Program Derivation = Rules + Strategies*

Alberto Pettorossi† and Maurizio Proietti‡

†DISP, Università di Roma Tor Vergata, Roma, Italy. adp@iasi.rm.cnr.it
‡IASI-CNR, Roma, Italy. proietti@iasi.rm.cnr.it

Abstract. In a seminal paper [38] Prof. Robert Kowalski advocated the paradigm *Algorithm = Logic + Control* which was intended to characterize program executions. Here we want to illustrate the corresponding paradigm *Program Derivation = Rules + Strategies* which is intended to characterize program derivations, rather than executions. During program execution, the *Logic* component guarantees that the computed results are correct, that is, they are true facts in the intended model of the given program, while the *Control* component ensures that those facts are derived in an efficient way. Likewise, during program derivation, the *Rules* component guarantees that the derived programs are correct and the *Strategies* component ensures that the derived programs are efficient. In this chapter we will consider the case of logic programs with locally stratified negation and we will focus on the following three important methodologies for program derivation: program transformation, program synthesis, and program verification. Based upon the *Rules + Strategies* approach, we will propose a unified method for applying these three programming methodologies. In particular, we will present: (i) a set of rules for program transformation which preserve the perfect model semantics and (ii) a general strategy for applying the transformation rules. We will also show that we can synthesize correct and efficient programs from first order specifications by: (i) converting an arbitrary first order formula into a logic program with locally stratified negation by using a variant of the Lloyd-Topor transformation, and then (ii) applying our transformation rules according to our general strategy. Finally, we will demonstrate that the rules and the strategy for program transformation and program synthesis can also be used for program verification, that is, for proving first order properties of systems described by logic programs with locally stratified negation.

1 Introduction

Various models of computation were proposed since the early history of computing. Among others, we may recall the von Neumann machine for imperative languages, term rewriting for functional languages, and resolution for logical

* Published in: A. Kakas and F. Sadri (Eds). *Computational Logic: Logic Programming and Beyond* (*Essays in Honour of Robert A. Kowalski - Part I*), Lecture Notes in Artificial Intelligence 2407, Springer, 2002, pp. 273-309. Revised April 2009.

languages. In these three different language paradigms, people explored and analyzed different programming methodologies. In particular, in the area of logical languages, it was realized that both computing and programming can be viewed as a deductive activity.

The idea of computation as deduction may be traced back to the beginnings of the computation theory and recursive function theory, but it emerged clearly within the Theorem Proving community through the pioneering work of Robinson [62] and later, the paper by Kowalski [37], where the author proposed a particular deduction rule, namely, SLD-resolution, to compute in a logical theory consisting of Horn clauses. The deductive approach to computation was still considered to be not very practical at that time, but the situation changed when Warren [75] proposed a Prolog compiler based on SLD-resolution with performance comparable to that of the functional language Lisp. Efficiency is obtained by sacrificing correctness in some cases, but fortunately, that incorrectness turns out not to be a problem in practice.

The idea of programming and program development as a deduction from logical specifications to executable expressions in a formal setting, has its roots in the works by Burstall-Darlington and Manna-Waldinger [10,49] for functional languages and in the works by Clark *et al.*, Hogger, and Kowalski [11,12,32,39] for the case of logical languages. Similar ideas were proposed also in the case of imperative languages and one should mention, among others, the contributions of Dijkstra and Hoare (see, for instance, [21,31]).

In the paper [38] Kowalski proposes the motto: *Algorithm = Logic + Control*, to promote a separation of concern when writing programs: a concern for correctness in the *Logic* component, and a concern for efficiency in the *Control* component. This separation idea for program development goes back to the seminal paper by Burstall and Darlington [10]. The aim is to derive programs which are correct and efficient by applying transformation rules in a disciplined manner according to suitable strategies. In this case the *Logic* component consists of the transformation *rules*, such as unfolding and folding, which are correct because they preserve the semantics of interest, and the *Control* component consists of the *strategies* which direct the use of the rules so to derive efficient programs. Our motto, which can be viewed as an application of Kowalski's motto to the case of program development, is: *Program Derivation = Rules + Strategies*.

As we will illustrate in this chapter, our motto also indicates a way of understanding the relationship among various techniques for program development such as program synthesis, program reuse, and program verification. Some of these techniques based on rules and strategies, are described in [19,20,33,52].

The main objective of this chapter is to provide a unified view of: (i) program transformation, (ii) program synthesis, and (iii) program verification as deductive activities based on the unfolding/folding transformation rules and strategies. We consider the class of logic programs with locally stratified negation. The semantics of a program P in this class is given by its unique perfect model, denoted $M(P)$, which coincides with its unique stable model and its (total) well-founded model [2].

In our setting program transformation, synthesis, and verification can be formulated as follows.

Program Transformation. Given a program P and a goal G with free variables $X_1, \ldots, X_n$, we want to find a computationally efficient program T for a new n-ary predicate g such that, for all ground terms $t_1, \ldots, t_n$,

$$M(P) \models G\{X_1/t_1, \ldots, X_n/t_n\} \quad \text{iff} \quad M(T) \models g(t_1, \ldots, t_n) \qquad (\mathit{Transf})$$

Notice that our formulation of program transformation includes *program specialization* [27,33,44,47] which can be regarded as the particular case where G is an atom with instantiated arguments.

Program Synthesis. Given a program P and a *specification* of the form $g(X_1, \ldots, X_n) \leftrightarrow \varphi$, where: (i) φ is a first order formula with free variables $X_1, \ldots, X_n$, and (ii) g is a new n-ary predicate, we want to derive a computationally efficient program T for the predicate g such that, for all ground terms $t_1, \ldots, t_n$,

$$M(P) \models \varphi\{X_1/t_1, \ldots, X_n/t_n\} \quad \text{iff} \quad M(T) \models g(t_1, \ldots, t_n) \qquad (\mathit{Synth})$$

Program Verification. Given a program P and a *closed* first order formula φ, we want to check whether or not

$$M(P) \models \varphi \qquad (\mathit{Verif})$$

In order to get a unified view of program transformation, program synthesis, and program verification, let us first notice that each of these three tasks starts from a given program P and a first order formula. This formula, say γ, is: (i) the goal G in the case of program transformation, (ii) the formula φ of the specification $g(X_1, \ldots, X_n) \leftrightarrow \varphi$ in the case of program synthesis, and (iii) the closed first order formula φ in the case of program verification. Thus, we can provide a unified treatment of program transformation, program synthesis, and program verification, by viewing them as instances of the following general, two step method for program derivation, which takes as input a given program P and a first order formula γ.

