Perfect Model Checking*
via Unfold/Fold Transformations

Alberto Pettorossi[1], Maurizio Proietti[2]
(1) DISP, University of Roma Tor Vergata, Roma, Italy
(2) IASI-CNR, Roma, Italy
{adp,proietti}@iasi.rm.cnr.it

Abstract

We show how program transformation rules and strategies may be used for proving the satisfiability of first order formulas in some classes of models. In particular, we propose a technique for showing that a closed first order formula φ holds in the perfect model $M(P)$ of a logic program P with locally stratified negation. This property is denoted by $M(P) \models \varphi$.

For this purpose we consider a new version of the unfold/fold transformation rules and we show that this version preserves the perfect model semantics. Our proof method, called *unfold/fold proof method*, shows $M(P) \models \varphi$ by: (i) introducing a new predicate symbol f and constructing a conjunction $F(f,\varphi)$ of clauses such that $M(P) \models \varphi$ iff $M(P \wedge F(f,\varphi)) \models f$, and then (ii) transforming the program $P \wedge F(f,\varphi)$ into a new program of the form $Q \wedge f$, for some conjunction Q of clauses.

We also present a strategy for applying our unfold/fold rules in a semi-automatic way. Our strategy may or may not terminate, depending on the initial program P and formula φ. However, we identify some classes of programs such that, for each closed formula φ, our strategy terminates either for φ or $\neg\varphi$. Thus, our strategy is a decision procedure for programs and formulas in the given classes.

1 Introduction

One of the main motivations of this paper is to better understand the relationship between unfold/fold program transformation [4, 27] and theorem proving. It is usually recognized that folding steps during program transformation correspond to applications of inductive hypotheses during proofs by induction, and goal replacements correspond to lemma applications. Some transformational techniques for proving equivalences of functional and logic programs are presented in [8, 16] and [20, 24], respectively.

In this paper we extend these techniques by introducing a method for proving that a closed first order formula φ holds in the perfect model [22] of a locally stratified logic program P, written as $M(P) \models \varphi$. Perfect models are the usual intended semantics for logic programs with locally stratified negation, and for those programs all major approaches to the semantics of negation coincide. Indeed, a locally stratified program has a unique perfect model which is equal to its unique stable model, and also equal to its total well-founded model (see [1] for a survey on the semantics of negation in logic programming).

Our proof method for showing that $M(P) \models \varphi$ holds, consists of two steps:

Step 1: we use a variant of the Lloyd-Topor transformation [18] for transforming a statement of the form: $f \leftarrow \varphi$ where f is a new predicate symbol, into a conjunction $F(f,\varphi)$ of clauses, possibly with negated atoms in their bodies, such that $P \wedge F(f,\varphi)$ is locally stratified and $M(P) \models \varphi$ iff $M(P \wedge F(f,\varphi)) \models f$, and

* *Technical Report R.513, IASI-CNR, Rome, Italy, November 1999 (to appear in the Proceedings of: First International Conference on Computational Logic, London, July 24-28, 2000).*

Step 2: we show that $M(P \wedge F(f,\varphi)) \models f$ holds by applying semantics preserving transformation rules and deriving from $P \wedge F(f,\varphi)$ a new program of the form: $Q \wedge f$.

Indeed, we have that $M(Q \wedge f) \models f$ holds, and since the transformation rules are semantics preserving, at the end of Step 2 we have that $M(P \wedge F(f,\varphi)) \models f$ iff $M(Q \wedge f) \models f$.

Since f is false in the perfect model of any program which has no clause with head f, our method can also be used to show that $M(P) \not\models \varphi$, that is, $M(P) \models \neg\varphi$, by deriving from $P \wedge F(f,\varphi)$ a new program R, such that f does not occur in the head of any clause in R.

We illustrate our transformational proof method using the following example where we prove a property of a given transition system.

Example. [*Semaphore*] Consider the following program P, where x and y are variables, and $s \ldots s0$ with n occurrences of the successor function s, denotes the natural number n:

1. $\mathit{down}(sx) \leftarrow \neg \mathit{down}(x)$
2. $\mathit{up}(0,0)$
3. $\mathit{up}(sx,0) \leftarrow \mathit{down}(x)$
4. $\mathit{up}(sx,sy) \leftarrow \mathit{up}(sx,y),\ x>y$

Program P describes a semaphore which as time progresses, alternates between the states *up* and *down*. When the semaphore goes *up* for the n-th time, it stays *up* for a period of $2n$ time-units, and when it goes *down*, it stays *down* for one time-unit only (see the following table).

up :	0,0		2,0	2,1		4,0	4,1	4,2	4,3		6,0	...	6,5		8,0	...
down :		1			3					5				7		
time :	0	1	2	3	4	5	6	7	8	9	10	...	15	16	17	...

We want to prove the following Property (A) of the semaphore:

(A) $\forall x,y\,(x>y,\ \mathit{up}(x,y) \rightarrow \mathit{up}(ssx,0))$

which states that if the semaphore is *up* then it will be *up* again in the future. We start from the statement: $f \leftarrow (\forall x,y\,(x > y,\ \mathit{up}(x,y)) \rightarrow \mathit{up}(ssx,0))$. By applying our LT transformation (see Section 3) which is a variant of the Lloyd-Topor transformation, we get the clauses:

5. $f \leftarrow \neg g$
6. $g \leftarrow x>y,\ \mathit{up}(x,y),\ \neg \mathit{up}(ssx,0)$

This concludes Step 1 of our proof method.

Step 2 is realized by applying the following semantics preserving transformation rules: definition introduction, positive and negative unfolding, folding, tautology, clause deletion, and goal replacement, which are extensions of the familiar rules presented in [25, 27] (see Section 4 for the definitions of our rules).

In this Semaphore Example we introduce the definitions: (i) $h \leftarrow x>y,\ \mathit{up}(sx,y),\ \neg \mathit{up}(sssx,0)$ and (ii) $k(n) \leftarrow x > z,\ \mathit{plus}(y,n,z),\ \mathit{up}(sx,y),\ \neg \mathit{up}(sssx,0)$. By positive and negative unfolding, folding, and goal replacement, we get the following program:

$$Q:\ f \leftarrow \neg g,\ \ g \leftarrow h,\ \ h \leftarrow k(s0),\ \ k(n) \leftarrow k(sn)$$

The details of the derivation are given in Appendix 1.

Now, since k is a useless predicate being defined by the recursive clause $k(n) \leftarrow k(sn)$ only, without base case, we may apply the clause deletion rule (see rule R6 in Section 4) and we get the program

$$R:\ f \leftarrow \neg g,\ \ g \leftarrow h,\ \ h \leftarrow k(s0)$$

By unfolding clause $h \leftarrow k(s0)$ w.r.t. $k(s0)$ we delete that clause, because the definition of the predicate k in program R is empty. Then, by unfolding $g \leftarrow h$ w.r.t. h we also delete that clause, because the definition of h has been deleted in the previous unfolding step. Thus, the derived program is the clause $f \leftarrow \neg g$ only. Finally, by unfolding $f \leftarrow \neg g$ w.r.t. $\neg g$ we get f, because the definition of g in the derived program is empty. This completes our transformational proof of Property (A) we wanted to show. $\Box$

Notice that our proof of Property (A) is based on the crucial step of deleting the clause $k(n) \leftarrow k(sn)$, and this step is correct because in the perfect model of program R there are no atoms with predicate k.

The tight correspondence between program transformation and theorem proving can also be exploited to turn program transformation strategies into proof strategies. In particular, at Step 2 of our proof method, in order to direct the transformation rules, we may use the so called UFS strategy which is an enhancement of a strategy introduced in the case of definite logic programs for deriving programs without existential variables [21]. When our UFS strategy terminates, it derives from $P \wedge F(f, \varphi)$ *either* (i) a new program of the form $Q \wedge f$, in which case $M(P) \models \varphi$, *or* (ii) a program where no clause has head predicate f, in which case $M(P) \not\models \varphi$. We will show that the UFS strategy indeed terminates for some classes of properties and some classes of logic programs with locally stratified negation, and thus, it can be used for deciding whether or not $M(P) \models \varphi$ holds when φ and P belong to those classes.

The plan of the paper is as follows. In Section 3 we present Step 1 of our proof method by introducing a variant of the Lloyd-Topor transformation technique and we show that it preserves the perfect model semantics. The original Lloyd and Topor's correctness result w.r.t. the completion semantics [18] is often not suitable for proving properties of perfect models. In Section 4 we present the transformation rules, including the Definition rule, the Positive and Negative Unfolding rule (that is, unfolding of positive and negative literals), the Folding rule, the Tautology rule, the Clause Deletion rule, and the Goal Replacement rule. We show that all these rules preserve the perfect model semantics, while previous sets of rules proposed by Seki [25] include the Definition, Positive Unfolding and Folding rules only.

In Section 5 we present our UFS strategy for locally stratified programs, and in Section 6 we define some classes of properties and programs for which a deterministic version of the UFS strategy terminates. We show that these classes are non trivial by proving that both the equivalence problem of tree automata [11] and the decision problem for the clausal fragment of the weak monadic second order theory with n successors [23] can be reduced to the problem of deciding whether or not $M(P) \models \varphi$ holds for suitable programs and formulas in those classes. Finally, in Section 7 we compare our transformational proof method to other theorem proving methods, such as, (i) methods based on the *Clark completion* [6, 18], (ii) methods based on *resolution* augmented by the *negation as* (*finite or infinite*) *failure rule* [1, 22], (iii) methods based on *induction*, and (iv) the *proof by consistency* method [14, 15, 19].

2 Preliminaries

In presenting our proof method we consider locally stratified logic programs. Let us begin by recalling some basic definitions. For notions not defined here the reader may refer to [1, 18, 22].

