that $\mathcal{L}_\omega$ is a typed language [13] with three basic types: (i) `fterm`, which is the type of the finite terms, (ii) `elem`, which is the type of the constants in Σ, and (iii) `ilist`, which is the type of the infinite lists of Σ^ω.

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 $\rightarrow$. The function symbol $[\![\_|\_]\!]$ has type $(\texttt{elem}\times\texttt{ilist})\rightarrow\texttt{ilist}$. A predicate symbol of arity $n\,(\geq 0)$ in *Pred* has type of the form $\tau_1 \times \cdots \times \tau_n$, where $\tau_1, \ldots, \tau_n \in \{\texttt{fterm}, \texttt{elem}, \texttt{ilist}\}$. An *ω-clause* γ is a formula of the form $A \leftarrow L_1 \wedge \ldots \wedge L_m$, with $m \geq 0$, where A is an atom and $L_1, \ldots, L_m$ are (positive or negative) literals, constructed as usual from symbols in the typed language $\mathcal{L}_\omega$, with the following extra condition: every predicate in γ has, among its arguments, *at most one* argument of type `ilist`. This condition makes it easier to prove the correctness of the positive and negative unfolding rules (see Section 3 for further details). An *ω-program* is a set of ω-clauses.

Given a term or a formula f, by $\mathit{vars}(f)$ we denote the set of variables occurring in f.

Let HU be the Herbrand universe constructed from the set $\mathit{Fun} - (\Sigma \cup \{[\![\_|\_]\!]\})$ of function symbols. An interpretation for our typed language $\mathcal{L}_\omega$, called an ω-interpretation, is a function I such that: (i) I assigns to the types `fterm`, `elem`, and `ilist`, respectively, the sets HU, Σ, and Σ^ω, (which by our assumptions are non-empty) (ii) I assigns to the function symbol $[\![\_|\_]\!]$, the function $[\![\_|\_]\!]_I$ such that, for any element $s \in \Sigma$ and infinite list $t \in \Sigma^\omega$, $[\![s|t]\!]_I$ is the infinite list $[\![s|t]\!]$, (iii) I is an Herbrand interpretation for all function symbols in $\mathit{Fun} - (\Sigma \cup \{[\![\_|\_]\!]\})$, and (iv) I assigns to every n-ary predicate $p \in \mathit{Pred}$ 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, if τ_i is either `fterm` or `elem` or `ilist`, respectively. We say that an ω-interpretation I is an *ω-model* of an ω-program P if for every clause $\gamma \in P$ we have that $I \models \forall X_1 \ldots \forall X_k\, \gamma$, where $\mathit{vars}(\gamma) = \{X_1, \ldots, X_k\}$.

A *valuation* is a function $v\colon \mathit{Var} \to \mathit{HU} \cup \Sigma \cup \Sigma^\omega$ such that: (i) if X has type `fterm` then $v(X) \in \mathit{HU}$, (ii) if X has type `elem` then $v(X) \in \Sigma$, and (iii) if X has type `ilist` then $v(X) \in \Sigma^\omega$. The valuation function v can be extended to any term t, or literal L, or conjunction B of literals, or clause γ, by making the function v act on the variables occurring in t, or L, or B, or γ.

We extend the notion of *Herbrand base* [13] to ω-programs by defining it to be the set $\mathcal{B}_\omega = \{p(v(X_1), \ldots, v(X_n)) \mid p$ is an n-ary predicate symbol and v is a valuation$\}$. Thus, any ω-interpretation can be identified with a subset of $\mathcal{B}_\omega$.

A *local stratification* is a function $\sigma\colon \mathcal{B}_\omega \to W$, where W is the set of countable ordinals. Given $A \in \mathcal{B}_\omega$, we define $\sigma(\neg A) = \sigma(A) + 1$. Given an ω-clause γ of the form $H \leftarrow L_1 \wedge \ldots \wedge L_m$ 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*.

A *level mapping* is a function $\ell : \mathit{Pred} \to \mathbb{N}$. 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 [13]. Clearly, every stratified ω-program is a locally stratified ω-program.

Similarly to the case of logic programs on finite terms, for every locally stratified ω-program P, we can construct a unique *perfect ω-model* (or *perfect model*, for short) denoted by $M(P)$ (see [2] for the case of logic programs on finite terms). Now we present an example of this construction.

Example 1. Let: (i) $\Sigma = \{a, b\}$ be the set of constants of type `elem`, (ii) S be a variable of type `elem`, and (iii) X be a variable of type `ilist`. Let p and q be predicates of type `ilist`. Let us consider the following ω-program P:

$$p(X) \leftarrow \neg q(X) \qquad\qquad q([\![b|X]\!]) \leftarrow \qquad\qquad q([\![a|X]\!]) \leftarrow q(X)$$

where: (i) $p(u)$ holds iff u is an infinite list of a's and (ii) $q(u)$ holds iff at least one b occurs in u. Program P is stratified w.r.t. the level mapping ℓ such that $\ell(q) = 0$ and $\ell(p) = 1$. The perfect model $M(P)$ is constructed by starting from the ground atoms of level 0 (i.e., those with predicate q). We have that, for all $u \in \{a, b\}^\omega$, $q(u) \in M(P)$ iff $u \in a^* b(a+b)^\omega$, that is, $q(u) \notin M(P)$ iff $u \in a^\omega$ (note that if $q(u) \in M(P)$ then there is at least one occurrence of b in u and, if $q(u) \notin M(P)$, then there are no occurrences of b in u, and thus, $u = a^\omega$). Then, we consider the ground atoms of level 1 (i.e., those with predicate p). For all $u \in \{a, b\}^\omega$, $p(u) \in M(P)$ iff $q(u) \notin M(P)$. Thus, $p(u) \in M(P)$ iff $u \in a^\omega$.

Let us now introduce a subclass of ω-programs, called *monadic ω-programs*, which enjoy decidability properties that will be used in the transformation-based verification method presented in Section 4.

Definition 1 (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 $\mathit{vars}(A_0) \not\supseteq \mathit{vars}(L_i)$, 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.