The Unfold/Fold Method for Program Derivation.

We are given a locally stratified program P and a first order formula γ.

Step 1. We construct a conjunction of clauses, denoted by $\mathit{Cls}(g, \gamma)$ such that $P \wedge \mathit{Cls}(g, \gamma)$ is a locally stratified program and, for all ground terms $t_1, \ldots, t_n$,

$$M(P) \models \gamma\{X_1/t_1, \ldots, X_n/t_n\} \quad \text{iff} \quad M(P \wedge \mathit{Cls}(g, \gamma)) \models g(t_1, \ldots, t_n)$$

where $X_1, \ldots, X_n$ are the free variables of γ.

Step 2. We apply unfold/fold transformation rules which preserve the perfect model semantics and we derive a new program T such that, for all ground terms $t_1, \ldots, t_n$,

$$M(P \wedge \mathit{Cls}(g, \gamma)) \models g(t_1, \ldots, t_n) \ \text{ iff } \ M(T) \models g(t_1, \ldots, t_n)$$

The derivation of program T is made according to a transformation strategy which guides the application of the rules.

Let us now briefly explain how this general unfold/fold method for program derivation will be instantiated to three specific methods for program transformation, program synthesis, and program verification. More details and examples will be given in Sections 2, 3, and 4.

Among the tasks of program transformation, program synthesis, and program verification, the one which has the most general formulation is program synthesis, because the formula φ of a specification is *any* first order formula, whereas the inputs for program transformation and program verification consist of a *goal* (that is, a conjunction of literals) and a *closed* first order formula, respectively.

A method for program synthesis can be obtained from the general unfold/fold method for program derivation in a straightforward way by taking γ as the formula φ of the specification $g(X_1, \ldots, X_n) \leftrightarrow \varphi$. In Section 3 we will see how the conjunction of clauses $Cls(g, \varphi)$ can be constructed by using a suitable variant of the *Lloyd-Topor transformation* [46]. Moreover, we will propose (see Section 2) a general transformation strategy for deriving a suitable program T from program $P \wedge Cls(g, \varphi)$ as required by Step 2 of the unfold/fold method. From the fact that our variant of the Lloyd-Topor transformation and the unfold/fold transformation rules preserve the perfect model semantics, it follows that the equivalence (*Synth*) indeed holds for this program T.

Similarly, if we consider our general unfold/fold method for program derivation in the case where γ is the goal G, then we derive a program T which satisfies the relation (*Transf*), and thus, in this case the general method becomes a method for program transformation.

Finally, program verification can be viewed as an instance of our general unfold/fold method in the case where γ is the closed first order formula φ. In particular, the conjunction of clauses $Cls(g, \varphi)$ can be constructed as in the case of program synthesis by starting from the specification $g \leftrightarrow \varphi$. Then, one can prove that $M(P) \models \varphi$ holds by applying Step 2 of our method for program derivation and obtaining a program T which includes the clause $g \leftarrow$.

The contributions of this chapter are the following ones. (i) We describe in some detail our general, two step method based on rules and strategies, for the unified treatment of program transformation, synthesis, and verification, and through some examples, we show that our method is effective for each of these tasks. (ii) We establish the correctness of the transformation rules by giving sufficient conditions for the preservation of perfect model semantics. These correctness results extend results already published in the literature [70]. In particular, we take into consideration also the unfolding and folding rules w.r.t. negative literals, and these rules are crucial in the examples we will present. (iii) We outline a general strategy for the application of the transformation rules and we demonstrate that various techniques for rather different tasks, such as program transformation, program synthesis, and program verification, can all be realized by that single strategy.

The plan of the chapter is as follows. In Section 2 we present a set of transformation rules for locally stratified programs and we give sufficient conditions which ensure their correctness w.r.t. the perfect model semantics. We also present

our general strategy for the application of the transformation rules. In Section 3 we present the instance of our two step unfold/fold method for the synthesis of logic programs from specifications provided by first order formulas. In Section 4 we show that also program verification can be performed using our two step method.

2 Transformation Rules and Strategies for Locally Stratified Logic Programs

In this section we recall the basic concepts of locally stratified programs and perfect model semantics. We then present the transformation rules which we use for program transformation, and we provide a sufficient condition which ensures that these rules preserve the perfect model semantics. We also outline a general strategy for applying the transformation rules.

2.1 Preliminaries: Syntax and Semantics of Stratified Logic Programs

We recall some basic definitions and we introduce some terminology and notation concerning general logic programs and their semantics. In particular, we will recall the definitions of *locally stratified* logic programs and their *perfect models*. For notions not defined here the reader may refer to [2,46,59].

Given a *first order language* $\mathcal{L}$, its *formulas* are constructed out of *variables*, *function* symbols, *predicate* symbols, *terms*, *atomic formulas* (also called *atoms*), the formula *true*, the connectives $\neg$ and $\wedge$, and the quantifier $\exists$ (see, for instance, [2,46]). We feel free to write formulas using also the symbols *false*, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\forall$, but we regard them as abbreviations of the equivalent formulas written using the symbols *true*, $\neg$, $\wedge$, and $\exists$ only. Following the usual logic programming convention, we use upper case letters for variables and lower case letters for function and predicate symbols.

A *literal* is an atom (i.e., a *positive* literal) or a negated atom (i.e., a *negative* literal). A *goal* G is a conjunction of n (≥ 0) literals.