The formulas and programs we consider are constructed by using a fixed first-order language $\mathcal{L}$. Programs are conjunctions of clauses and clauses are of the form: $H \leftarrow L_1, \ldots, L_n$, where H is an atom and for every $i = 1, \ldots, n$, L_i is a literal, that is, an atom (i.e., *positive* literal) or a negated atom (i.e., *negative* literal). A goal is a conjunction of literals. The empty conjunction is *true* and the empty disjunction is *false*. Conjunction between literals, goals, and clauses is denoted by comma and also by $\wedge$. Conjunction is associative, commutative, and its neutral element is *true*. Thus, the order of literals in a goal is immaterial. For $n=0$, the clause $H \leftarrow L_1, \ldots, L_n$ is said to be a *unit* clause and it is written as $H \leftarrow true$ or simply H. A clause (or a program) is said to be *definite* iff no negated atom occurs in it. We will feel free to rename the variables occurring in clauses. This renaming process is called *renaming apart* [18]. The *head* and the *body* of a clause C are denoted by

$hd(C)$ and $bd(C)$, respectively. The predicate symbol of the atom $hd(C)$ is called the *head predicate* of C.

Given a term t we denote by $vars(t)$ the set of all variables occurring in t. Similar notations will be used for the variables occurring in literals, goals, clauses, and formulas. Given a clause C, a variable in $bd(C)$ is said to be *existential* iff it belongs to $vars(bd(C)) - vars(hd(C))$. Given a formula φ we denote by $freevars(\varphi)$ the set of all free variables of φ. A clause C is said to be *ground* iff no variable occurs in it. A literal is said to be *propositional* iff its predicate symbol is nullary, that is, it has arity 0. A goal (or a clause, or a program) is propositional iff all its literals are propositional. A formula is said to be *function-free* iff no function symbols occur in it.

The *definition* of a predicate p in a program P, denoted by $Def(p,P)$, is the conjunction of all clauses of P whose head predicate is p. We say that p *is defined in* P iff $Def(p,P)$ is not empty. We say that a predicate p *depends on* a predicate q in P iff either there exists in P a clause of the form: $p(\ldots) \leftarrow B$ such that q occurs in the goal B or there exists in P a predicate r such that p depends on r in P and r depends on q in P. The *extended definition* of a predicate p in a program P, denoted by $Def^*(p,P)$, is the conjunction of the definition of p and the definitions of all the predicates on which p depends in P.

The set of the *useless* predicates of a program P is the maximal set U of predicates of P such that a predicate p is in U iff the body of each clause of $Def(p,P)$ has a positive literal whose predicate is in U. For instance, p and q are useless and r is not useless in the program $(p \leftarrow q, r) \wedge (q \leftarrow p) \wedge (r \leftarrow)$.

By $ground(P)$ we denote the conjunction of all clauses in $\mathcal{L}$ which are ground instances of clauses of P, and by $B_{\mathcal{L}}$ we denote the *Herbrand Base* of $\mathcal{L}$, that is, the set of all ground atoms in $\mathcal{L}$. A stratification σ is a total function from $B_{\mathcal{L}}$ to the set W of countable ordinals. Given a ground literal L which is the atom A or the negated atom $\neg A$, we say that L is in *stratum* α iff $\sigma(A) = \alpha$.

A ground clause $H \leftarrow L_1, \ldots, L_n$ is locally stratified w.r.t. a stratification σ iff for every $i = 1, \ldots, n$, if L_i is an atom then $\sigma(H) \geq \sigma(L_i)$, and if L_i is a negated atom then $\sigma(H) > \sigma(L_i)$. We say that the program P is *locally stratified* iff there exists a stratification σ such that every clause in $ground(P)$ is locally stratified w.r.t. σ.

Let P_α be the conjunction of the clauses in $ground(P)$ whose head is in the stratum α. We may assume without loss of generality, that the stratum of every ground literal is greater than 0, so that P_0 may be assumed to be the empty conjunction of clauses. The *perfect model* $M(P)$ of a program P which is locally stratified w.r.t. a stratification σ, is the subset $\bigcup_{\alpha \in W} M_\alpha$ of $B_{\mathcal{L}}$, where for every ordinal α in W, the set M_α is defined as follows:
(1) M_0 is the empty set,
(2) for every successor ordinal $\alpha+1$, $M_{\alpha+1}$ is the least Herbrand model of the program $P_{\alpha+1}$ whose ground clauses have been transformed as follows: (i) every literal L in stratum τ $(\leq \alpha)$ in the body of a clause is deleted iff $M_\tau \models L$, and (ii) every clause C in $P_{\alpha+1}$ is deleted iff in $bd(C)$ there exists a literal L in stratum τ $(\leq \alpha)$ such that $M_\tau \not\models L$, and
(3) for every limit ordinal λ, M_λ is $\bigcup_{\alpha<\lambda} M_\alpha$.

3 Translating First-Order Formulas into Logic Programs

In this section we present a method that given a locally stratified program P and a closed first order formula φ in $\mathcal{L}$, introduces a new predicate f and constructs a locally stratified program $P \wedge F(f,\varphi)$ such that $M(P) \models \varphi$ iff $M(P \wedge F(f,\varphi)) \models f$. This method is based upon a variant of a transformation proposed by Lloyd and Topor [18].

In order to construct the locally stratified program $F(f,\varphi)$ for a given closed first-order formula φ, we need to consider a class of formulas, called *statements* [18], of the form $A \leftarrow \beta$ where A is an atom and β is a (possibly open) first-order logic formula. We write $C[\gamma]$ to denote a first-order formula where the subformula γ occurs as a conjunct 'at top level', that is, $C[\gamma] = \rho_1 \wedge \ldots \wedge \rho_r \wedge \gamma \wedge \sigma_1 \wedge \ldots \wedge \sigma_s$ for some first-order formulas $\rho_1, \ldots, \rho_r, \sigma_1, \ldots, \sigma_s$, and some $r \geq 0$ and $s \geq 0$. When we say that the formula $C[\gamma]$ is transformed into the formula $C[\delta]$, we mean that $C[\delta]$ is obtained from $C[\gamma]$ by replacing the top level conjunct γ by the new top level conjunct δ.

From any given conjunction of statements the following LT transformation, similar to the one proposed by Lloyd and Topor in [18], terminates and it produces a conjunction of clauses which is a locally stratified program.

The LT transformation from conjunctions of statements to locally stratified programs. Let $\mathcal{S}$ be a conjunction of statements.
(A) Eliminate from the body of every statement in $\mathcal{S}$ all occurrences of logical constants, connectives, and quantifiers other than $\mathit{true}, \neg, \wedge$, and $\exists$. For every statement st of $\mathcal{S}$, rename the bound variables of st so that none of them occurs in $\mathit{freevars}(st)$ and all of them are distinct.

(B) Transform every statement derived at the end of Step (A) by applying *as long as possible* the following rules:

(B.1) $A \leftarrow C[\neg \mathit{true}]$ is deleted
(B.2) $A \leftarrow C[\neg\neg\varphi]$ is transformed into $A \leftarrow C[\varphi]$
(B.3) $A \leftarrow C[\neg(\varphi \wedge \psi)]$ is transformed into $A \leftarrow C[\neg \mathit{newp}(y_1, \ldots, y_k)] \;\wedge$
$\mathit{newp}(y_1, \ldots, y_k) \leftarrow \varphi \wedge \psi$

where newp is a new predicate symbol and $\{y_1, \ldots, y_k\} = \mathit{freevars}(\varphi \wedge \psi)$. We assume that $\varphi \neq \mathit{true}$ and $\psi \neq \mathit{true}$.

(B.4) $A \leftarrow C[\neg \exists x\, \varphi]$ is transformed into $A \leftarrow C[\neg \mathit{newp}(y_1, \ldots, y_k)] \;\wedge$
$\mathit{newp}(y_1, \ldots, y_k) \leftarrow \varphi$

where newp is a new predicate symbol and $\{y_1, \ldots, y_k\} = \mathit{freevars}(\exists x\, \varphi)$.

(B.5) $A \leftarrow C[\exists x\, \varphi]$ is transformed into $A \leftarrow C[\varphi]$ □

Given a locally stratified program P and a closed first-order formula φ, we denote by $F(f, \varphi)$ the conjunction of the clauses derived by applying the LT transformation to the statement $f \leftarrow \varphi$, where f is a new predicate symbol occurring neither in φ nor in P. We assume that the new predicates introduced during the construction of $F(f, \varphi)$ do not occur in P.

The reader may verify that in the Semaphore Example of Section 1, clauses 5 and 6 have been derived by applying the LT transformation starting from the following statement:

$$f \;\leftarrow\; \forall x, y\, (x > y,\; \mathit{up}(x, y) \rightarrow \mathit{up}(ssx, 0))$$

The following result states that the LT transformation preserves perfect models, and thus, Step 1 of our proof method is sound. Recall that, instead, the Lloyd-Topor transformation as described in [18], is known to preserve the completion semantics only.

Theorem 1. [Correctness of LT Transformation w.r.t. Perfect Models] Let P be a locally stratified program, φ be a closed first-order formula, and f be a predicate symbol occurring neither in φ nor in P. If $F(f, \varphi)$ is obtained from $f \leftarrow \varphi$ by the LT transformation, then (i) $P \wedge F(f, \varphi)$ is a locally stratified program, and (ii) $M(P) \models \varphi$ iff $M(P \wedge F(f, \varphi)) \models f$.

Proof. (i) Since f is a new predicate symbol and rules B.1-B.5 do not introduce cycles in the dependency graph, we have that the dependency graph of $F(f, \varphi)$ has no cycles. Since no predicate in P depends on the predicates in $F(f, \varphi)$ and P is locally stratified, we have that $(P \wedge F(f, \varphi))$ is locally stratified.

(ii) We have that $M(P \wedge F(f, \varphi)) \models \mathit{comp}(P \wedge F(f, \varphi))$ where $\mathit{comp}(P)$ denotes the Clark completion of program P [18]. By construction we have that $\mathit{comp}(P \wedge F(f, \varphi)) \models f \leftrightarrow \varphi$ and thus, $M(P \wedge F(f, \varphi)) \models f \leftrightarrow \varphi$. To conclude the proof it remains to show that $M(P) \models \varphi$ holds iff $M(P \wedge F(f, \varphi)) \models \varphi$. This follows from the fact that no predicate defined in P and no predicate occurring in φ depends on a predicate defined in $F(f, \varphi)$. □

4 Transformation Rules

In this section we present the rules for transforming logic programs and we provide a sufficient condition which ensures that perfect models are preserved during program transformation.