We denote by $\mathcal{F}$ the set of formulas F such that F is *either* (1) of the form p, where p is a 0-ary predicate symbol, *or* (2) 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 $\neg q_i(X)$. The following result has been presented in [17].

Theorem 1 (Decidability of Monadic ω-Programs). *There is a decision procedure MDec for the problem of checking, for any monadic ω-program P and formula F in $\mathcal{F}$, whether or not $M(P) \models F$ holds.*

3 Transformation Rules

Given an ω-program P_0, a *transformation sequence* is a sequence $P_0, \ldots, P_n$, with $n \geq 0$, of ω-programs constructed as follows. Suppose that we have constructed a sequence $P_0, \ldots, P_k$, for $0 \leq k \leq n-1$. Then, the next program P_{k+1}

in the sequence is derived from program P_k by applying one of the following transformation rules R1–R7.

First we have the *definition introduction* rule which allows us to introduce a new predicate definition.

R1. Definition Introduction. Let us consider m (≥ 1) clauses of the form:

$\delta_1 : \mathit{newp}(X_1, \ldots, X_d) \leftarrow B_1, \quad \ldots, \quad \delta_m : \mathit{newp}(X_1, \ldots, X_d) \leftarrow B_m$

noindent where: (i) *newp* is a predicate symbol not occurring in $\{P_0, \ldots, P_k\}$, (ii) $X_1, \ldots, X_d$ are distinct variables occurring in $\{B_1, \ldots, B_m\}$, (iii) none of the B_i's is the empty conjunction of literals, and (iv) every predicate symbol occurring in $\{B_1, \ldots, B_m\}$ also occurs in P_0. The set $\{\delta_1, \ldots, \delta_m\}$ of clauses is said to be the *definition* of *newp*.
By *definition introduction* from program P_k we derive the new program $P_{k+1} = P_k \cup \{\delta_1, \ldots, \delta_m\}$. For $n \geq 0$, Defs_n denotes the set of clauses introduced by the definition rule during the transformation sequence $P_0, \ldots, P_n$. In particular, $\mathit{Defs}_0 = \{\}$.

In the following *instantiation* rule we assume that the set of the constants of type `elem` in the language $\mathcal{L}_\omega$ is the finite set $\Sigma = \{s_1, \ldots, s_h\}$.

R2. Instantiation. Let γ: $H \leftarrow B$ be a clause in program P_k and X be a variable of type `ilist` occurring in γ. By *instantiation* of X in γ, we get the clauses:

γ_1: $(H \leftarrow B)\{X/[\![s_1|X]\!]\}, \quad \ldots, \quad \gamma_h$: $(H \leftarrow B)\{X/[\![s_h|X]\!]\}$

and we say that clauses $\gamma_1, \ldots, \gamma_h$ are *derived from* γ. From P_k we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\gamma_1, \ldots, \gamma_h\}$.

The *unfolding* rule consists in replacing an atom A occurring in the body of a clause by its definition in P_k. We present two unfolding rules: (1) the *positive unfolding*, and (2) the *negative unfolding*. They correspond, respectively, to the case where A or $\neg A$ occurs in the body of the clause to be unfolded.

R3. Positive Unfolding. Let $\gamma : \ H \leftarrow B_L \wedge A \wedge B_R$ be a clause in program P_k and let P'_k be a variant of P_k without variables in common with γ. Let

$\gamma_1 : \ K_1 \leftarrow B_1, \quad \ldots, \quad \gamma_m : \ K_m \leftarrow B_m \qquad (m \geq 0)$

be all clauses of program P'_k such that, for $i = 1, \ldots, m$, A is unifiable with K_i, with most general unifier ϑ_i.
By *unfolding* γ *w.r.t.* A we get the clauses $\eta_1, \ldots, \eta_m$, where for $i = 1, \ldots, m$, η_i is $(H \leftarrow B_L \wedge B_i \wedge B_R)\vartheta_i$, and we say that clauses $\eta_1, \ldots, \eta_m$ are *derived from* γ. From P_k we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_m\}$.

In rule R3, and also in the following rule R4, the most general unifier can be computed by using a unification algorithm for finite terms (see, for instance, [13]). Note that this is correct, even in the presence on infinite terms, because in any ω-program each predicate has at most one argument of type `ilist`. On the contrary, if predicates may have more than one argument of type `ilist`, in the unfolding rule it is necessary to use a unification algorithm for infinite structures [5]. For reasons of simplicity, we do not make that extension of the unfolding rule and we stick to our assumption that every predicate has at most one argument of type `ilist`.

The *existential variables* of a clause γ are the variables occurring in the body of γ and not in its head.

R4. Negative Unfolding. Let γ: $H \leftarrow B_L \wedge \neg A \wedge B_R$ be a clause in program P_k and let P'_k be a variant of P_k without variables in common with γ. Let

$\gamma_1 : \ K_1 \leftarrow B_1, \ \ \ldots, \ \ \gamma_m : \ K_m \leftarrow B_m \quad (m \geq 0)$

be all clauses of program P'_k, such that, for $i = 1, \ldots, m$, A is unifiable with K_i, with most general unifier ϑ_i. Assume that: (1) $A = K_1\vartheta_1 = \cdots = K_m\vartheta_m$, that is, for $i = 1, \ldots, m$, A is an instance of K_i, (2) for $i = 1, \ldots, m$, γ_i has no existential variables, and (3) from $\neg(B_1\vartheta_1 \vee \ldots \vee B_m\vartheta_m)$ we get a logically equivalent disjunction $D_1 \vee \ldots \vee D_r$ of conjunctions of literals, with $r \geq 0$, by first pushing $\neg$ inside and then pushing $\vee$ outside.
By *unfolding* γ *w.r.t.* $\neg A$ *using* P_k we get the clauses $\eta_1, \ldots, \eta_r$, where, for $i = 1, \ldots, r$, clause η_i is $H \leftarrow B_L \wedge D_i \wedge B_R$, and we say that clauses $\eta_1, \ldots, \eta_r$ are *derived from* γ. From P_k we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_r\}$.