General logic programs, simply called *logic programs*, or *programs*, are first order formulas defined as follows. A *program* is a conjunction of *clauses,* each of which is of the form: $G \rightarrow H$, where G is a goal and H is an atom different from *true* and *false*. Normally a clause will be written as $H \leftarrow G$. The atom H is called the *head* of the clause, denoted by $hd(C)$, and the goal G is called the *body* of the clause, denoted by $bd(C)$. A clause $H \leftarrow G$ where G is the empty conjunction *true*, is said to be a *unit* clause and it is written as $H \leftarrow$. When writing goals, clauses, and programs, we also denote conjunctions by using comma ',' instead of $\wedge$. Thus, usually, a goal will be written as $L_1, \ldots, L_n$, where the L_i's are literals, a clause will be written as $H \leftarrow L_1, \ldots, L_n$, and a program will be written as $C_1, \ldots, C_n$, where the C_i's are clauses. When writing programs we will also feel free to omit commas between clauses, if no confusion arises.

A clause is said to be *definite* iff no negated atom occurs in its body. A *definite* program is a conjunction of definite clauses.

Given a term t we denote by $vars(t)$ the set of all variables occurring in t. Similar notation will be used for the variables occurring in 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 variables of φ which have a free occurrence in φ. A clause C is said to be *ground* iff no variable occurs in it. We may freely rename the variables occurring in clauses, and the process of renaming the variables of a clause by using new variables, is called *renaming apart* [46].

The *definition* of a predicate p in a program P, denoted by $Def(p, P)$, is the conjunction of the 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 definition of every predicate on which p depends in P. We say that a predicate p *depends on existential variables* in a program P iff in $Def^*(p, P)$ there exists a clause C whose body has an existential variable.

The set of *useless* predicates of a program P is the maximal set U of the 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 following program:

$p \leftarrow q, r$
$q \leftarrow p$
$r \leftarrow$

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, say $\neg A_i$, then $\sigma(H) > \sigma(A_i)$. 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 every ground atom is in a stratum which is greater than 0, so that P_0 may be assumed to be the empty conjunction of clauses.

An *Herbrand interpretation* is a subset of $B_{\mathcal{L}}$. We say that a closed first order formula φ is true in an Herbrand interpretation I, written as $I \models \varphi$, iff one of the following cases holds: (i) φ is the formula *true*, (ii) φ is a ground atom A which is in I, (iii) φ is $\neg\varphi_1$ and φ_1 is not true in I, (iv) φ is $\varphi_1 \wedge \varphi_2$ and both

φ_1 and φ_2 are true in I, (v) φ is $\exists X\, \varphi_1$ and there exists a ground term t such that $\varphi_1\{X/t\}$ is true in I.

Given a formula φ and an Herbrand interpretation I, if it is not the case that $I \models \varphi$, we say that φ is false in I and we write $I \not\models \varphi$.

The *perfect model* $M(P)$ of a program P which is locally stratified w.r.t. a stratification σ, is the Herbrand interpretation defined as the subset $\bigcup_{\alpha \in W} M_\alpha$ of $B_{\mathcal{L}}$, where for every ordinal α in W, the set M_α is constructed as follows:
(1) M_0 is the empty set, and
(2) if $\alpha > 0$, M_α is the *least Herbrand model* [46] of the definite program derived from P_α as follows: (i) every literal L in stratum τ, with $\tau < \alpha$, in the body of a clause in P_α is deleted iff $M_\tau \models L$, and (ii) every clause C in P_α is deleted iff in $bd(C)$ there exists a literal L in stratum τ, with $\tau < \alpha$ such that $M_\tau \not\models L$.

For a locally stratified program P, with $vars(P) = \{X_1, \ldots, X_n\}$, we have that $M(P) \models \forall X_1, \ldots, X_n\, P$.

Our construction of the perfect model differs from the construction presented in [2,59], but as the reader may verify, the two constructions yield the same model.

Recall that 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 [2]. Indeed, as already mentioned, 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*.

2.2 Unfold/Fold 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.

For the application of the transformation rules we divide the predicate symbols of the language into two classes: (i) *basic* predicates and (ii) *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 every basic atom is in a strictly smaller stratum w.r.t. every non-basic atom, and thus, in any given program no basic predicate depends on a non-basic one. Our partition of the set of predicates into basic or non-basic predicates is arbitrary and it may be different for different program derivations.

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

We consider a set *Preds* of predicates of interest. We also consider, for $0 \leq k \leq n$, the conjunction $Defs_k$ of the clauses introduced by using the following rule R1 during the whole transformation sequence $P_0, \ldots, P_k$.

R1. Definition Introduction Rule. We get the new program P_{k+1} by adding to program P_k a conjunction of m clauses of the form:

$$\begin{cases} newp(X_1, \ldots, X_s) \leftarrow Body_1 \\ \quad \ldots \\ newp(X_1, \ldots, X_s) \leftarrow Body_m \end{cases}$$

such that:

(i) the predicate *newp* is a non-basic predicate which does not occur in $P_0 \wedge Defs_k$,
(ii) $X_1, \ldots, X_s$ are distinct variables occurring in $Body_1, \ldots, Body_m$, and
(iii) every predicate occurring in $Body_1, \ldots, Body_m$ also occurs in P_0.

R2. Definition Elimination Rule. By *definition elimination* w.r.t. *Preds*, from program P_k we derive the new program P_{k+1} by deleting the definitions of all predicates on which no predicate belonging to *Preds* depends in P_k.

R3. 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 $hd(D_1)$, ..., $hd(D_m)$, with most general unifiers $\vartheta_1, \ldots, \vartheta_m$, respectively, and
2. C_i is the clause $(H \leftarrow G_1, bd(D_i), G_2)\vartheta_i$, for $i = 1, \ldots, m$.

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

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 the new program P_{k+1} by deleting clause C.

R4. Negative Unfolding Rule. Let C be a renamed apart clause in P_k of the form: $H \leftarrow G_1, \neg A, G_2$. 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 $\vartheta_1, \ldots, \vartheta_m$, respectively. Assume that:

1. $A = hd(D_1)\vartheta_1 = \cdots = hd(D_m)\vartheta_m$, 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, $\neg(bd(D_1)\vartheta_1 \vee \ldots \vee bd(D_m)\vartheta_m)$, 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* clause C *w.r.t.* $\neg A$ we derive the clauses $C_1, \ldots, C_r$, where C_i is the clause $H \leftarrow Q_i$, for $i = 1, \ldots, r$. From program P_k we derive the new program P_{k+1} by replacing C with $C_1, \ldots, C_r$.