A *transformation sequence* is a sequence of programs $P_0, \ldots, P_n$, where, for $0 \leq k \leq n-1$, program P_{k+1} is derived from program P_k by the application of a transformation rule as indicated below.

For the application of the transformation rules we divide the predicate symbols of the language into two categories: *basic* predicates and *non-basic* predicates. Atoms, literals, and goals which have occurrences of basic predicates only, are called *basic atoms*, *basic literals*, and *basic goals*, respectively. We assume that each basic atom is in a strictly smaller stratum w.r.t. any non-basic atom.

Our partition of the set of predicates into basic or non-basic predicates is arbitrary and it is required only for proving the Correctness Theorem below. The actual partition one may want to consider in practice, depends on the given initial program P_0. For instance, if one is interested in proving properties of a program on the domain of integer numbers, one may assume that addition, multiplication, and ordering relations are defined by basic predicates.

For $0 \leq k \leq n$, we also consider the conjunction $Defs_k$ of *definitions,* constructed as follows:
(1) $Defs_0$ is the conjunction of every clause C in program P_0 of the form $p(x_1, \ldots, x_m) \leftarrow L_1, \ldots, L_n$, with $n > 0$ such that: (i) $x_1, \ldots, x_m$ are distinct variables (possibly not all variables) occurring in the goal $L_1, \ldots, L_n$, (ii) at least one literal among $L_1, \ldots, L_n$ is a non-basic positive literal, (iii) no predicate symbol occurring in the goal $L_1, \ldots, L_n$ depends on p in P_0, and (iv) $C = Def(p, P_0)$;
(2) for $k > 0$, $Defs_k$ is the conjunction of the clauses in $Defs_0$ and those introduced by the definition rule (see rule R1) during the transformation sequence $P_0, \ldots, P_k$.

R1. Definition Rule. We get program P_{k+1} by adding to program P_k a clause C of the form: $newp(x_1, \ldots, x_m) \leftarrow L_1, \ldots, L_n$, with $n > 0$, such that: (i) $x_1, \ldots, x_m$ are distinct variables occurring in $L_1, \ldots, L_n$, (ii) at least one literal among $L_1, \ldots, L_n$ is a non-basic positive literal, (iii) the predicate symbol *newp* is a non-basic predicate which does not occur in $P_0, \ldots, P_k$, and (iv) all predicate symbols occurring in the goal $L_1, \ldots, L_n$ also occur in P_k.

R2. Positive Unfolding Rule. Let C be a renamed apart clause in P_k of the form $H \leftarrow G_1, A, G_2$, where A is an atom, and G_1 and G_2 are (possibly empty) goals. Suppose that:

1. $D_1, \ldots, D_m$, with $m \geq 0$, are all clauses of program P_k, such that A is unifiable with $hd(D_1)$, $\ldots$, $hd(D_m)$, with most general unifiers $\theta_1, \ldots, \theta_m$, respectively, and
2. C_i is the clause $(H \leftarrow G_1, bd(D_i), G_2)\theta_i$, for $i = 1, \ldots, m$.

By *unfolding C w.r.t. A* we derive from program P_k the new program P_{k+1} by replacing C by $C_1, \ldots, C_m$.

In particular, if $m = 0$, that is, if we unfold a clause C in program P_k w.r.t. an atom which is not unifiable with the head of any clause in P_k, then we derive from program P_k the new program P_{k+1} by deleting clause C.

R3. Negative Unfolding Rule. Let C be a renamed apart clause in P_k of the form $H \leftarrow G_1, \neg A, G_2$, where A is an atom, and G_1 and G_2 are (possibly empty) goals. Let $D_1, \ldots, D_m$, with $m \geq 0$, be all clauses of program P_k, such that A is unifiable with $hd(D_1), \ldots, hd(D_m)$, with most general unifiers $\theta_1, \ldots, \theta_m$, respectively. Assume that:

1. $A = hd(D_1)\theta_1 = \cdots = hd(D_m)\theta_m$, that is, for $i = 1, \ldots, m$, A is an instance of $hd(D_i)$,
2. for $i = 1, \ldots, m$, D_i has no existential variables, and
3. from G_1, $\neg(bd(D_1)\theta_1 \vee \ldots \vee bd(D_m)\theta_m)$, G_2 we get an equivalent disjunction $Q_1 \vee \ldots \vee Q_r$ of goals, with $r \geq 0$, by first pushing $\neg$ inside and then pushing $\vee$ outside.

By *unfolding* C *w.r.t.* $\neg A$ we derive from program P_k the new program P_{k+1} by replacing C by $C_1, \ldots, C_r$, where C_i is the clause $H \leftarrow Q_i$, for $i = 1, \ldots, r$.

In particular: (i) if $m = 0$, that is, if we unfold a clause C w.r.t. a negative literal $\neg A$ such that A is not unifiable with the head of any clause in P_k, then we get the new program P_{k+1} by deleting $\neg A$ from the body of clause C, and (ii) if for some $i \in 1, \ldots, m$, $bd(D_i) = true$, that is, if we unfold a clause C w.r.t. a negative literal $\neg A$ such that A is an instance of the head of a unit clause in P_k, then we derive from program P_k the new program P_{k+1} by deleting clause C.

R4. Folding Rule. Let D be a renamed apart definition in $Defs_k$ and C be a clause in P_k of the form $H \leftarrow G_1, B, G_2$, where B, G_1, and G_2 are (possibly empty) goals. Suppose that for some substitution θ: (i) $B = bd(D)\theta$, and (ii) for every variable x in the set $vars(D) - vars(hd(D))$, we have that $x\theta$ is a variable which occurs neither in $\{H, G_1, G_2\}$ nor in the term $y\theta$, for any variable y occurring in $bd(D)$ and different from x.

By *folding* clause C w.r.t. B using clause D we derive clause $E : H \leftarrow G_1, hd(D)\theta, G_2$ and we get from program P_k the new program P_{k+1} by replacing C by E.

R5. Tautology Rule. We get the new program P_{k+1} by replacing in P_k a conjunction of clauses by the corresponding equivalent conjunction of clauses, according to the following equivalences, where G_1 and G_2 denote (possibly empty) goals, and H and A denote atoms:

(1) $(H \leftarrow G_1, A, \neg A, G_2) \;\leftrightarrow\; true$
(2) $(H \leftarrow G_1, H, G_2) \;\leftrightarrow\; true$
(3) $(H \leftarrow G_1,\;\; H \leftarrow G_1, G_2) \;\leftrightarrow\; (H \leftarrow G_1)$
(4) $(H \leftarrow G_1, A_1, \ldots, A_n, G_2,\;\; H \leftarrow G_1, \neg A_1, G_2,\; \ldots,\; H \leftarrow G_1, \neg A_n, G_2) \;\leftrightarrow\; (H \leftarrow G_1, G_2)$

R6. Clause Deletion Rule. We get the new program P_{k+1} by removing from P_k the definitions of the useless predicates of P_k.

R7. Goal Replacement Rule. Let C be a renamed apart clause in P_k of form $H \leftarrow G_1, Q, G_2$, where Q, G_1, and G_2 are (possibly empty) goals. Suppose that, for some goal R,

$M(P_0) \models \forall x_1 \ldots x_u\, (\exists y_1 \ldots y_v\, Q \leftrightarrow \exists z_1 \ldots z_w\, R)$

where: (i) $\{y_1, \ldots, y_v\} = vars(Q) - vars(H, G_1, G_2)$, (ii) $\{z_1, \ldots, z_w\} = vars(R) - vars(H, G_1, G_2)$, and (iii) $\{x_1, \ldots, x_u\} = vars(Q, R) - \{y_1, \ldots, y_v, z_1, \ldots, z_w\}$. Suppose also that Q and R are basic goals and H is a non-basic atom. Then from program P_k we derive the new program P_{k+1} by replacing C by the clause $H \leftarrow G_1, R, G_2$.

Theorem 2. [Correctness of the Transformation Rules] Let $P_0, \ldots, P_n$, be a transformation sequence such that the following holds: *if* for some k with $1 \leq k < n$, we have applied rule R4 for folding clause C in P_k using clause D in $P_0 \wedge Defs_k$, *then* there exists i, with $0 \leq i < k$ such that D occurs in P_i and P_{i+1} is derived from P_i by positive unfolding of D w.r.t. a non-basic atom. Then we have that $M(P_0 \wedge Defs_n) = M(P_n)$.

Proof. Notice that negative unfolding may destroy correctness as shown by the following example. Let P_0 be the program $(p \leftarrow \neg q(x)) \wedge (q(x) \leftarrow q(x)) \wedge (q(x) \leftarrow r)$. By negative unfolding of $\neg q(x)$ we get program P_1: $(p \leftarrow \neg q(x), \neg r) \wedge (q(x) \leftarrow q(x)) \wedge (q(x) \leftarrow r)$. Then by folding we get program P_2: $(p \leftarrow p, \neg r) \wedge (q(x) \leftarrow q(x)) \wedge (q(x) \leftarrow r)$. We have that $M(P_0) \models p$, while $M(P_2) \models \neg p$.