The following *subsumption* rule allows us to remove from P_k a clause γ such that $M(P_k) = M(P_k - \{\gamma\})$.

R5. Subsumption. Let γ_1: $H \leftarrow$ be a clause in program P_k and let γ_2 in $P_k - \{\gamma_1\}$ be a variant of $(H \leftarrow B)\vartheta$, for some conjunction of literals B and substitution ϑ. Then, we say that γ_2 *is subsumed* by γ_1 and by *subsumption*, from P_k we derive the new program $P_{k+1} = P_k - \{\gamma_2\}$.

The *folding* rule consists in replacing instances of the bodies of the clauses that define an atom A by the corresponding instance of A. Similarly to the case of the unfolding rule, we have two folding rules: (1) *positive folding* and (2) *negative folding*. They correspond, respectively, to the case where folding is applied to positive or negative occurrences of literals.

R6. Positive Folding. Let γ be a clause in P_k and let Defs'_k be a variant of Defs_k without variables in common with γ. Let the definition of a predicate in Defs'_k consist of the clause $\delta : \ K \leftarrow B$, where B is a non-empty conjunction of literals. Suppose that there exists a substitution ϑ such that clause γ is of the form $H \leftarrow B_L \wedge B\vartheta \wedge B_R$ and, for every variable $X \in \mathit{vars}(B) - \mathit{vars}(K)$, the following conditions hold: (i) $X\vartheta$ is a variable not occurring in $\{H, B_L, B_R\}$, and (ii) $X\vartheta$ does not occur in the term $Y\vartheta$, for any variable Y occurring in B and different from X.
By *folding* γ *using* δ we get the clause η: $H \leftarrow B_L \wedge K\vartheta \wedge B_R$, and we say that clause η is *derived from* γ. From P_k we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

R7. Negative Folding. Let γ be a clause in P_k and let Defs'_k be a variant of Defs_k without variables in common with γ. Let the definition of a predicate in Defs'_k consist of the q clauses $\delta_1 : K \leftarrow L_1, \ldots, \delta_q : K \leftarrow L_q$ such that, for

$i = 1, \ldots, q$, L_i is a literal and δ_i has no existential variables. Suppose that there exists a substitution ϑ such that clause γ is of the form $H \leftarrow B_L \wedge (M_1 \wedge \ldots \wedge M_q)\vartheta \wedge B_R$, where, for $j = 1, \ldots, q$, if L_j is the negative literal $\neg A_j$ then M_j is A_j, and if L_j is the positive literal A_j then M_j is $\neg A_j$.
By *folding* γ *using* $\delta_1, \ldots, \delta_q$ we get the clause η: $H \leftarrow B_L \wedge \neg K\vartheta \wedge B_R$, and we say that clause η is *derived from* γ. From P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

In order to prove that the transformation rules R1–R7 are correct we now introduce the notion of correctness of a transformation sequence w.r.t. the perfect model semantics.

Definition 2 (Correctness of a Transformation Sequence). *Let P_0 be a locally stratified ω-program and $P_0, \ldots, P_n$, with $n \geq 0$, be a transformation sequence. We say that $P_0, \ldots, P_n$ is* correct *if* (i) *$P_0 \cup Defs_n$ and P_n are locally stratified ω-programs and* (ii) *$M(P_0 \cup Defs_n) = M(P_n)$.*

In order to guarantee the correctness of a transformation sequence $P_0, \ldots, P_n$ (see Theorem 2 below) we will require that the application of the transformation rules satisfy some suitable conditions that refer to the local stratification σ for P_0. We begin by providing the following definitions.

Definition 3 (σ-Maximal Atom). *Consider a clause γ: $H \leftarrow G$. An atom A in G is said to be σ-*maximal *if, for every valuation v and for every literal L in G, we have that $\sigma(v(A)) \geq \sigma(v(L))$.*

Definition 4 (σ-Tight Clause). *A clause δ: $H \leftarrow G$ is said to be σ-*tight *if there exists a σ-maximal atom A in G such that, for every valuation v, we have that $\sigma(v(H)) = \sigma(v(A))$.*

Definition 5 (Descendant Clause). *A clause η is said to be a* descendant *of a clause γ if either η is γ itself or there exists a clause δ such that η is derived from δ by using a rule in* $\{$R2, R3, R4, R6, R7$\}$, *and δ is a descendant of γ.*

Definition 6 (Admissible Transformation Sequence). *Let P_0 be a locally stratified ω-program and let σ be a local stratification for P_0. A transformation sequence $P_0, \ldots, P_n$, with $n \geq 0$, is said to be* admissible *if:*
(1) *every clause in $Defs_n$ is locally stratified w.r.t. σ,*
(2) *for $k = 0, \ldots, n-1$, if P_{k+1} is derived from P_k by positive folding of clause γ using clause δ, then:* (2.1) *δ is σ-tight and* either (2.2.i) *the head predicate of γ occurs in P_0,* or (2.2.ii) *γ is a descendant of a clause β in P_j, with $0 < j \leq k$, such that β has been derived by positive unfolding of a clause α in P_{j-1} w.r.t. an atom which is σ-maximal in the body of α and whose predicate occurs in P_0, and*
(3) *for $k = 0, \ldots, n$–1, if P_{k+1} is derived from P_k by applying the negative folding rule thereby deriving a clause η, then η is locally stratified w.r.t. σ.*

Note that Condition (1) can always be fulfilled because the predicate introduced in program P_{k+1} by rule R1 does not occur in any of the programs $P_0, \ldots, P_k$. Conditions (2) and (3) are technical conditions which are required for establishing Lemma 1 and Theorem 2. Unfortunately, those conditions cannot be checked in an algorithmic way for arbitrary programs and arbitrary local stratification functions. In particular, the program property of being locally stratified is undecidable. However, there are significant classes of programs, like, for instance, the stratified programs, where these conditions are decidable and easy to verify.