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.

R5. Positive Folding Rule. Let $C_1, \ldots, C_m$ be renamed apart clauses in P_k and $D_1, \ldots, D_m$ be the definition of a predicate in $Defs_k$. For $i = 1, \ldots, m$, let C_i be of the form: $H \leftarrow G_1, B_i, G_2$. Suppose that there exists a substitution ϑ such that, for $i = 1, \ldots, m$ the following conditions hold:
(1) $B_i = bd(D_i)\vartheta$, and
(2) for every variable X in the set $vars(D_i) - vars(hd(D_i))$, we have that $X\vartheta$ is a variable which occurs neither in $\{H, G_1, G_2\}$ nor in the term $Y\vartheta$, for any variable Y occurring in $bd(D_i)$ and different from X.
By *folding* clauses $C_1, \ldots, C_m$ using clauses $D_1, \ldots, D_m$ we derive the clause E: $H \leftarrow G_1, hd(D_1)\vartheta, G_2$. From program P_k we derive the new program P_{k+1} by replacing $C_1, \ldots, C_m$ with E.
Notice that by definition of rule R1, we have that $hd(D_1) = \ldots = hd(D_m)$.

R6. Negative Folding Rule. Let C be a renamed apart clause in P_k and let *newp* be a predicate in $Defs_k$ whose definition consists of a single clause D. Let C be of the form: $H \leftarrow G_1, \neg A, G_2$. Suppose that the following conditions hold:
(1) $A = bd(D)\vartheta$, for some substitution ϑ, and
(2) $vars(hd(D)) = vars(bd(D))$.
By *folding* clause C w.r.t. $\neg A$ using clause D we derive the clause E: $H \leftarrow G_1, \neg hd(D)\vartheta, G_2$. From program P_k we derive the new program P_{k+1} by replacing C with E.

R7. Tautology Rule. We derive the new program P_{k+1} by replacing in P_k a conjunction of clauses γ_1 with a new conjunction of clauses γ_2, according to the following rewritings $\gamma_1 \Rightarrow \gamma_2$, where H and A, denote atoms, G, G_1, G_2, G_3, and G_4 denote goals, and C_1, C_2 denote clauses:

(1)	$H \leftarrow A, \neg A, G$		$\Rightarrow$	$true$
(2)	$H \leftarrow H, G$		$\Rightarrow$	$true$
(3)	$H \leftarrow G_1, G_2, G_3, G_4$		$\Rightarrow$	$H \leftarrow G_1, G_3, G_2, G_4$
(4)	$H \leftarrow A, A, G$		$\Rightarrow$	$H \leftarrow A, G$
(5)	$H \leftarrow G_1,$	$H \leftarrow G_1, G_2$	$\Rightarrow$	$H \leftarrow G_1$
(6)	$H \leftarrow A, G_1, G_2,$	$H \leftarrow \neg A, G_1$	$\Rightarrow$	$H \leftarrow G_1, G_2, \quad H \leftarrow \neg A, G_1$
(7)	$C_1,\ C_2$		$\Rightarrow$	$C_2,\ C_1$

R8. Clause Deletion Rule. We derive the new program P_{k+1} by removing from P_k the definitions of the useless predicates of P_k.

R9. Basic Goal Replacement Rule. Let us consider $r\,(> 0)$ renamed apart clauses in P_k of the form: $H \leftarrow G_1, Q_1, G_2, \ldots, H \leftarrow G_1, Q_r, G_2$. Suppose that, for some goals $R_1, \ldots, R_s$, we have:

$M(P_0) \models \forall X_1 \ldots X_u\, (\exists Y_1 \ldots Y_v\, (Q_1 \vee \ldots \vee Q_r) \leftrightarrow \exists Z_1 \ldots Z_w\, (R_1 \vee \ldots \vee R_s))$

where:
(i) $\{Y_1, \ldots, Y_v\} = vars(Q_1, \ldots, Q_r) - vars(H, G_1, G_2)$,
(ii) $\{Z_1, \ldots, Z_w\} = vars(R_1, \ldots, R_s) - vars(H, G_1, G_2)$, and

(iii) $\{X_1,\ldots,X_u\} = vars(Q_1,\ldots,Q_r,R_1,\ldots,R_s) - \{Y_1,\ldots,Y_v,Z_1,\ldots,Z_w\}$.
Suppose also that $R_1,\ldots,R_s$ 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 the clauses $H \leftarrow G_1,Q_1,G_2,\ \ldots,\ H \leftarrow G_1,Q_r,G_2$ with the clauses $H \leftarrow G_1,R_1,G_2,\ \ldots,$ $H \leftarrow G_1,R_s,G_2$.

We assume that the *equality predicate* $=$ is a basic predicate which is defined in each program by the single clause $X=X \leftarrow$.

R10. Equality Introduction and Elimination. Let C be a clause of the form $(H \leftarrow Body)\{X/t\}$, such that the variable X does not occur in t and let D be the clause: $H \leftarrow X=t,\ Body$.
By *equality introduction* we derive clause D from clause C. By *equality elimination* we derive clause C from clause D.
If C occurs in P_k then we derive the new program P_{k+1} by replacing C with D.
If D occurs in P_k then we derive the new program P_{k+1} by replacing D with C.

The transformation rules from rule R1 to rule R10 we have introduced above, will collectively be called *unfold/fold transformation rules*.

Theorem 1. [Correctness of the Unfold/fold Transformation Rules] *Let $P_0,\ldots,P_n$ be a transformation sequence and Preds be a set of predicates of interest. Let us assume that:*
(1) *during the construction of $P_0,\ldots,P_n$, each clause introduced by the definition introduction rule and used for folding, is unfolded (before or after its use for folding) w.r.t. a non-basic positive literal in its body, and*
(2) *during the transformation sequence $P_0,\ldots,P_n$, either the definition elimination rule is never applied or it is applied at the end of that sequence.*
Then, for all ground atoms A with predicate in Preds, $M(P_0 \wedge Defs_n) \models A$ iff $M(P_n) \models A$.