The proof of Theorem 2 is an extension of the proofs in [25, 27] and it is based on the fact that if we establish a recursion via a folding step, thereby generating from program P_i the new program P_{i+1}, then the size (according to a suitable measure) of the derivations of ground atoms w.r.t. P_i, is not smaller than the size of the corresponding derivations w.r.t. P_{i+1}. This property is indeed ensured by the positive unfolding of the clause used for the folding step leading from P_i to P_{i+1}. $\square$

5 A Strategy for Unfold/Fold Proofs

In order to verify whether or not $M(P) \models \varphi$ holds, Step 2 of our unfold/fold proof method requires the construction of a transformation sequence from program $P \wedge F(f, \varphi)$ to a new program, say T, such that *either* (i) $Def(f, T)$ is f, and in this case we infer that $M(P) \models \varphi$, *or* (ii) T is a program where $Def(f, T)$ is the empty conjunction, and in this case we infer that $M(P) \not\models \varphi$. To construct this transformation sequence we need a strategy for guiding the application of the transformation rules, and indeed, in this section we present such a strategy, called UFS (short for Unfold/Fold proof Strategy). It is an extension of the strategy introduced in the case of definite logic programs for eliminating existential variables [21]. The basic idea is that by eliminating existential variables, from $P \wedge F(f, \varphi)$ we may derive a program, say S, such that $Def^*(f, S)$ is a *propositional* program. Then, we can transform S by using the clause deletion, unfolding, tautology, and goal replacement rules, into a program T satisfying either (i) or (ii) above.

Obviously, since in general $M(P) \models \varphi$ is an undecidable property, our UFS strategy may fail to produce a propositional program S starting from an arbitrary program $P \wedge F(f, \varphi)$. However, in the next section we will show that in some cases our strategy is guaranteed to terminate and it produces a propositional program S, and hence, the desired program T. Thus, our UFS strategy is a decision procedure for the problem of verifying whether or not $M(P) \models \varphi$ holds, when P and φ belong to suitable classes of programs and formulas, respectively. Moreover, even when our strategy derives a program which is *not* propositional, it may still be the case that we are able to infer whether or not $M(P) \models \varphi$ holds, and indeed, this happens in the Semaphore Example given in the Introduction.

For specifying the UFS strategy, we need the following definition (see also [18]).

Definition 1. A *level mapping* of a program P is a mapping from the set of predicate symbols occurring in P to the set of natural numbers. The *level* of predicate p is the value of p under this mapping.

By the definition of the LT transformation there exists a level mapping of $P \wedge F(f, \varphi)$ such that: (i) the level of every predicate defined in P is 0, (ii) the level of every predicate defined in $F(f, \varphi)$ is greater than 0, (iii) for each clause $p(\ldots) \leftarrow B$ in $F(f, \varphi)$ the level of every predicate symbol in B is strictly smaller than the level of p, and (iv) the predicate f has the highest level, say K. For instance, in the Semaphore Example we may choose the level mapping m defined by the following equations: $m(down) = m(up) = m(>) = 0$, $m(g) = 1$, and $m(f) = 2$ (thus, $K = 2$).

The UFS strategy, as we will present it now, requires three substrategies: (1) UNFOLD, (2) TAUTOLOGY & REPLACE, and (3) DEFINE & FOLD. Below they will be made algorithmic for the particular classes of programs and formulas for which the UFS strategy terminates.

The Unfold/Fold Proof Strategy UFS.
Input: (i) $P \wedge F(f, \varphi)$, where P is a locally stratified program and φ is a closed first order formula, and (ii) a set *Laws* of equivalences of the form: $M(P) \models \forall x_1 \ldots x_u\, (\exists y_1 \ldots y_v\, Q \leftrightarrow \exists z_1 \ldots z_w\, R)$, where Q and R are basic goals.
Output: A program T such that: (i) $M(T) \models f$ iff $M(P \wedge F(f, \varphi)) \models f$, and (ii) $Def(f, T)$ is either f or the empty conjunction.

Let $P \wedge F(f, \varphi)$ be $P \wedge P^1 \wedge \ldots \wedge P^K$, where for $i = 1, \ldots, K$, program P^i is the conjunction of the clauses whose head predicate has level i.

$T := P$;
FOR $i = 1, \ldots, K$ **DO**
Let *Pos* be the conjunction of the clauses of P^i whose body has at least one non-basic positive literal, and *Neg* be the conjunction of the clauses of P^i which are not in *Pos*.
Let *Defs* be *Pos* and *Out* be the empty conjunction.
WHILE *Pos* is not the empty conjunction **DO** (†)

(1) UNFOLD(T, Pos, U): From program $T \wedge Pos$ we derive $T \wedge U$ by a finite sequence of applications of the positive or negative unfolding rules to the clauses in Pos. We require that:
[*Progression*] the positive unfolding rule is applied at least once to each clause in Pos.
(2) TAUTOLOGY & REPLACE(T, $Laws$, U, R): From program $T \wedge U$ we derive $T \wedge R$ by a finite sequence of applications of the tautology and goal replacement rules to the clauses in U, using the equivalences in the set $Laws$.
(3) DEFINE & FOLD(T, R, $Defs$, $OutClauses$, $NewDefs$): From program $T \wedge R$ we derive $T \wedge OutClauses \wedge NewDefs$ by: (i) a finite sequence of applications of the definition rule by which we introduce the (possibly empty) conjunction $NewDefs$ of clauses, followed by (ii) a finite sequence of applications of the folding rule to the clauses in R, using clauses occurring in $Defs \wedge NewDefs$. We assume that the definition and folding steps are such that the following conditions are satisfied:
(3.1) [*Positive definitions*] The body of each clause in $NewDefs$ has at least one non-basic positive literal.
(3.2) [*No existential variables*] Each clause in $OutClauses$ which has been derived by folding has no existential variables.
(3.3) [*Full folding*] Each predicate in the body of each clause in $OutClauses$ which has been derived by folding, occurs in $Defs \wedge NewDefs$.
$Out := Out \wedge OutClauses$; $Pos := NewDefs$; $Defs := Defs \wedge NewDefs$

OD;
Delete from Out the definitions of useless predicates, thereby deriving Out'; $T := T \wedge Out' \wedge Neg$;
Initialize D to the conjunction of all definitions of nullary predicates in T;
Initialize Q to the conjunction of all definitions of non-nullary predicates in T;
WHILE D is a not a conjunction of unit clauses **DO**   (††)

UNFOLD(Q, D, U): From program $Q \wedge D$ we derive $Q \wedge U$ by a (possibly empty) sequence of applications of the positive or negative unfolding rules to the clauses in D. (Here the Progression requirement need not be satisfied.)
TAUTOLOGY & REPLACE(Q, $Laws$, U, R); $D := R$

OD;
Unfold the clauses of Q w.r.t. their literals with nullary predicates, thereby deriving Q';   (†††)
$T := Q' \wedge D$
END FOR □

Our UFS strategy proceeds by iterating over levels (from level 1 to level K) a sequence of transformations on the program T, which is initialized to P. For $i = 1, \ldots, K$, program P^i which is the conjunction of the clauses defining the predicates with level i, is processed by the UFS strategy, and the WHILE loop (†) generates from program $T \wedge P^i$ (which is $T \wedge Neg \wedge Pos$) the new program T. The objective of the WHILE loop (†) is: (i) to ensure that positive unfolding steps w.r.t. non-basic atoms are performed before folding, and (ii) to avoid existential variables via definition and folding steps. Then useless predicates are deleted. Finally, the WHILE loop (††) performs on the definitions of nullary predicates unfolding, tautology, and goal replacement steps so to reduce each of them, if possible, either to the empty definition (in which case the corresponding predicate is *false*) or to a unit clause (in which case the corresponding predicate is *true*). The truth values of the propositional predicates are then propagated by the unfolding steps (†††). When the last program level K has been processed, we get for the predicate f either the empty definition or the clause f. Thus, we may establish whether or not $M(P \wedge F(f, \varphi)) \models f$ holds.

The reader may check that our introductory Semaphore Example has indeed been worked out using the UFS proof strategy.

The soundness of our proof strategy follows from the fact that the transformation rules are used in such a way that the hypothesis of the Correctness Theorem of Section 4 holds.

Obviously, the UFS strategy process may not terminate, because in general $M(P \wedge F(f, \varphi)) \models f$ is undecidable, and indeed: (i) during the execution of the WHILE loop (†) we may introduce by applying the definition rule, an infinite number of new clauses, and thus, *Pos* never becomes the empty conjunction, and (ii) the WHILE loop (††) may not terminate for nullary predicates which depend on non-nullary predicates.

6 Decision Procedures Using the Unfold/Fold Proof Strategy

In this section we present two classes of formulas, called *tree-typed formulas* and *tree-typed clausal formulas*, and two classes of programs, called *MR programs* and *DL programs*, for which a deterministic version of the UFS strategy, called dUFS, terminates. Thus, the dUFS strategy is a decision procedure for establishing whether or not $M(P) \models \varphi$ holds, when φ and P belong to the given classes.

In this section we assume that all predicates are non-basic. The tree-typed formulas are function-free formulas whose variables range over sets of trees denoted by *tree programs* which we now define.

Definition 2. [Tree Programs] A *tree program* is a conjunction of tree clauses. A *tree clause* is a clause of the form: $r_0(t(x_1, \ldots, x_n)) \leftarrow r_1(x_1), \ldots, r_n(x_n)$, with $n \geq 0$, where t is a function symbol and $x_1, \ldots, x_n$ are distinct variables. A *tree predicate* is a predicate defined by a tree program. A *tree atom* is an atom with a tree predicate.

Definition 3. [Tree-typed Formulas] A *tree-typed formula* over a program P is a first-order formula φ, defined as follows:

$$\varphi \ ::= \ p(x_1, \ldots, x_n) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \forall x\, (r(x) \rightarrow \varphi) \mid \exists x\, (r(x) \wedge \varphi)$$

where: (i) $x_1, \ldots, x_n$, with $n \geq 0$, are distinct variables, (ii) all predicates occurring in φ are defined in P, and (iii) $Def^*(r, P)$ is a tree program.