The following Lemma 1 and Theorem 2, whose proofs can be found in [19], show that: (i) when constructing an admissible transformation sequence $P_0, \ldots, P_n$, the application of the transformation rules preserves the local stratification σ for the initial program P_0 and, thus, all programs in the transformation sequence are locally stratified w.r.t. σ, and (ii) any admissible transformation sequence preserves the perfect model.

Lemma 1 (Preservation of Local Stratification). *Let P_0 be a locally stratified ω-program, σ be a local stratification for P_0, and $P_0, \ldots, P_n$ be an admissible transformation sequence. Then the programs $P_0 \cup Defs_n$, $P_1, \ldots, P_n$, are all locally stratified w.r.t. σ.*

Theorem 2 (Correctness of Admissible Transformation Sequences). *Every admissible transformation sequence is correct.*

The notion of admissible transformation sequence will be used in the following Section 4 to prove that the transformation strategy we propose is correct, that is, it generates only admissible transformation sequences.

4 Transformation Strategies for Verifying Properties of ω-Programs

In this section we present a general method for verifying properties of ω-programs. Our method is based on a strategy that guides the application of the transformation rules presented in Section 3, so that verification can be performed in an automatic way.

We assume we are given an ω-program P defining a unary predicate *prop* of type `ilist`, which specifies a property of interest, and we want to check whether or not $M(P) \models \exists X\, prop(X)$. Our verification method consists of two steps:

Step 1. Starting from P, by using the transformation rules of Section 3 according to a strategy, called *TransfM*, that we will present below, we derive a *monadic* ω-program T (see Definition 1 below), such that $M(P) \models \exists X\, prop(X)$ iff $M(T) \models \exists X\, prop(X)$.

Step 2. We apply to T the decision procedure *MDec* for monadic ω-programs (see Theorem 1) and we check whether or not $M(T) \models \exists X\, prop(X)$.

Note that there exists no strategy which always terminates and transforms a given ω-program into a monadic ω-program. Indeed, the problem of verifying whether or not, for an arbitrary ω-program P, $M(P) \models \exists X\ prop(X)$ holds, is undecidable (because ω-programs include locally stratified logic programs on finite terms), while as already mentioned the same problem for *monadic* ω-programs is decidable.

Now we introduce our transformation strategy, called *TransfM*, which transforms an ω-program into a monadic ω-program. Obviously, the strategy *TransfM* may not terminate due to the undecidability results we have mentioned above.

We leave it for future research the problem of determining suitable syntactic restrictions on the ω-programs which ensure termination of our strategy *TransfM*, thereby defining subclasses of ω-programs for which the verification problem is decidable. In this paper, we only show some examples where our transformation strategy terminates (see Section 5).

Our strategy is composed of two sub-strategies. The first sub-strategy, called *Specialize*, is an extension to arbitrary ω-programs of the strategy presented in [17] for specializing programs which encode the branching time logic CTL* w.r.t. a given CTL* formula. The *Specialize* sub-strategy performs a specialization of the given ω-program P w.r.t. the goal $prop(X)$ which encodes the property of interest. If the *Specialize* sub-strategy terminates, then it produces a *stratified* ω-program *SpecP*.

The second sub-strategy, called *Eliminate-Finite-Terms*, or *EFT* for short, eliminates from the program *SpecP* the arguments of type `fterm`. It is developed along the lines of the strategies in [16,20] for eliminating unnecessary variables. The *EFT* sub-strategy works bottom-up from the lowest stratum of the stratified program *SpecP* to its highest stratum with the objective of eliminating the arguments of type `fterm`. If the *EFT* sub-strategy terminates, then it returns a monadic ω-program T (which, as indicated in Definition 1, has no arguments of type `fterm`).

The Transformation Strategy *TransfM*

Input: An ω-program P.

Output: A monadic ω-program T such that
$M(P) \models \exists X\ prop(X)$ iff $M(T) \models \exists X\ prop(X)$.

Specialize$(P, SpecP)$;
Eliminate-Finite-Terms$(SpecP, T)$

We do not provide here the details of this strategy. We only present some examples of its application in the next section.

The following result, whose proof is omitted, follows from the fact that every transformation sequence generated by the strategy *TransfM* is admissible.

Theorem 3 (Correctness of the Transformation Strategy). *Every transformation sequence generated by the strategy TransfM is correct.*

5 Applications of the Verification Method

In this section we consider two examples where we apply our transformation-based verification methodology and, in particular, our strategy *TransfM*. For lack of space, we cannot show all steps of the transformation sequences generated by that strategy, and we will not show that every step indeed complies with the conditions which ensure admissibility.

Example 2 (Containment Between ω-Regular Languages).
In this first application of our verification methodology, we will consider regular sets of infinite words over a finite alphabet Σ [28]. These sets are denoted by ω-regular expressions whose definition is as follows: for any $a \in \Sigma$,

$e ::= a \mid e_1 e_2 \mid e_1{+}e_2 \mid e^*$ (regular expressions)

$f ::= e^\omega \mid e_1 e_2^\omega \mid f_1{+}f_2$ (ω-regular expressions)

Given a regular expression e denoting the language $\mathcal{L}(e) \subseteq \Sigma^*$, e^ω denotes the set $\{u_0 u_1 \ldots \in \Sigma^\omega \mid \text{for } i \geq 0, u_i \in \mathcal{L}(e)\}$.