Notice that the statement obtained from Theorem 1 by replacing 'positive unfolding' by 'negative unfolding' is not a theorem as shown by the following example.

Example 1. Let P_0 be the program:
1. $p \leftarrow \neg q(X)$
2. $q(X) \leftarrow q(X)$
3. $q(X) \leftarrow r$

By negative unfolding w.r.t. $\neg q(X)$, from clause 1 we get the following clause 4:
4. $p \leftarrow \neg q(X), \neg r$

Then by folding clause 4 w.r.t. $\neg q(X)$, we get the following clause 5:
5. $p \leftarrow p, \neg r$

The final program P_1 consists of clauses 2, 3, and 5. We have that $M(P_0) \models p$, while $M(P_1) \models \neg p$. □

Our presentation of the transformation rules essentially follows the style of Tamaki and Sato who first introduced the unfold/fold transformation rules in the case of definite programs [74] and proved their correctness w.r.t. the least Herbrand model semantics. Among the rules presented in this section, the following

ones were introduced by Tamaki and Sato in [74] (actually, their presentation was a bit different): R1 restricted to $m = 1$, R3, R5 restricted to $m = 1$, R7 restricted to definite clauses, R8, R9 restricted to $r = s = 1$, and R10. Thus, some of our rules may be considered an extension of those in [74].

One of the most relevant features of Tamaki and Sato's rules is that their correctness is ensured by conditions on the construction of the transformation sequences similar to Condition (1) of Theorem 1.

A subset of Tamaki and Sato's rules, namely R3 (positive unfolding) and R5 (positive folding) with $m = 1$, has been extended to general logic programs by Seki and proved correct w.r.t. various semantics, including the perfect model semantics [70,71].

An extension of Seki's rules has been recently proposed by Roychoudhury *et al.* in [63]. In particular, they drop the restrictions that we can fold one clause only and the clauses used for folding are not recursive. The correctness of this extension of Seki's rules is ensured by a rather sophisticated condition which, in the case where recursive clauses cannot be used for folding, is implied by Condition (1) of Theorem 1.

Thus, the positive folding rule presented here is less powerful than the folding rule of [63], because we can only fold using clauses taken from $Defs_k$, and according to the definition introduction rule R1, we cannot introduce recursive clauses in $Defs_k$. However, our set of rules includes the negative unfolding (R4), the negative folding (R6), and the basic goal replacement rules (R9) which are not present in [63], and these rules are indeed very useful in practice and they are needed for the program derivation examples given in the next sections. We believe that we can easily incorporate the more powerful folding rule of [63] into our set of rules, but for reasons of simplicity, we stick to our version of the positive folding rule which has much simpler applicability conditions.

2.3 A Transformation Method

Now we outline our two step method for program transformation based on: (i) the unfold/fold transformation rules presented in Section 2.2, and (ii) a simple, yet powerful strategy, called *unfold/fold transformation strategy*, for guiding the application of the transformation rules. This method is an instance of the general unfold/fold method described in Section 1. Actually, our strategy is not fully specified, in the sense that many transformation steps can be performed in a nondeterministic way, and thus, we cannot prove that it improves efficiency in all cases. However, our strategy can be regarded as a generalization and adaptation to the case of general logic programs of a number of efficiency improving transformation strategies for definite programs presented in the literature, such as strategies for specializing programs, achieving tail recursion, avoiding intermediate data structures, avoiding redundant computations, and reducing nondeterminism (see [53] for a survey). Through some examples, we will indeed show that program efficiency can be improved by applying our unfold/fold transformation strategy.

The Unfold/Fold Transformation Method.
Given a locally stratified program P and a goal G such that $vars(G) = \{X_1, \ldots, X_n\}$, our transformation method consists of two steps as follows.

Step 1. We introduce a new n-ary predicate, say g, not occurring in $\{P, G\}$ and we derive a conjunction $Cls(g, G)$ of clauses such that $P \wedge Cls(g, G)$ is a locally stratified program and, for all ground terms $t_1, \ldots, t_n$,

(1) $M(P) \models G\{X_1/t_1, \ldots, X_n/t_n\}$ iff $M(P \wedge Cls(g, G)) \models g(t_1, \ldots, t_n)$.

Step 2. From the program P, the conjunction $Cls(g, G)$ of clauses, and a set of equivalences to be used for rule R9, by applying the unfold/fold transformation strategy described below, we derive a program T such that, for all ground terms $t_1, \ldots, t_n$,

(2) $M(P \wedge Cls(g, G)) \models g(t_1, \ldots, t_n)$ iff $M(T) \models g(t_1, \ldots, t_n)$

and thus, the relation (*Transf*) considered in the Introduction holds.

Clearly, a program T which satisfies (2) is $P \wedge Cls(g, G)$ itself. However, most often we are not interested in such trivial derivation because, as already mentioned, we look for an efficient program T which satisfies (2).

Now let us look at the above two steps of our transformation method in more detail.
Step 1 is performed by first introducing the clause C_1: $g(X_1, \ldots, X_n) \leftarrow G$ and then replacing this clause by a conjunction $Cls(g, G)$ of clauses as follows: for each non-basic negative literal $\neg p(u_1, \ldots, u_m)$ in G such that p depends on existential variables in P,

(i) we introduce the clause D: $new(Y_1, \ldots, Y_k) \leftarrow p(u_1, \ldots, u_m)$, where $vars(p(u_1, \ldots, u_m)) = \{Y_1, \ldots, Y_k\}$, and

(ii) we fold clause $g(X_1, \ldots, X_n) \leftarrow G$ w.r.t. $\neg p(u_1, \ldots, u_m)$ using D.