Example. [*Even-Odd Paths*] Let us consider the following *Even-Odd* program:

1. $bin(leaf)$
2. $bin(t(x,y)) \leftarrow bin(x), bin(y)$
3. $even(leaf)$
4. $even(t(x,y)) \leftarrow \neg even(x)$
5. $even(t(x,y)) \leftarrow \neg even(y)$
6. $odd(t(x,y)) \leftarrow \neg odd(x), \neg odd(y)$

Clauses 1 and 2 are tree clauses. The formula $\varphi : \forall x\, (bin(x) \rightarrow even(x) \vee odd(x))$ is a tree-typed formula over *Even-Odd*. Informally, φ means that in every binary tree there exists a path of even length or all paths have odd length. Thus, we expect that $M(P) \models \varphi$ holds, as we will formally prove below. □

The Tree-typed LT Transformation.
In order to construct the locally stratified program $F(f, \varphi)$ for a given tree-typed formula φ, we apply the so called *tree-typed LT transformation*, which is a variant of the LT transformation (see Section 3) obtained by replacing rules (B.3) and (B.4) by the following ones:

(B.3)* $A \leftarrow C[\neg(\varphi \wedge \psi)]$ is transformed into $A \leftarrow C[\neg newp(y_1, \ldots, y_k)] \ \wedge$
$newp(y_1, \ldots, y_k) \leftarrow \rho \wedge \varphi \wedge \psi$

where *newp* is a new predicate symbol, $\{y_1, \ldots, y_k\} = freevars(\varphi \wedge \psi)$, and ρ is the conjunction of all tree atoms $r(y)$ occurring as conjuncts at top level in $C[\neg(\varphi \wedge \psi)]$ and such that $y \in \{y_1, \ldots, y_k\}$.

(B.4)* $A \leftarrow C[\neg\exists x\, (r(x) \wedge \varphi)]$ is transformed into $A \leftarrow C[\neg newp(y_1, \ldots, y_k)] \ \wedge$
$newp(y_1, \ldots, y_k) \leftarrow \rho \wedge r(x) \wedge \varphi$

where *newp* is a new predicate symbol, $\{y_1, \ldots, y_k\} = freevars(\exists x\, (r(x) \wedge \varphi))$, and ρ is the conjunction of all tree atoms $r(y)$ occurring as conjuncts at top level in $C[\neg\exists x\, (r(x) \wedge \varphi)]$ and such that $y \in \{y_1, \ldots, y_k\}$. □

Example. [*Even-Odd Paths. Continued*] For the tree-typed formula $\varphi : \forall x\, (bin(x) \rightarrow even(x) \vee odd(x))$, the corresponding $F(f, \varphi)$ obtained by tree-typed LT transformation is the conjunction of the following two clauses:

7. $f \leftarrow \neg g$
8. $g \leftarrow bin(x), \neg even(x), \neg odd(x)$ □

The tree-typed LT transformation preserves perfect models as stated by the following theorem.

Theorem 3. [Correctness of the tree-typed LT Transformation w.r.t. Perfect Models] For every locally stratified logic program P and closed tree-typed formula φ over P, if $F(f, \varphi)$ is obtained from φ by the tree-typed LT transformation, then
(i) $P \wedge F(f, \varphi)$ is a locally stratified program and $M(P) \models \varphi$ iff $M(P \wedge F(f, \varphi)) \models f$, and
(ii) for each clause C in $F(f, \varphi)$ we have that: (ii.1) C is of the form $newp(x_1, \ldots, x_m) \leftarrow r_1(x_1), \ldots, r_n(x_n), G$, where $r_1(x_1), \ldots, r_n(x_n)$ are tree atoms, (ii.2) $vars(C) = \{x_1, \ldots, x_n\}$, (ii.3) G is a function-free goal, and (ii.4) $C = Def(newp, P \wedge F(f, \varphi))$.

Proof. See Appendix 2.

Definition 4. [Monadic Regular Programs] A *monadic regular program* is a conjunction of monadic regular clauses. A *monadic regular clause* (or *an MR clause*, for short) is a locally stratified clause of the form:

$$p_0(t(x_1, \ldots, x_n)) \leftarrow p_1(y_1), \ldots, p_k(y_k), \neg p_{k+1}(y_{k+1}), \ldots, \neg p_m(y_m)$$

with $n, m \geq 0$, where t is a function symbol, $x_1, \ldots, x_n$ are distinct variables, and $y_1, \ldots, y_m$ are (not necessarily distinct) variables occurring in $\{x_1, \ldots, x_n\}$.

Tree programs are MR programs.

We now describe the deterministic version of the UFS strategy, which we call dUFS.

The Unfold/Fold Proof Strategy dUFS.
The dUFS strategy is obtained from the UFS strategy defined in Section 5, by replacing the sub-strategies: (1) UNFOLD, (2) TAUTOLOGY & REPLACE, and (3) DEFINE & FOLD by the following ones, respectively:

(d1) BREADTH-FIRST UNFOLD(T, *Pos*, U): From program $T \wedge Pos$ we derive $T \wedge U$ by: (i) one application of positive unfolding w.r.t. each positive literal occurring in the body of a clause in *Pos*, followed by (ii) one application of negative unfolding w.r.t. each negative literal occurring in the body of a clause in *Pos*.

(d2) TAUTOLOGY(T, U, R): From program $T \wedge U$ we derive $T \wedge R$ by a sequence of applications of the tautology rule, constructed by rewriting as long as possible an instance of the left hand side of an equivalence in Rule R5 by the corresponding instance of the right hand side.

(d3) BLOCK-DEFINE & FOLD(T, R, *Defs*, *OutClauses*, *NewDefs*): From program $T \wedge R$ we derive program $T \wedge OutClauses \wedge NewDefs$ as follows:
for each non-unit clause C in R
(i) we partition $bd(C)$ into subconjunctions, called *blocks*, such that $bd(C) = B_1 \wedge \ldots \wedge B_m$, and two literals occur in the same subconjunction B_i, for some i, $1 \leq i \leq m$, iff they share a variable,
(ii) for $i = 1, \ldots, m$, we apply the definition rule and we add to *NewDefs* a clause of the form $Newp \leftarrow B_i$, where $vars(Newp) = vars(B_i) \cap vars(hd(C))$, unless a variant clause already occurs in *Defs*, modulo the head predicate symbol, and
(iii) we fold C w.r.t. B_1, and we fold the resulting clause w.r.t. B_2, and so on, until we fold w.r.t. B_m. □

We have that:
- the BREADTH-FIRST UNFOLD substrategy fulfills the Progression requirement,
- in the TAUTOLOGY substrategy we have omitted the *Laws* argument because goal replacements are not performed, and

- the BLOCK-DEFINE & FOLD substrategy fulfills the conditions (3.1), (3.2), and (3.3) of the DEFINE & FOLD substrategy of Section 5. Indeed, Condition (3.1) is fulfilled, because each variable occurring in the body of a clause generated by the strategy dUFS also occurs in a tree atom, and therefore, each block has at least one non-basic positive literal. Moreover, by Points (ii) and (iii) in (d3) above, also Conditions (3.2) and (3.3) are fulfilled.

Example. [*Even-Odd Paths. Continued*] The proof that $\forall x\, (bin(x) \rightarrow even(x) \vee odd(x))$ holds in the perfect model of the *Even-Odd* program is as follows. We start from program P made out of clauses from 1 to 6, program P^1 made out of clause 8, and program P^2 made out of clause 7.

We start from level 1. Initially $Pos = \text{clause}\, 8$. By breadth-first unfolding, from clause 8 we get the following clauses:

9. $g \leftarrow bin(x), bin(y), even(x), even(y), odd(x)$
10. $g \leftarrow bin(x), bin(y), even(x), even(y), odd(y)$

By the tautology rule R5.(3) we delete clause 10 (it is subsumed by clause 9). Then we apply the BLOCK-DEFINE & FOLD substrategy by introducing the following two definitions:

11. $h_{eo} \leftarrow bin(x), even(x), odd(x)$
12. $h_e \leftarrow bin(y), even(y)$

and by folding clauses 9 we get:

9.f $\quad g \leftarrow h_{eo}, h_e$

Now $Pos = \text{clause}\, 11 \wedge \text{clause}\, 12$ and we have to perform once more the body of the WHILE loop (†) of the dUDS strategy. By unfolding clauses 11 and 12 we get:

11.1 $h_{eo} \leftarrow bin(x), bin(y), \neg even(x), \neg odd(x), \neg odd(y)$
11.2 $h_{eo} \leftarrow bin(x), bin(y), \neg even(y), \neg odd(x), \neg odd(y)$
12.1 h_e
12.2 $h_e \leftarrow bin(x), bin(y), \neg even(x)$
12.3 $h_e \leftarrow bin(x), bin(y), \neg even(y)$

By the tautology rule R5.(3) we delete clause 11.2 (it is subsumed by clause 11.1) and we also delete clauses 12.2 and 12.3 (they are subsumed by clause 12.1). Then we apply the BLOCK-DEFINE & FOLD substrategy by introducing the following definition:

13. $h_o \leftarrow bin(x), \neg odd(x)$

and by folding clause 11.1, we get:

11.1f $h_{eo} \leftarrow g, h_o$

Now $Pos = \text{clause}\, 13$. By unfolding clause 13 we get:

13.1 h_o
13.2 $h_o \leftarrow bin(x), bin(y), \neg odd(x), \neg odd(y)$

By the tautology rule R5.(3) we delete clause 13.2 (it is subsumed by clause 13.1). No application of the BLOCK-DEFINE & FOLD substrategy is required. Now *Pos* is empty, and *Out* is the conjunction of the following clauses:

9.f $\quad g \leftarrow h_{eo}, h_e$
11.1f $h_{eo} \leftarrow g, h_o$
12.1 $\quad h_e$
13.1 $\quad h_o$

We then delete clauses 9.f and 11.1f because they are the definitions of the predicates g and h_{eo} which are useless in *Out*. The WHILE loop (††) does not change D. Since $Q = Q'$, we have that T is made out of the clauses of P together with the clauses 12.1 and 13.1.