Now, we introduce an ω-program, called P_f, which defines the predicate ω-*acc* such that, given an ω-regular expression f, and an infinite word u, $\omega\text{-}acc(f,u)$ holds iff $u \in \mathcal{L}(f)$. Any infinite word $a_0 a_1 \ldots \in \Sigma^\omega$ is represented by the infinite list $[\![a_0, a_1, \ldots]\!]$ of symbols in Σ. The ω-program P_f is made out of the following clauses:

1. $acc(E, [E]) \leftarrow symb(E)$
2. $acc(E_1 E_2, X) \leftarrow app(X_1, X_2, X) \wedge acc(E_1, X_1) \wedge acc(E_2, X_2)$
3. $acc(E_1{+}E_2, X) \leftarrow acc(E_1, X)$
4. $acc(E_1{+}E_2, X) \leftarrow acc(E_2, X)$
5. $acc(E^*, [\,]) \leftarrow$
6. $acc(E^*, X) \leftarrow app(X_1, X_2, X) \wedge acc(E, X_1) \wedge acc(E^*, X_2)$
7. $\omega\text{-}acc(F_1{+}F_2, X) \leftarrow \omega\text{-}acc(F_1, X)$
8. $\omega\text{-}acc(F_1{+}F_2, X) \leftarrow \omega\text{-}acc(F_2, X)$
9. $\omega\text{-}acc(E^\omega, X) \leftarrow \neg\, new_1(E, X)$
10. $\omega\text{-}acc(E_1 E_2^\omega, X) \leftarrow \omega\text{-}app(X_1, X_2, X) \wedge acc(E_1, X_1) \wedge \omega\text{-}acc(E_2^\omega, X_2)$
11. $new_1(E, X) \leftarrow nat(M) \wedge \neg\, new_2(E, M, X)$
12. $new_2(E, M, X) \leftarrow geq(N, M) \wedge prefix(X, N, V) \wedge acc(E^*, V)$
13. $geq(N, 0) \leftarrow$
14. $geq(s(N), s(M)) \leftarrow geq(N, M)$
15. $nat(0) \leftarrow$
16. $nat(s(N)) \leftarrow nat(N)$
17. $prefix(X, 0, [\,]) \leftarrow$
18. $prefix([\![S|X]\!], s(N), [S|Y]) \leftarrow prefix(X, N, Y)$
19. $\omega\text{-}app([\,], Y, Y) \leftarrow$
20. $\omega\text{-}app([S|X], Y, [\![S|Z]\!]) \leftarrow \omega\text{-}app(X, Y, Z)$
21. $app([\,], Y, Y) \leftarrow$
22. $app([S|X], Y, [S|Z]) \leftarrow app(X, Y, Z)$

together with the clauses defining the predicate *symb*, where $symb(a)$ holds iff $a \in \Sigma$. Note that: (i) $prefix(X, N, Y)$ holds iff Y is the list of the N (≥ 0) leftmost symbols of the infinite list X, and (ii) $\omega\text{-}app(X, Y, Z)$ holds iff the concatenation of the finite list X and the infinite list Y is the infinite list Z. Clauses 1–6 stipulate that, for any finite word u and regular expression e, $acc(e, u)$ holds iff $u \in \mathcal{L}(e)$. Analogously, clauses 7–12 stipulate that, for any infinite word u and ω-regular expression f, $\omega\text{-}acc(f, u)$ holds iff $u \in \mathcal{L}(f)$. In particular, clauses 9, 11, and 12 correspond to the following definition:

$$\omega\text{-}acc(E^\omega, X) \equiv_{def} \forall N(N \geq 0 \rightarrow \exists M \exists V(M \geq N \wedge prefix(X, M, V) \wedge acc(E^*, V)))$$

The program P_f is stratified.

Now, let us consider the ω-regular expressions $f_1 \equiv_{def} a^\omega$ and $f_2 \equiv_{def} (b^*a)^\omega$. The following two clauses:

23. $expr_1(X) \leftarrow \omega\text{-}acc(a^\omega, X)$ 24. $expr_2(X) \leftarrow \omega\text{-}acc((b^*a)^\omega, X)$

together with program P_f, define the predicates $expr_1$ and $expr_2$ such that, for every infinite word u, $expr_1(u)$ holds iff $u \in \mathcal{L}(f_1)$ and $expr_2(u)$ holds iff $u \in \mathcal{L}(f_2)$. Now, we introduce the predicate *prop* defined by the following clause:

25. $prop(X) \leftarrow expr_1(X) \wedge \neg\, expr_2(X)$

We have that $\mathcal{L}(f_1) \subseteq \mathcal{L}(f_2)$ iff $M(P_f \cup \{23, 24, 25\}) \not\models \exists X\, prop(X)$.

In order to check whether or not $\mathcal{L}(f_1) \subseteq \mathcal{L}(f_2)$, we proceed in two steps as indicated in Section 4. In the first step we apply our strategy *TransfM* which, in turn, consists of the application of the *Specialize* and the *EFT* sub-strategies. The *Specialize* sub-strategy takes as input the ω-program $P_f \cup \{23, 24, 25\}$ and returns the following specialized program *SpecP*:

25. $prop(X) \leftarrow expr_1(X) \wedge \neg\, expr_2(X)$
26. $expr_1(X) \leftarrow \neg\, new_1(X)$
27. $new_1(X) \leftarrow nat(Y) \wedge \neg\, new_2(X, Y)$
28. $new_2(X, Y) \leftarrow geq(Z, Y) \wedge prefix(X, Z, W) \wedge new_3(W)$
29. $new_3([\,]) \leftarrow$
30. $new_3([a|X]) \leftarrow new_3(X)$
31. $expr_2(X) \leftarrow \neg\, new_4(X)$
32. $new_4(X) \leftarrow nat(Y) \wedge \neg\, new_5(X, Y)$
33. $new_5(X, Y) \leftarrow geq(Z, Y) \wedge prefix(X, Z, W) \wedge new_6(W)$
34. $new_6([\,]) \leftarrow$
35. $new_6(X) \leftarrow app(Y, Z, X) \wedge new_7(Y) \wedge new_6(Z)$
36. $new_7(X) \leftarrow new_8(X, Y) \wedge new_9(Y)$
37. $new_8([a], [\,]) \leftarrow$
38. $new_8([X|Y], [X|Z]) \leftarrow new_8(Y, Z)$
39. $new_9([\,]) \leftarrow$
40. $new_9([b|X]) \leftarrow new_9(X)$

The *EFT* sub-strategy takes as input the program *SpecP* and returns the following monadic ω-program T:

41. $prop([\![a|X]\!]) \leftarrow \neg new_{10}(X) \wedge new_{11}(X)$
42. $new_{10}([\![a|X]\!]) \leftarrow new_{10}(X)$
43. $new_{10}([\![b|X]\!]) \leftarrow$
44. $new_{11}([\![a|X]\!]) \leftarrow new_{11}(X)$
46. $new_{12}([\![a|X]\!]) \leftarrow new_{11}(X)$
47. $new_{12}([\![b|X]\!]) \leftarrow new_{12}(X)$
48. $new_{12}([\![b|X]\!]) \leftarrow \neg\, new_{13}(X)$
49. $new_{13}([\![a|X]\!]) \leftarrow$

45. $new_{11}([\![b|X]\!]) \leftarrow new_{12}(X)$ 50. $new_{13}([\![b|X]\!]) \leftarrow new_{13}(X)$

The predicate symbols $new_1, \ldots, new_{13}$ are new symbols introduced by the strategy *TransfM*. By using the *MDec* decision procedure we have that $M(T) \not\models \exists X\, prop(X)$ and, thus, $\mathcal{L}(f_1) \subseteq \mathcal{L}(f_2)$.

Example 3 (Non-Emptiness of Languages Accepted by Büchi Automata).
In the second application of our verification methodology, we will consider *Büchi automata*, which are finite automata acting on infinite words [28], and we will present a method for checking whether or not the language they accept is empty. It is well known that this verification problem has important applications in the area of model checking (see, for instance, [4]).

A *Büchi automaton* $\mathcal{A}$ is a nondeterministic finite automaton $\langle \Sigma, Q, q_0, \delta, F\rangle$, where, as usual, Σ is the input alphabet, Q is the set of states, q_0 is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and F is the set of final states. A *run* of the automaton $\mathcal{A}$ on an infinite input word $u = a_0\, a_1 \ldots \in \Sigma^\omega$ is an infinite sequence $\rho = \rho_0\, \rho_1 \ldots \in Q^\omega$ of states such that ρ_0 is the initial state q_0 and, for all $n \geq 0$, $\langle \rho_n, a_n, \rho_{n+1}\rangle \in \delta$. Let $\mathit{Inf}(\rho)$ denote the set of states that occur infinitely often in the infinite sequence ρ of states. An infinite word $u \in \Sigma^\omega$ is *accepted* by $\mathcal{A}$ if there exists a run ρ of $\mathcal{A}$ on u, called an *accepting run*, such that $\mathit{Inf}(\rho) \cap F \neq \emptyset$ or, equivalently, if there is no state ρ_m in ρ such that all states ρ_n, with $n \geq m$, are not final. The *language accepted* by $\mathcal{A}$ is the subset of Σ^ω, denoted $\mathcal{L}(\mathcal{A})$, of the infinite words accepted by $\mathcal{A}$. In order to check whether or not the language $\mathcal{L}(\mathcal{A})$ is empty, we construct an ω-program which defines a predicate *prop* such that:

(α) $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff there exist a word u and an accepting run X of $\mathcal{A}$ on u
iff $\exists X\, prop(X)$

The predicate *prop* is defined by the following formulas:

(1) $prop(X) \equiv_{def} run(X) \wedge \neg\, rejecting(X)$

(2) $run(X) \equiv_{def} \exists S\, (occ(0, X, S) \wedge initial(S)) \wedge$
$\forall N\, \forall S_1\, \forall S_2\, (nat(N) \wedge occ(N, X, S_1) \wedge occ(s(N), X, S_2) \rightarrow \exists A\, tr(S_1, A, S_2)))$

(3) $rejecting(X) \equiv_{def} \exists N (nat(N) \wedge \forall M \forall S (geq(M, N) \wedge occ(M, X, S) \rightarrow \neg final(S)))$

where, for all $n \geq 0$, for all $\rho = \rho_0\, \rho_1 \ldots \in Q^\omega$, for all $q, q_1, q_2 \in Q$, and for all $a \in \Sigma$, $occ(s^n(0), \rho, q)$ iff $\rho_n = q$, $initial(q)$ iff $q = q_0$, $nat(s^n(0))$ iff $n \geq 0$, $tr(q_1, a, q_2)$ iff $\langle q_1, a, q_2\rangle \in \delta$, $geq(s^n(0), s^m(0))$ iff $n \geq m$, and $final(q)$ iff $q \in F$.

By (α) above, $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff there exists an infinite sequence $\rho = \rho_0\, \rho_1 \ldots \in Q^\omega$ of states such that: (i) ρ_0 is the initial state q_0 and, for $n \geq 0$, $\langle \rho_n, a, \rho_{n+1}\rangle \in \delta$, for some $a \in \Sigma$ (see (2)), and (ii) there is no state ρ_m in ρ such that, for all $n \geq m$, $\rho_n \notin F$ (see (3)).

Now we introduce an ω-program $P_{\mathcal{A}}$ defining the predicates *prop*, *run*, *rejecting*, *nat*, *occ*, and *geq*. In particular, clause 1 corresponds to formula (1), clauses 2–4 correspond to formula (2), and clauses 5 and 6 correspond to formula (3). (Actually, clauses 1–6 can be derived from formulas (1)–(3) by applying the Lloyd-Topor transformation [13].) In program $P_{\mathcal{A}}$ any infinite sequence $\rho_0 \rho_1 \ldots$ of states is represented by the infinite list $[\![\rho_0, \rho_1, \ldots]\!]$ of constants.