For instance, in Example 2 below, from the initial goal

G: $word(W), \neg derive([s], W)$

we introduce the clause: $g(W) \leftarrow word(W), \neg derive([s], W)$, because the definition of the predicate *derive* includes clause 3 which has the existential variables B and T. At the end of Step 1, we derive the following two clauses:

16. $g(W) \leftarrow word(W), \neg new1(W)$
17. $new1(W) \leftarrow derive([s], W)$

Step 1 is motivated by the fact that it is often useful, for reasons of efficiency, to transform the definitions of the predicates occurring in negative literals, if these definitions include clauses with existential variables. Indeed, since the unfolding w.r.t. a negative literal, say $\neg p(u_1, \ldots, u_m)$, is defined only if the clauses whose heads unify with $p(u_1, \ldots, u_m)$, have no existential variables, it is desirable to transform $Def^*(p, P) \wedge (new1(Y_1, \ldots, Y_k) \leftarrow p(u_1, \ldots, u_m))$ so to derive a new definition for the predicate $new1$ whose clauses do not have existential variables. Then, this new definition of $new1$ can be used for performing unfolding steps

w.r.t. literals of the form $\neg new1(u_1, \ldots, u_m)$ and it may also allow more effective transformations of the clauses where $new1$ occurs.

Step 2 consists in applying the unfold/fold transformation strategy which we describe below. This strategy constructs n program transformation sequences $S^1, \ldots, S^n$, where for $i = 1, \ldots, n-1$, the final program of the sequence S^i coincides with the initial program of the sequence S^{i+1}. Each transformation sequence corresponds to a *level* which is induced by the construction of the conjunction $Cls(g, G)$ of clauses. We will define these levels according to the following notion of *level mapping* [46].

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. Given a level mapping m, the* level *of the predicate p is the number assigned to p by m.*

Given a program P and a goal G, by construction there exists a level mapping of $Cls(g, G)$ such that: (1) the conjunction $Cls(g, G)$ can be partitioned into K subconjunctions: $D^1, \ldots, D^K$, such that $Cls(g, G) = D^1 \wedge \ldots \wedge D^K$, and, for $i = 1, \ldots, K$, the subconjunction D^i of clauses consists of all clauses in $Cls(g, G)$ whose head predicates are at level i, (2) for $i = 1, \ldots, K$ and for each clause $p(\ldots) \leftarrow B$ in D^i, the level of each predicate symbol in the goal B is strictly smaller than the level of p, (3) the predicate g is at the highest level K, and (4) all predicates of $Cls(g, G)$ which occur in P, are at level 0.

The reader may notice that, according to our definition of Step 1 above, K is at most 2. However, we have considered the case of an arbitrary value of K, because this will be appropriate when in Sections 3 and 4 below we consider program synthesis and program verification, respectively.

For the construction of each transformation sequence S^i, for $i = 1, \ldots, n-1$, our unfold/fold transformation strategy uses the following three *subsidiary strategies*: (i) UNFOLD(P, Q), (ii) TAUTOLOGY-REPLACE$(Laws, P, Q)$, and (iii) DEFINE-FOLD$(Defs, P, Q \wedge NewDefs)$.

(i) Given a program P, UNFOLD(P, Q) specifies how to derive a new program Q by performing positive and negative unfolding steps (rules R3 and R4).

(ii) Given a program P and a set *Laws* of equivalences needed for the application of the goal replacement rule, TAUTOLOGY-REPLACE$(Laws, P, Q)$ specifies how to derive a new program Q by applying the tautology, goal replacement, and equality introduction and elimination rules (rules R8, R9, and R10).

(iii) Given a program P and a conjunction *Defs* of predicate definitions, DEFINE-FOLD$(Defs, P, Q \wedge NewDefs)$ specifies how to derive a new program $Q \wedge NewDefs$ by introducing a new conjunction *NewDefs* of predicate definitions and performing folding steps using clauses occurring in $Defs \wedge NewDefs$ (rules R1, R5, and R6).

The effectiveness of the unfold/fold transformation strategy depends upon the choice of these subsidiary strategies, and much research, mostly in the case of definite programs, has been devoted to devise subsidiary strategies which allow us to derive very efficient programs [53]. For instance, the introduction of new predicate definitions, also called *eureka definitions*, influences the efficiency

of the derived programs. Various techniques have been proposed for determining the suitable eureka definitions to be introduced. Here we only want to mention that it is often useful to introduce new predicates whose definition clauses have bodies which are: (i) instances of atoms, so to perform *program specialization*, (ii) conjunctions of literals that share variables, so to derive programs that simultaneously perform the computations relative to several literals, and (iii) disjunctions of goals, so to derive programs with reduced nondeterminism, because they simultaneously perform the computations relative to several alternative goals.

We omit here the detailed description of the UNFOLD, TAUTOLOGY-REPLACE, and DEFINE-FOLD subsidiary strategies. We will see them in action in the examples given below. Here is our Unfold/Fold Transformation Strategy.

The Unfold/Fold Transformation Strategy.

Input: (i) a program P, (ii) a conjunction $Cls(g, G)$ of clauses constructed as indicated at Step 1, and (iii) a set *Laws* of equivalences for the application of rule R9. These equivalences are assumed to hold in $M(P \wedge Cls(g, G))$.
Output: A program T such that, for all ground terms $t_1, \ldots, t_n$,

$M(P \wedge Cls(g, G)) \models g(t_1, \ldots, t_n)$ iff $M(T) \models g(t_1, \ldots, t_n)$.

Let us partition $Cls(g, G)$ into K subconjunctions: $D^1, \ldots, D^K$, as indicated in Step 2 above.

$T := P$;
FOR $i = 1, \ldots, K$ **DO**
We construct a transformation sequence S^i as follows.
$Defs := D^i$; $\quad InDefs := D^i$;

By the definition introduction rule we add the clauses of *InDefs* to T, thereby obtaining $T \wedge InDefs$.

WHILE *InDefs* is not the empty conjunction **DO**

(1) UNFOLD($T \wedge InDefs$, $T \wedge U$): From program $T \wedge InDefs$ we derive $T \wedge U$ by a finite sequence of applications of the positive and negative unfolding rules to the clauses in *InDefs*.
(2) TAUTOLOGY-REPLACE(*Laws*, $T \wedge U$, $T \wedge R$): 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(*Defs*, $T \wedge R$, $T \wedge F \wedge NewDefs$): From program $T \wedge R$ we derive $T \wedge F \wedge NewDefs$ by: (3.i) a finite sequence of applications of the definition introduction rule by which we add to $T \wedge R$ the (possibly empty) conjunction *NewDefs* of clauses, followed by (3.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 all non-basic predicates occurring in the body of a clause which has been derived by folding, are defined in $Defs \wedge NewDefs$.
$T := T \wedge F$; $\quad Defs := Defs \wedge NewDefs$; $\quad InDefs := NewDefs$

END WHILE;
Delete from T the definitions of useless predicates.
END FOR
Delete from T the definitions of the predicates upon which the predicate g does not depend.