We can now start processing level 2, that is, program P^2. Since *Pos* is the empty conjunction, the body of the WHILE loop (†) is never executed. *Neg* is clause 7, and before the execution of the WHILE loop (††) program T is made out of the clauses in P together with the following clauses:

7. $\quad f \leftarrow \neg g$
12.1 $\quad h_e$
13.1 $\quad h_o$

After the WHILE loop (††) and the statement (†††) we get the new program T made out of the clauses in P and the clause:

7.1 $\quad f$

together with clauses 12.1 and 13.1. Now, since $M(T) \models f$, we have that $M(P) \models \varphi$. $\square$

Obviously the BREADTH-FIRST UNFOLD, TAUTOLOGY, and BLOCK-DEFINE & FOLD substrategies terminate. Also the dUFS proof strategy terminates for tree-typed formulas and MR programs, as stated by the following theorem.

Theorem 4. [Termination of the dUFS Proof Strategy for MR Programs] Let P be an MR program and φ be a tree-typed formula over P. Then the dUFS proof strategy with input program $P \wedge F(f, \varphi)$ terminates with output program T. Moreover, $Def(f, T)$ is f iff $M(P) \models \varphi$ and $Def(f, T)$ is the empty conjunction iff $M(P) \not\models \varphi$.

Proof. See Appendix 3.

Although very restricted, the classes of MR programs and tree-typed formulas allow us to express some interesting properties, such as the equivalence of finite tree automata (see, for instance, [11]), as shown in the following example.

Example. [*Equivalence of Tree Automata*] The tree language recognized by a (nondeterministic top-down) finite tree automaton $T = (Q, \Sigma, \delta, q_{in}, F)$ corresponds to a subset of the perfect model of a tree program P_T (which is also its least Herbrand model) defined as follows:

- for each state $q \in Q$ we define a unary predicate p_q,
- for each symbol $a \in \Sigma$ of arity k we define a k-ary function symbol f_a,
- for each tuple $(q0, a, q1, \ldots, qk)$ in the transition relation δ, where a is a k-ary symbol with $k \geq 1$, P_T has the clause:

$p_{q0}(f_a(x_1, \ldots, x_k)) \leftarrow p_{q1}(x_1), \ldots, p_{qk}(x_k)$

- for each state $q \in F \subseteq Q$ and 0-ary symbol b such that q is a final state for b, P_T has the unit clause:

$p_q(f_b)$

Then the tree language recognized by the tree automaton T with initial state qin, corresponds to the subset $\{t \mid M(P_T) \models p_{qin}(t)\}$. Given another tree automaton $T1$ with initial state $qin1$ and represented by program P_{T1}, we have that T and $T1$ recognize the same language iff

$$M(P_T \wedge P_{T1}) \models \forall x\, (p_{qin}(x) \rightarrow p_{qin1}(x)) \ \wedge\ \forall x\, (p_{qin1}(x) \rightarrow p_{qin}(x)).$$

Thus the equivalence of T and $T1$ can be reduced to the verification of a tree-typed formula over the tree program $P_T \wedge P_{T1}$ (recall that each tree program is also an MR program). $\square$

We now introduce a second class of formulas, called *tree-typed clausal formulas*, and a second class of programs, called *deterministic linear programs* (or *DL programs*, for short) for which the unfold/fold proof strategy dUFS terminates. Thus, given a tree-typed clausal formula φ, and a DL program P, we can decide whether or not $M(P) \models \varphi$ holds by using the proof strategy dUFS.

Definition 5. [Tree-typed Clausal Formulas] Let P be any program and R be a tree program. A *tree-typed clausal formula* over a program $P \wedge R$ is a closed first-order formula φ generated by the following grammar:

$$\varphi \ ::=\ \forall x\, (r(x) \rightarrow \varphi) \mid \delta$$
$$\delta \ ::=\ p(x_1, \ldots, x_n) \mid \neg p(x_1, \ldots, x_n) \mid \delta \vee \delta$$

where: (i) $x_1, \ldots, x_n$, with $n \geq 0$, are distinct variables, (ii) all predicates occurring in φ are defined in $P \wedge R$, (iii) for each predicate p, $Def^*(p, P \wedge R)$ is a subconjunction of P, (iv) for each predicate r, $Def^*(r, P \wedge R)$ is a subconjunction of R, and (v) no predicate is defined in both P and R.

Definition 6. [Deterministic Linear Programs] A *linear clause* is a locally stratified clause of one of these three forms:

$p(t_1(x_1, \ldots, x_m), \ldots, t_n(x_u, \ldots, x_v))$
$p(t_1(x_1, \ldots, x_m), \ldots, t_n(x_u, \ldots, x_v)) \leftarrow q(y_1, \ldots, y_k)$
$p(t_1(x_1, \ldots, x_m), \ldots, t_n(x_u, \ldots, x_v)) \leftarrow \neg q(y_1, \ldots, y_k)$

with $n \geq 0$, where: (i) $t_1, \ldots, t_n$ are function symbols, (ii) $x_1, \ldots, x_m, \ldots, x_u, \ldots, x_v$ are distinct variables, and (iii) $y_1, \ldots, y_k$ are distinct variables occurring in $\{x_1, \ldots, x_m, \ldots, x_u, \ldots, x_v\}$.
A *deterministic linear program* (or *DL program*, for short) is a conjunction of linear clauses such that no two clause heads are unifiable.

Notice that there are MR programs which are not DL programs and vice versa. We have the following result whose proof is given in Appendix 4.

Theorem 5. [Termination of the dUFS Proof Strategy for DL Programs] Let P be a DL program, R be a tree program, and φ be a tree-typed clausal formula over $P \wedge R$. Then the dUFS strategy with input program $P \wedge F(f, \varphi)$ terminates with output program T. Moreover, $Def(f, T)$ is f iff $M(P) \models \varphi$ and $Def(f, T)$ is the empty conjunction iff $M(P) \not\models \varphi$.

Example. [*Clausal wMSnS*] The clausal fragment of the weak monadic second order theory of two successor functions (cwMS2S) is the set of closed formulas φ such that $\mathcal{W} \models \varphi$, where:
(i) φ is a closed formula generated by the following grammar:

$$\varphi \ ::= \ \forall w\, (word(w) \rightarrow \varphi) \mid \forall x\, (set(x) \rightarrow \varphi) \mid \delta$$

$$\delta \ ::= \ member(w, x) \mid \neg member(w, x) \mid \delta \vee \delta$$

(ii) $\mathcal{W}$ is the structure $\mathcal{P}_{fin}(\{0,1\}^*)$ of finite sets of words over $\{0,1\}$, where: (ii.1) the successor functions s_0 and s_1 are interpreted as $s_0(w) = w0$ and $s_1(w) = w1$, respectively, (ii.2) the predicates *word* and *set* hold of all words and finite sets of words, respectively, and (ii.3) $member(w, x)$ is interpreted as membership of word w to set x.

In order to write a logic program which computes the *word*, *set*, and *member* predicates, we represent a finite set of words as a finite tree, where (1) t (of arity 3) and *leaf* (of arity 1) are the tree constructors, (2) both the internal nodes and the leaf nodes are labeled by the constants *accept* or *refuse,* and they are called *accept* and *refuse nodes*, respectively, and (3) left arcs are labeled by 0 and right arcs are labeled by 1. The empty word is represented by the constant *emptyword.* The *accept paths* of a tree x are the sequences of labels from an *accept* node to the root. A word w is member of a set x iff w is an accept path of x. For instance, the set $\{0, 01\}$ is represented by the tree $t(leaf(acc), ref, t(leaf(acc), ref, leaf(ref)))$, where we have abbreviated *refuse* and *accept* by *ref* and *acc*, respectively.
We may define the *word*, *set*, and *member* predicates by means of the following *Member* program, which is $P \wedge R$, where:

P:	R:
$member(emptyword, leaf(n)) \leftarrow acceptlabel(n)$	$word(emptyword)$
$member(emptyword, t(x_0, n, x_1)) \leftarrow acceptlabel(n)$	$word(s_0(w)) \leftarrow word(w)$
$member(s_0(w), t(x_0, n, x_1)) \leftarrow member(w, x_0)$	$word(s_1(w)) \leftarrow word(w)$
$member(s_1(w), t(x_0, n, x_1)) \leftarrow member(w, x_1)$	$set(leaf(n)) \leftarrow label(n)$
$acceptlabel(accept)$	$set(t(x_0, n, x_1)) \leftarrow set(x_0), label(n), set(x_1)$
	$label(accept)$
	$label(refuse)$

We have that the structure $\mathcal{W}$ corresponds to the perfect model (equal to the least Herbrand model) of *Member*, in the sense that: $\mathcal{W} \models \varphi$ iff $M(Member) \models \varphi$. Since (1) P is a DL program, (2) R is a tree program, and (3) every formula φ which is generated by the grammar of Point (i) above is a tree-typed clausal formula over $P \wedge R$, we have that by using our dUFS proof strategy we can test whether or not $M(Member) \models \varphi$ holds. Thus, dUFS is a decision procedure for cwMS2S. The extension from cwMS2S to cwMSnS (with n successors, instead of 2) can be obtained by a straightforward modification of the *Member* program. □

There are various possibilities of enhancing the classes of programs for which our dUFS proof strategy terminates. In particular, we may add constant arguments to predicates, we may add nullary predicates, and we may also combine the MR and LD programs in several ways. We do not discuss these enhancements here.

7 Conclusions and Related Work

The idea of using unfold/fold transformations for proving program properties goes back to [16], where it was advocated as a method for proving the equivalence of functional terms. Some limitations and completeness results for this proof method are presented in [2, 8]. Our completeness result (Theorem 4) shows that the unfold, fold, clause deletion, and tautology rules are sufficient to prove the equivalence between tree automata, while Courcelle in [8] proves a similar result by making use of the (restricted) unfold, (restricted) fold, cycle elimination, and redefinition rule. We leave for future research further investigation on the power of the various sets of program transformations.

The present paper extends the techniques proposed in [20] for showing equivalences of definite programs w.r.t. least Herbrand models. The main extensions presented here are the following: (i) we consider logic programs with stratified negation and perfect model semantics, (ii) we prove first-order formulas, and (iii) we present an automated strategy for performing proofs. A different extension of [20] has been recently presented in [24], where the authors prove equivalences of definite programs w.r.t. least Herbrand models by using a more powerful folding rule.