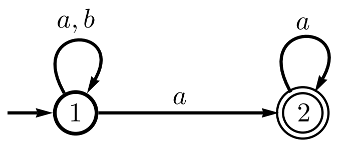
Given a Büchi automaton $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, F\rangle$, the encoding ω-program $P_{\mathcal{A}}$ consists of the following clauses (independent of $\mathcal{A}$):

1. $\mathit{prop}(X) \leftarrow \mathit{run}(X) \wedge \neg\, \mathit{rejecting}(X)$
2. $\mathit{run}(X) \leftarrow \mathit{occ}(0, X, S) \wedge \mathit{initial}(S) \wedge \neg\, \mathit{not\_a\_run}(X)$
3. $\mathit{not\_a\_run}(X) \leftarrow \mathit{nat}(N) \wedge \mathit{occ}(N,X,S_1) \wedge \mathit{occ}(s(N),X,S_2) \wedge \neg\, \mathit{exists\_tr}(S_1,S_2)$
4. $\mathit{exists\_tr}(S_1, S_2) \leftarrow \mathit{tr}(S_1, A, S_2)$
5. $\mathit{rejecting}(X) \leftarrow \mathit{nat}(N) \wedge \neg\, \mathit{exists\_final}(N, X)$
6. $\mathit{exists\_final}(M, X) \leftarrow \mathit{geq}(N, M) \wedge \mathit{occ}(N, X, S) \wedge \mathit{final}(S)$
7. $\mathit{nat}(0) \leftarrow$
8. $\mathit{nat}(s(N)) \leftarrow \mathit{nat}(N)$
9. $\mathit{occ}(0, \llbracket S|X \rrbracket, S) \leftarrow$
10. $\mathit{occ}(s(N), \llbracket S|X \rrbracket, R) \leftarrow \mathit{occ}(N, X, R)$
11. $\mathit{geq}(N, 0) \leftarrow$
12. $\mathit{geq}(s(N), s(M)) \leftarrow \mathit{geq}(N, M)$

together with the clauses (depending on $\mathcal{A}$) which define the predicates *initial*, *tr*, and *final*, where: (i) $\mathit{initial}(s)$ holds iff s is q_0, (ii) for all states $s_1, s_2 \in Q$ and all symbols $a \in \Sigma$, $\mathit{tr}(s_1,a,s_2)$ holds iff $\langle s_1,a,s_2\rangle \in \delta$, and (iii) $\mathit{final}(s)$ holds iff $s \in F$.

Now, let us consider a Büchi automaton $\mathcal{A}$ such that:

$\Sigma = \{a, b\}$, $Q = \{1, 2\}$, $q_0 = 1$, $\delta = \{\langle 1, a, 1\rangle, \langle 1, b, 1\rangle, \langle 1, a, 2\rangle, \langle 2, a, 2\rangle\}$, $F = \{2\}$ which can be represented by the following graph:



For this automaton $\mathcal{A}$, program $P_{\mathcal{A}}$ consists of clauses 1–12 and the following clauses 13–18 which encode the initial state (clause 13), the transition relation (clauses 14–17), and the final state of $\mathcal{A}$ (clause 18):

13. $\mathit{initial}(1) \leftarrow$ 14. $\mathit{tr}(1, a, 1) \leftarrow$ 15. $\mathit{tr}(1, b, 1) \leftarrow$
16. $\mathit{tr}(1, a, 2) \leftarrow$ 17. $\mathit{tr}(2, a, 2) \leftarrow$ 18. $\mathit{final}(2) \leftarrow$

Program $P_{\mathcal{A}}$ is stratified.

In order to check whether or not $\mathcal{L}(\mathcal{A}) = \emptyset$ we proceed, again, in two steps. The first step consists in applying our strategy *TransfM* to program $P_{\mathcal{A}}$. In this example *TransfM* terminates and returns the monadic ω-program T which consists of the following clauses 19–33:

19. $\mathit{prop}(\llbracket 1|X \rrbracket) \leftarrow \neg\, \mathit{not\_a\_run}(X) \wedge \mathit{new}_1(X) \wedge \neg\, \mathit{rejecting}(X)$
20. $\mathit{not\_a\_run}(\llbracket 1|X \rrbracket) \leftarrow \mathit{not\_a\_run}(X)$
21. $\mathit{not\_a\_run}(\llbracket 2|X \rrbracket) \leftarrow \mathit{new}_2(X)$
22. $\mathit{not\_a\_run}(\llbracket 2|X \rrbracket) \leftarrow \mathit{not\_a\_run}(X)$
23. $\mathit{rejecting}(\llbracket 1|X \rrbracket) \leftarrow \neg\, \mathit{new}_1(X)$
24. $\mathit{rejecting}(\llbracket 1|X \rrbracket) \leftarrow \mathit{rejecting}(X)$
25. $\mathit{rejecting}(\llbracket 2|X \rrbracket) \leftarrow \mathit{rejecting}(X)$
26. $\mathit{new}_1(\llbracket 2|X \rrbracket) \leftarrow$

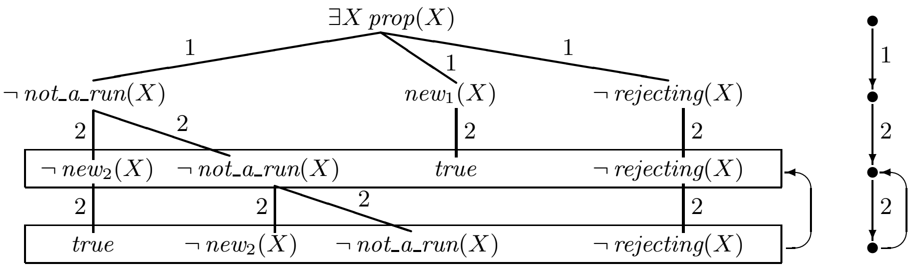


Fig. 1. Proof of $\exists X\, prop(X)$ w.r.t. the monadic ω-program T. On the right we have shown the infinite loop and the associated accepting run 122^ω on $\mathcal{A}$ (that is, 12^ω).

27. $new_1([\![1|X]\!]) \leftarrow new_1(X)$
28. $new_2([\![1|X]\!]) \leftarrow$

In the second step of our verification methodology, we check whether or not $\exists X\, prop(X)$ holds in $M(T)$ by applying the proof method of [17], that is, the decision procedure *MDec*. We construct the tree depicted in Figure 1, where the literals occurring in the two lowest levels are the same (see the two rectangles) and, thus, we have detected an infinite loop. According to conditions given in Definition 6 of [17], this tree is a proof of $\exists X\, prop(X)$. The run $\rho = 12^\omega$ is a witness for X and corresponds to the accepted word a^ω. Thus, $\mathcal{L}(\mathcal{A}) \neq \emptyset$.