The unfold/fold transformation strategy is correct in the sense that for all ground terms $t_1, \ldots, t_n$, $M(P \wedge \mathit{Cls}(g, G)) \models g(t_1, \ldots, t_n)$ iff $M(T) \models g(t_1, \ldots, t_n)$, if each clause used for folding when executing the DEFINE-FOLD subsidiary strategy is unfolded w.r.t. a positive literal during an execution of the UNFOLD subsidiary strategy. If this condition is satisfied, then the correctness of our transformation strategy w.r.t. the perfect model semantics follows from the Correctness Theorem 1 of Section 2.2.

Notice that the unfold/fold transformation strategy may not terminate, because during the execution of the WHILE loop, *InDefs* may never become the empty conjunction.

Notice also that the iterations of our strategy over the various levels from 1 to K, correspond to the construction of the perfect model of program $P \wedge \mathit{Cls}(g, G)$ derived at the end of Step 1. This construction is done, so to speak, level by level moving upwards and starting from the perfect model of the program P whose predicates are assumed to be at level 0.

Let us now present an example of program derivation using our unfold/fold transformation method.

Example 2. Complement of a context-free language. Let us consider the following program *CF* for deriving a word of a given context-free language over the alphabet $\{a, b\}$:

Program *CF*

1. $\mathit{derive}([\,],[\,]) \leftarrow$
2. $\mathit{derive}([A|S],[A|W]) \leftarrow \mathit{terminal}(A),\ \mathit{derive}(S,W)$
3. $\mathit{derive}([A|S],W) \leftarrow \mathit{nonterminal}(A),\ \mathit{production}(A,B),\ \mathit{append}(B,S,T),\ \mathit{derive}(T,W)$
4. $\mathit{terminal}(a) \leftarrow$
5. $\mathit{terminal}(b) \leftarrow$
6. $\mathit{nonterminal}(s) \leftarrow$
7. $\mathit{nonterminal}(x) \leftarrow$
8. $\mathit{production}(s,[a,x,b]) \leftarrow$
9. $\mathit{production}(x,[\,]) \leftarrow$
10. $\mathit{production}(x,[a,x]) \leftarrow$
11. $\mathit{production}(x,[a,b,x]) \leftarrow$
12. $\mathit{append}([\,],A,A) \leftarrow$
13. $\mathit{append}([A|B],C,[A|D]) \leftarrow \mathit{append}(B,C,D)$
14. $\mathit{word}([\,]) \leftarrow$
15. $\mathit{word}([A|W]) \leftarrow \mathit{terminal}(A),\ \mathit{word}(W)$

The relation $\mathit{derive}([s],W)$ holds iff the word W can be derived from the *start symbol* s using the following productions of the grammar defining the given context-free language (see clauses 8–11):

$s \to a\,x\,b \qquad x \to \varepsilon \qquad x \to a\,x \qquad x \to a\,b\,x$

The terminal symbols are a and b (see clauses 4 and 5), the nonterminal symbols are s and x (see clauses 6 and 7), the empty word ε is represented as the empty list $[\,]$, and words in $\{a, b\}^*$ are represented as lists of a's and b's.

In general, the relation $derive(L, W)$ holds iff L is a sequence of terminal or nonterminal symbols from which the word W can be derived by using the productions.

We would like to derive an efficient program for an initial goal G of the form: $word(W)$, $\neg derive([s], W)$, which is true in $M(CF)$ iff W is a word which is *not* derived by the given context-free grammar. We perform our program derivation as follows.

Step 1. We derive the two clauses:

16. $g(W) \leftarrow word(W), \neg new1(W)$
17. $new1(W) \leftarrow derive([s], W)$

as indicated in the description of the Step 1 above. The predicate g is at level 2 and the predicate $new1$ is at level 1. All predicates in program CF are at level 0.

Step 2. We apply our unfold/fold transformation strategy. During the application of this strategy we never apply rules R7, R8, R9, and R10. Thus, we use neither the TAUTOLOGY-REPLACE subsidiary strategy nor the deletion of useless predicates. We have that $K = 2$, $D^1 = \{\text{clause } 17\}$, and $D^2 = \{\text{clause } 16\}$.

Level 1. Initially program T is CF. We start off by adding clause 17 to T. Both *Defs* and *InDefs* consist of clause 17 only. We will perform four iterations of the body of the WHILE loop of our strategy before *InDefs* becomes the empty conjunction, and then we exit the WHILE loop. Here we show only the first and fourth iterations.

First Iteration.

UNFOLD. By unfolding, from clause 17 we get:

18. $new1([a|A]) \leftarrow derive([x, b], A)$

DEFINE-FOLD. We introduce the following clause

19. $new2(A) \leftarrow derive([x, b], A)$

and by folding clause 18 using clause 19 we get:

20. $new1([a|A]) \leftarrow new2(A)$

which is added to program T.

At the end of the first iteration T is made out of the clauses of CF together with clause 20, *Defs* consists of clauses 17 and 19, and *InDefs* consists of clause 19. Since *InDefs* is not empty, we continue by iterating the execution of the body of the WHILE loop of our strategy.

During the second and third iteration of the WHILE loop, by the definition rule we introduce the following clauses:

21. $new3(A) \leftarrow derive([\,], A)$

22. $new4(A) \leftarrow derive([x, b], A)$
23. $new4(A) \leftarrow derive([b, x, b], A)$
24. $new5(A) \leftarrow derive([\,], A)$
25. $new5(A) \leftarrow derive([x, b], A)$

At the beginning of the fourth iteration *InDefs* is made out of clauses 24 and 25 only. Here are the details of this fourth iteration which is the last one.

Fourth Iteration.