Our transformational method for proving properties of the perfect model of a locally stratified logic program is related to traditional methods for theorem proving. Among these methods we would like to mention the following ones: (i) the method based on the *Clark completion* [6, 18], (ii) the methods based on *resolution* augmented by the *negation as (finite or infinite) failure rule* [1, 22], (iii) the methods based on *induction*, and (iv) the *proof by consistency* method [14, 15, 19].

(i) The method based on the Clark completion amounts to prove that $M(P) \models \varphi$ by showing that $comp(P) \vdash \varphi$, where $comp(P)$ denotes the Clark completion of program P. (Equivalently, since the Lloyd-Topor transformation from $f \leftarrow \varphi$ to $F(f, \varphi)$ preserves the completion semantics, one may show that $comp(P \wedge F(f, \varphi)) \vdash f$.) This method is sound because $M(P) \models comp(P)$ and it reduces the proof of a satisfaction relation to the proof a theorem in first-order predicate calculus. Notice that for some program P and formula φ, such as the ones introduced in our Semaphore and Even-Odd Paths Examples, the relation $M(P) \models \varphi$ can be proved by our unfold/fold proof method and yet $comp(P) \not\vdash \varphi$. This is due to the fact that the transformation rules presented in this paper, with the only exception of the tautology rule, preserve the perfect model of a program P, which is a model of $comp(P)$, but they do *not* necessarily preserve *all* models of $comp(P)$.

(ii) Several enhancements of the resolution method have been proposed in the literature for verifying properties of logic programs with negation w.r.t. the perfect model semantics (or generalizations thereof). SLDNF-resolution [18] is based on the *negation as finite failure* rule, but it is unable to deal with infinitely failed derivations and moreover, it is not complete w.r.t. the Clark completion. We have shown that the ability of detecting and eliminating infinitely failed derivations (via the clause deletion rule) is a crucial step in our method for proving theorems.

SLS-resolution [1, 22] enhances the resolution method by using the *negation as* (finite or infinite*)* *failure* rule. In the absence of *floundering* (see [18] for this notion), SLS resolution is sound and complete w.r.t. the perfect model semantics, however it is not an effective inference rule, because the set of consequences of the perfect model of a logic program is not recursively enumerable.

SLG resolution [5] combines resolution and memoing (also called *tabling*) and it is more effective than the SLDNF or SLS resolution methods for dealing with infinitely failed derivations. Indeed, in [26] it has been shown that logic programming systems based on tabling are good for inductive theorem proving and they are able to verify CTL properties of finite-state transition system [10]. However, the unfold/fold proof method is more powerful than logic programming systems based on tabling, because during program transformation one has the possibility of using the folding rule. This rule is more powerful than tabling, because it allows us to tabulate a conjunction of literals, instead of one literal only. Moreover, during program transformation we may apply the goal replacement rule, which is equivalent to applying lemmas in the proofs by induction. Of course, the automation of the unfold/fold method needs more effort than the automation of tabling, but we believe that this difficulty can be overcome by considering specific classes of programs and properties. And indeed, in this paper we have presented two such classes.

(iii) The satisfaction relation $M(P) \models \varphi$ may also be proved by adding to $comp(P)$ a set of formulas, or *induction schemata*, that formalize an induction principle over terms of the Herbrand universe. Thus, one may use standard techniques for inductive theorem proving [3]. The main difference between this method and our unfold/fold proof method is that the latter does not require any induction schema. We leave for future research a formal comparison between the class of formulas that can be proved by induction and the class of formulas that can be proved by the unfold/fold proof method.

(iv) The unfold/fold proof method is related to methods for *proof by consistency* (also called *inductionless induction* method) of equational formulas by using term rewriting systems (see [14, 15, 19] and [7] for a brief survey). This relationship is based on the ability of the unfold/fold proof method of proving inductive properties without using an explicit induction schema. However, the proofs by consistency are refutational proofs, they work by finding minimal counterexamples, and they require suitable well-founded orderings on terms, while the unfold/fold proof method does not require such term orderings.

Finally, we would like to mention that in [24] the use of unfold/fold proofs (of definite programs) is advocated for verifying parametrized concurrent systems. More generally, since finite and infinite transition systems can be represented in a straightforward way by logic programs, we can use our unfold/fold transformation techniques for proving properties of those systems and thus, performing finite or infinite model checking. The fact that transitions may be subject to negative conditions [13] is not a difficulty in our approach because, as we already mentioned, we allow logic programs with stratified negation. The use of logic programming techniques for infinite state model checking seems to be very promising, as shown in [9, 12, 17]. We believe that the combination of these techniques with transformational proof methods may be very fruitful.

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] G. Boudol and L. Kott. Recursion induction principle revisited. *Theoretical Computer Science*, 22:135–173, 1983.

[3] R. S. Boyer and J. S. Moore. *A Computational Logic*. Academic Press, New York, 1979.

[4] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

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

[6] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[7] H. Comon and R. Nieuwenhuis. Induction = I-axiomatization + first-order consistency. *Information and Computation*, 1999. to appear.

[8] B. Courcelle. Equivalences and Transformations of Regular Systems – Applications to Recursive Program Schemes and Grammars. *Theoretical Computer Science*, 42:1–122, 1986.

[9] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *5th International Conference TACAS'99*, Lecture Notes in Computer Science 1579, pages 223–239. Springer-Verlag, 1999.

[10] E. A. Emerson. Temporal and modal logic. In J. van Leuveen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 997–1072. Elsevier, 1990.

[11] J. Engelfriet. Tree Automata and Tree Grammars. DAIMI FN-10, Department of Computer Science, University of Aarhus, 8000 Aarhus, Denmark, April 1975.

[12] L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *CONCUR '97*, Lecture Notes in Computer Science 1243, pages 96–107. Springer-Verlag, 1997.

[13] J. F. Groote. Transition system specification with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[14] G. Huet and J.-M. Hullot. Proofs by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25:239–266, 1982.

[15] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1):1–33, 1989.

[16] L. Kott. Unfold/fold program transformation. In M. Nivat and J.C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 411–434. Cambridge University Press, 1985.

[17] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In *PreProceedings of LOPSTR '99, Venice, Italy*, pages 137–144. Università Ca' Foscari di Venezia, Dipartimento di Informatica, 1999.

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

[19] D. R. Musser. On proving inductive properties of abstract data types. In *Proceedings of the 7-th ACM Symposium on Principles of Programming Languages (POPL'80)*, pages 154–162. ACM Press, 1980.

[20] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2&3):197–230, 1999.

[21] 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.

[22] T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journ. of Automated Reasoning*, 5:167–205, 1989.

[23] M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–34, July 1969.

[24] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. Proofs by program transformation. In *PreProceedings of LOPSTR '99, Venice, Italy*, pages 57–64. Università Ca' Foscari di Venezia, Dipartimento di Informatica, 1999.

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

[26] 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 *CAV '97*, Lecture Notes in Computer Science 1254, pages 143–154. Springer-Verlag, 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, Uppsala, Sweden*, pages 127–138. Uppsala University, 1984.

Appendix 1. *Derivation of program Q from clauses* 1-6. By unfolding clause 6 w.r.t. $up(x,y)$ we get three clauses: (i) one of them can be deleted because of the atom $0>0$ in its body, (ii) a second one: $g \leftarrow down(x),\, \neg up(sssx,0)$ can also be deleted, because by unfolding $\neg up(sssx,0)$ using clause 3 we get: $g \leftarrow down(x),\, \neg down(ssx)$, and eventually, we get: $g \leftarrow down(x),\, \neg down(x)$, and (iii) the third clause produces, after a goal replacement step based on the equivalence $(sx>sy,\, x>y) \leftrightarrow x>y$, the following clause:

7. $g \leftarrow x>y,\ up(sx,y),\ \neg up(sssx,0)$

By the definition and folding rules, from clause 7 we get:

8. $h \leftarrow x > y,\ up(sx, y),\ \neg up(sssx, 0)$

7.f $g \leftarrow h$

By unfolding clause 8 w.r.t. $up(sx, y)$ and by performing the same steps which led us from clause 6 to clause 7, we get:

9. $h \leftarrow x > sy,\ up(sx, y),\ \neg up(sssx, 0)$

From this clause, by applying the goal replacement rule based on the equivalence $plus(y, s0, sy) \leftrightarrow true$, and then by applying the definition and folding rules, we get:

10. $k(n) \leftarrow x > z,\ plus(y, n, z),\ up(sx, y),\ \neg up(sssx, 0)$

9.f $h \leftarrow k(s0)$

Notice that clause 10 has been derived by: (i) comparing clause 8 and clause 9, and (ii) generalizing their mismatching terms y and sy to the term $s \ldots s0$ with $n\,(\geq 0)$ occurrences of s, that is, generalizing y and sy to the variable z such that $plus(y, n, z)$. This generalization avoids the introduction of an unbounded number of new definitions, and it makes the UFS strategy to halt. Indeed, by unfolding clause 10 w.r.t. $up(sx, y)$ and $\neg up(sssx, 0)$ and by performing simple goal replacement steps involving properties of $>$ and the equivalence $plus(sy, n, z) \leftrightarrow plus(y, sn, z)$, we get:

11. $k(n) \leftarrow x > z,\ plus(y, sn, z),\ up(sx, y),\ \neg up(sssx, 0)$

which we can be folded using clause 10 and we get:

11.f $k(n) \leftarrow k(sn)$.

Appendix 2. *Proof of Theorem* 3. Point (i) is proved as in Theorem 1. For Point (ii), during the derivation of the stratified program $F(f, \varphi)$ from $f \leftarrow \varphi$, for every derived statement S we preserve the following invariant: (ii.1) S is of the form $newp(x_1, \ldots, x_m) \leftarrow r_1(x_1) \wedge \ldots \wedge r_n(x_n) \wedge \gamma$, where $r_1(x_1), \ldots, r_n(x_n)$ are *all* tree atoms occurring as conjuncts at top level in the body of the statement, (ii.2) $freevars(S) \subseteq \{x_1, \ldots, x_n\}$, and (ii.3) γ is a function-free formula.