6 Related Work and Conclusions

There have been various proposals for extending logic programming languages to infinite structures (see, for instance, [5,13,14,25]). These languages introduce new concepts to provide the semantics of infinite structures, such as *complete Herbrand interpretations*, *rational trees*, and *greatest models*. Also the operational semantics of these languages extends SLDNF-resolution by means of equational reasoning and new inference rules, such as the *coinductive hypothesis* rule.

On the contrary, the semantics of ω-programs we consider in this paper is very close to the usual perfect model semantics for logic programs on finite terms and we do not define any new operational semantics. The main objective of this paper is *not* to provide a new model for computing over infinite structures, but to present a methodology, based on unfold/fold transformation rules, for reasoning about such structures and proving their properties, and a strategy for guiding the application of these rules.

Very little work has been done for applying transformation techniques to logic languages that specify the (possible infinite) computations of reactive systems. Notable exceptions are [29] and [7], where the unfold/fold transformation rules have been studied in the context of *guarded Horn clauses* (GHC) and *concurrent constraint programs* (CCP). However, GHC and CCP programs are *definite* programs and do not manipulate terms denoting infinite lists. Moreover, these transformation rules have not been applied for proving program properties.

The transformation rules we considered in this paper are those presented in [18] which extend to ω-programs the rules for general programs proposed in [8,16,22,23,24]. The main focus of the present paper has been the proposal of a novel transformation strategy which is an extension of those presented in [16,17,20]. The effectiveness of the proposed strategy has been demonstrated through the verification of properties of ω-regular languages (see Example 2) and the infinite behavior of Büchi automata (see Example 3). These examples both refer to finite state systems. However, the verification methodology based on transformations we have considered in this paper, is very general and it can also be applied to the proof of properties of *infinite state* systems and, thus, it goes beyond the capabilities of finite state model checkers.

Many other papers use logic programming, possibly with constraints, for specifying and verifying properties of finite or infinite state reactive systems (see, for instance, [1,6,9,11,12,15,21]), but they do not consider terms which explicitly represent infinite structures. As we have seen in Examples 2 and 3, infinite lists are very convenient for specifying those properties and the use of infinite lists avoids ingenious encodings which would have been necessary otherwise.

There are some directions in which the transformational approach used in this paper can be enhanced. Among them we would like to mention: (i) the extension of the transformation rules and the verification method to other infinite structures, such as the infinite trees, and (ii) the use of the transformational approach for the synthesis of reactive systems with infinite behavior starting from their logical specifications.

7 Acknowledgements

We thank Hirohisa Seki for stimulating conversations on the topics of this paper. We also thank the anonymous referees for their constructive comments.

References

1. M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8(3):277–295, 1989.
2. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
3. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
4. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
5. A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 231–251. Academic Press, 1982.
6. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
7. S. Etalle, M. Gabbrielli, and M. C. Meo. Transformations of CCP programs. *ACM Transactions on Programming Languages and Systems*, 23(3):304–395, 2001.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 292–340. Springer-Verlag, 2004.
9. L. Fribourg and H. Olsén. A decompositional approach for computing least fixed-points of Datalog programs with Z-counters. *Constraints*, 2(3/4):305–335, 1997.

10. G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In Verónica Dahl and Ilkka Niemelä, editors, *23rd International Conference on Logic Programming, ICLP 2007, Porto, Portugal, September 8-13, 2007*, Lecture Notes in Computer Science 4670, pages 27–44. Springer, 2007.
11. J. Jaffar, A. E. Santosa, and R. Voicu. A CLP proof method for timed automata. In J. Anderson and J. Sztipanovits, editors, *The 25th IEEE International Real-Time Systems Symposium*, pages 175–186. IEEE Computer Society, 2004.
12. 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.
13. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second Edition.
14. R. Min and G. Gupta. Coinductive logic programming with negation. In D. De Schreye, editor, *Proceedings of the 19th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR '09), Coimbra, Portugal, September 9-11, 2009*, Lecture Notes in Computer Science 6037, pages 97–112. Springer-Verlag, 2010.
15. 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.
16. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. 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 613–628. Springer-Verlag, 2000.
17. A. Pettorossi, M. Proietti, and V. Senni. Deciding full branching time logic by program transformation. In D. De Schreye, editor, *Proceedings of the 19th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR '09), Coimbra, Portugal, September 9-11, 2009*, Lecture Notes in Computer Science 6037, pages 5–21. Springer-Verlag, 2010.
18. A. Pettorossi, M. Proietti, and V. Senni. Transformations of logic programs on infinite lists. *Theory and Practice of Logic Programming, Special Issue on the 26th International Conference on Logic Programming (ICLP 2010), Edinburgh, Scotland, UK* , 2010. To appear.
19. A. Pettorossi, M. Proietti, and V. Senni. Transformations of logic programs on infinite lists. Technical Report 10-04, IASI-CNR, Roma, Italy, 2010.
20. 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.
21. 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.
22. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. *International Journal on Foundations of Computer Science*, 13(3):387–403, 2002.

23. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
24. H. Seki. On inductive and coinductive proofs via unfold/fold transformations. In D. De Schreye, editor, *Proceedings of the 19th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR '09), Coimbra, Portugal, September 9-11, 2009*, Lecture Notes in Computer Science 6037, pages 82–96. Springer-Verlag, 2010.
25. L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In S. Etalle and M. Truszczyński, editors, *22nd International Conference on Logic Programming, ICLP 2006, Seattle, WA, USA, August 17-20, 2006*, Lecture Notes in Computer Science 4079, pages 330–345. Springer, 2006.
26. L. Staiger. ω-Languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 339–387. Springer, Berlin, 1997.
27. 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.
28. W. Thomas. Automata on infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 135–191. Elsevier, Amsterdam, 1990.
29. K. Ueda and K. Furukawa. Transformation rules for GHC programs. In *Proceedings International Conference on Fifth Generation Computer Systems, ICOT, Tokyo, Japan*, pages 582–591, 1988.