This invariant holds for the initial statement $f \leftarrow \varphi$, because φ is closed and it is function-free. This invariant is obviously maintained for the application of the rules B.1 and B.2.

Let us now consider the case of rule B.3*. Assume that we apply rule B.3* to a statement of the form $newp(x_1, \ldots, x_m) \leftarrow r_1(x_1) \wedge \ldots \wedge r_n(x_n) \wedge \gamma$, where $r_1(x_1), \ldots, r_n(x_n)$ are *all* tree atoms occurring as conjuncts at top level in the body of the statement, (ii.2) $freevars(S) \subseteq \{x_1, \ldots, x_n\}$, and (ii.3) γ is a function-free formula. Then we have that γ is of the form: $D[\neg(\varphi \wedge \psi)]$ and we derive the two statements: (S1) $A \leftarrow r_1(x_1) \wedge \ldots \wedge r_n(x_n) \wedge D[\neg newp1(y_1, \ldots, y_k)]$, and (S2) $newp1(y_1, \ldots, y_k) \leftarrow t_1(y_1) \wedge \ldots \wedge t_k(y_k) \wedge \varphi \wedge \psi$, where $\{y_1, \ldots, y_k\} = freevars(\varphi \wedge \psi) \subseteq \{x_1, \ldots, x_n\}$ and $\{t_1(y_1), \ldots, t_k(y_k)\} \subseteq \{r_1(x_1), \ldots, r_n(x_n)\}$. Thus, for the two statements (S1) and (S2) properties (ii.1), (ii.2), and (ii.3) hold.

The proof for the case of rule B.4* is similar to the one of rule B.3*. The proof for the case of rule B.5 is straightforward because, by construction, $D[\exists x\, \varphi]$ is of the form $D[\exists x\, (r(x) \wedge \delta)]$ for some tree atom $r(x)$ and some formula δ.

Finally, property (ii.4) of Theorem 3 derives from the fact that the LT transformation generates precisely one clause for every new predicate introduced.

Appendix 3. *Proof of Theorem* 4. We will prove that each iteration of the FOR loop terminates. To this aim we use the fact that the following invariant holds after each iteration of the FOR loop :

(I1) the derived program T is a conjunction of: (i) MR clauses, and (ii) definite clauses of the form:

$\delta 1.\ newp_0(t_1(x_1, \ldots), \ldots, t_m(\ldots, x_n)) \leftarrow p_1(x_1), \ldots, p_n(x_n)$ with $m, n \geq 0$

where: (i) $t_1, \ldots, t_m$ are function symbols, (ii) $x_1, \ldots, x_n$ are distinct variables, and (iii) $p_1, \ldots, p_n$ are predicates whose definitions in T are conjunctions of MR clauses.

Before the execution of FOR loop T is P, and thus (I1) holds.

Now we suppose that (I1) holds after the i-th iteration of the FOR loop and we show that it is preserved by the $(i+1)$-th iteration of the body of the FOR loop.

We first prove that after each iteration of the WHILE loop (†) the following invariant holds:

(I2) the derived *Pos* is a conjunction of clauses of the form:

$\delta 2.\ Newp \leftarrow r(x), p_1(x), \ldots, p_h(x), \neg p_{h+1}(x), \ldots, \neg p_i(x)$

where: (i) $r(x)$ is a tree atom, (ii) $p_1, \ldots, p_i$ are unary predicates whose definitions in T are conjunctions of MR clauses, and (iii) *Newp* is either a propositional atom or a unary atom of the form $newp(x)$.

Indeed, by Theorem 3 and Definition 3, when the WHILE loop (†) is initialized, *Pos* is a conjunction of clauses of the form:

$\delta 3.\ newp_0(x_1, \ldots, x_m) \leftarrow r_1(x_1), \ldots, r_n(x_n), p_1(y_1), \ldots, p_h(y_h),$

$$\neg p_{h+1}(y_{h+1}), \ldots, \neg p_i(y_i), \neg newp_1(y_{i+1}, \ldots, y_k), \ldots, \neg newp_j(y_u, \ldots, y_v)$$

where: (i) $x_1, \ldots, x_m$ are distinct variables occurring in $\{x_1, \ldots, x_n\}$, (ii) $y_1, \ldots, y_v$ are (not necessarily distinct) variables occurring in $\{x_1, \ldots, x_n\}$, (iii) the predicates $r_1, \ldots, r_n$ are tree predicates defined in T, (iv) the predicates $p_1, \ldots, p_h, p_{h+1}, \ldots, p_i$ are unary predicates whose definitions in T are conjunctions of MR clauses, and (v) $newp_1, \ldots, newp_j$ are predicates whose definitions in T are conjunctions of clauses of the form ($\delta 1$). By applying the BREADTH-FIRST UNFOLD(T, Pos, U) substrategy, followed by the TAUTOLOGY(T, U, R) substrategy, and by the BLOCK-DEFINE & FOLD (T, R, $Defs$, $OutClauses$, $NewDefs$) substrategy, we derive: (a) by folding, the clauses in $OutClauses$, which are of the form:

$\delta 4.\ newp_0(t_1(x_1, \ldots), \ldots, t_m(\ldots, x_n)) \leftarrow newp_1(x_1), \ldots, newp_n(x_n), newq_1, \ldots, newq_k$ with $m, n, k \geq 0$

and

(b) by the definition rule, the clauses in $NewDefs$, which are of the form ($\delta 2$). Thus, because of the assignment $Pos := NewDefs$, and since ($\delta 2$) is an instance of ($\delta 3$) (for $n = 1$ and $j = 0$), property (I2) holds after each iteration of the WHILE loop (†).

By Point (ii) of the BLOCK-DEFINE & FOLD substrategy, if a clause C occurs in Pos at a given iteration, then no clause which differs from C for the head predicate symbol and the variable names only, may occur in Pos at a subsequent iteration. Thus, the WHILE loop (†) terminates, because there is only a finite number of clauses that are of the form ($\delta 2$), modulo the head predicate symbol and the variable names. Moreover, because of the assignment $Out := Out \wedge OutClauses$, all clauses in Out are of the form ($\delta 4$), and thus, after the end of the WHILE loop (†) and the assignment $T := T \wedge Out' \wedge Neg$, program T is a conjunction of: (i) MR clauses, (ii) clauses of the form ($\delta 4$) which may include propositional definite clauses, and (iii) clauses in Neg, which are propositional clauses with negative literals only in their bodies (recall that, if a variable occurs in a negative literal in the body of a clause in P^i, then also a tree atom should occur in its body). Thus, all definitions of nullary predicates in T are conjunctions of propositional clauses.

All propositional clauses in T are replaced by a (possibly empty) conjunction of unit clauses by deleting useless predicates and then performing the sequence of UNFOLD and TAUTOLOGY steps as indicated in the WHILE loop (††). This WHILE loop terminates because, after the deletion of useless predicates, no nullary predicate depends on itself in T.

All propositional atoms occurring in the bodies of non-propositional clauses (such as $newq_1, \ldots, newq_k$ in ($\delta 4$) with $m > 0$) are eliminated by the unfolding (†††). Thus the iteration of the FOR loop terminates and the invariant (I1) holds. □

Appendix 4. *Proof of Theorem* 5. For a tree-typed clausal formula φ the LT transformation generates a conjunction of clauses $F(f, \varphi)$ of the form:

$\delta 1.\ f \leftarrow \neg g$

$\delta 2.\ g \leftarrow r_1(x_1), \ldots, r_n(x_n), p_1(y_1, \ldots, y_j), \ldots, p_h(y_k, \ldots, y_m), \neg p_{h+1}(y_{m+1}, \ldots, y_u), \ldots, \neg p_i(y_v, \ldots, y_z)$

where: (i) $y_1, \ldots, y_z$ are variables occurring in $\{x_1, \ldots, x_n\}$, (ii) the predicates $r_1, \ldots, r_n$ are tree predicates defined in R, and (iii) for $s = 1, \ldots, i$, the $Def^*(p_s, P)$ is a DL program. Thus, the predicates defined in $F(f, \varphi)$ have two levels only, and the FOR loop of the dUFS strategy is iterated only twice. The first iteration of the FOR loop starts off with $T = P \wedge R$, $Pos =$ clause $\delta 2$, and Neg being the empty conjunction.

After each iteration of the WHILE loop (†) we have that the derived Pos is a conjunction of clauses of the form:

$\delta 3.\ newp \leftarrow r_1(x_1), \ldots, r_{n'}(x_{n'}),$
$\quad p_1(y_1, \ldots, y_{j'}), \ldots, p_{h'}(y_{k'}, \ldots, y_{m'}), \neg p_{h'+1}(y_{m'+1}, \ldots, y_{u'}), \ldots, \neg p_{i'}(y_{v'}, \ldots, y_{z'})$

with $i' \leq i$, and the derived Out is a conjunction of propositional definite clauses.

Thus, the WHILE loop (†) terminates, because there is only a finite number of clauses that are of the form ($\delta 3$) with $i' \leq i$, modulo the head predicate symbol and the variable names.

After the termination of the WHILE loop (†) and the assignment $T := T \wedge Out' \wedge Neg$ program T is a conjunction of $P \wedge R$ and propositional definite clauses.

All propositional clauses in T are replaced by a (possibly empty) conjunction of unit clauses by deleting useless predicates and then performing the sequence of UNFOLD and TAUTOLOGY steps as indicated in the WHILE loop (††). In particular, the definition of g is replaced either by the unit clause g or by the empty conjunction.

The unfoldings (†††) are not performed.

At the second iteration of the FOR loop, the WHILE loop (†) is vacuously executed because Pos is the empty conjunction. Clause ($\delta 1$) is replaced by: (i) the unit clause f, if the definition of g is the empty conjunction, or (ii) by the empty conjuntion, otherwise. □