UNFOLD. By unfolding, from clauses 24 and 25 we get:

26. $new5([\,]) \leftarrow$
27. $new5([b|A]) \leftarrow derive([\,], A)$
28. $new5([a|A]) \leftarrow derive([x, b], A)$
29. $new5([a|A]) \leftarrow derive([b, x, b], A)$

DEFINE-FOLD. We fold clause 27 using clause 21, and clauses 28 and 29 using clauses 24 and 25, and we get:

30. $new5([b|A]) \leftarrow new3(A)$
31. $new5([a|A]) \leftarrow new4(A)$

No new definition is introduced during this fourth iteration. Thus, *InDefs* is empty and we exit from the WHILE loop. The transformation strategy terminates for level 1, and program T is made out of CF together with the following clauses:

20. $new1([a|A]) \leftarrow new2(A)$
32. $new2([b|A]) \leftarrow new3(A)$
33. $new2([a|A]) \leftarrow new4(A)$
34. $new3([\,]) \leftarrow$
35. $new4([b|A]) \leftarrow new5(A)$
36. $new4([a|A]) \leftarrow new4(A)$
26. $new5([\,]) \leftarrow$
30. $new5([b|A]) \leftarrow new3(A)$
31. $new5([a|A]) \leftarrow new4(A)$

Level 2. We start off by adding clause 16 to T. Both *Defs* and *InDefs* consist of clause 16 only. Then we execute the body of the WHILE loop.

First Iteration.

UNFOLD. By positive unfolding from clause 16 we derive:

37. $g([\,]) \leftarrow \neg new1([\,])$
38. $g([a|A]) \leftarrow word(A), \neg new1([a|A])$
39. $g([b|A]) \leftarrow word(A), \neg new1([b|A])$

By negative unfolding from clauses 37, 38, and 39 we derive:

40. $g([\,]) \leftarrow$
41. $g([a|A]) \leftarrow word(A), \neg new2(A)$
42. $g([b|A]) \leftarrow word(A)$

DEFINE-FOLD. We introduce the following new definitions:

43. $new6(A) \leftarrow word(A), \neg new2(A)$
44. $new7(A) \leftarrow word(A)$

and by folding clauses 41 and 42 we derive:

45. $g([a|A]) \leftarrow new6(A)$
46. $g([b|A]) \leftarrow new7(A)$

Clauses 43 and 44 are added to *InDefs*. Since *InDefs* is not empty, we continue by a new iteration of the body of the WHILE loop and we stop after the fourth iteration, when *InDefs* becomes empty. We do not show the second, third, and fourth iterations. The final program, whose clauses are listed below, is derived by eliminating all predicate definitions upon which the predicate g does not depend.

40. $g([\,]) \leftarrow$
45. $g([a|A]) \leftarrow new6(A)$
46. $g([b|A]) \leftarrow new7(A)$
47. $new6([\,]) \leftarrow$
48. $new6([a|A]) \leftarrow new8(A)$
49. $new6([b|A]) \leftarrow new9(A)$
50. $new7([\,]) \leftarrow$
51. $new7([a|A]) \leftarrow new7(A)$
52. $new7([b|A]) \leftarrow new7(A)$
53. $new8([\,]) \leftarrow$
54. $new8([a|A]) \leftarrow new8(A)$
55. $new8([b|A]) \leftarrow new10(A)$
56. $new9([a|A]) \leftarrow new7(A)$
57. $new9([b|A]) \leftarrow new7(A)$
58. $new10([a|A]) \leftarrow new8(A)$
59. $new10([b|A]) \leftarrow new9(A)$

This final program corresponds to a deterministic finite automaton in the sense that: (i) each predicate corresponds to a state, (ii) g corresponds to the initial state, (iii) each predicate p which has a unit clause $p([\,]) \leftarrow$, corresponds to a final state, and (iv) each clause of the form $p([s|A]) \leftarrow q(A)$ corresponds to a transition labeled by the symbol s from the state corresponding to p to the state corresponding to q.

The derivation of the final program performed according to our transformation strategy, can be viewed as the derivation of a deterministic finite automaton from a general program for parsing a context free language. Obviously, this derivation has been possible, because the context free grammar encoded by the *production* predicate (see clauses 8–11) generates a regular language.

The final program is much more efficient than the initial program which constructs the complement of a context-free language by performing a nondeterministic search of the productions to apply (see clauses 10 and 11). □

3 Program Synthesis via Transformation Rules and Strategies

In this section we see how one can use for program synthesis the rules and the strategy for program transformation we have presented in Sections 2.2 and 2.3. The program synthesis problem can be defined as follows: Given a *specification* S, that is, a formula written in a specification language, we want to derive, by using some *derivation rules*, a *program* T in a suitable programming language, such that T satisfies S.

There are many synthesis methods described in the literature for deriving programs from specifications and these methods depend on the choice of: (i) the specification language, (ii) the derivation rules, and (iii) the programming language.

It has been recognized since the beginning of its development (see, for instance, [11,32,39]), that logic programming is one of the most effective settings for expressing program synthesis methods, because in logic programming both specifications and programs are formulas of the same language, i.e., the first order predicate calculus, and moreover, the derivation rules for deriving programs from specifications, may be chosen to be the inference rules of the first order predicate calculus itself.

Now we propose a program synthesis method in the case of logic programming. In this case the program synthesis problem can be more specifically defined as indicated in the Introduction. Given a locally stratified program P and a specification of the form: $g(X_1, \ldots, X_n) \leftrightarrow \varphi$, where: (i) g is a new predicate symbol not occurring in $\{P, \varphi\}$, and (ii) φ is a formula of the first order predicate calculus such that $\mathit{freevars}(\varphi) = \{X_1, \ldots, X_n\}$, we want to derive a computationally efficient program T such that, for all ground terms $t_1, \ldots, t_n$,

$$M(P) \models \varphi\{X_1/t_1, \ldots, X_n/t_n\} \quad \text{iff} \quad M(T) \models g(t_1, \ldots, t_n) \qquad (\mathit{Synth})$$

The derivation rules we consider for program synthesis are: (i) a variant of the Lloyd-Topor transformation rules [46], and (ii) the unfold/fold program transformation rules presented in Section 2.2.

Let us begin by presenting the following example of program synthesis. It is our running example for this section and it will be continued in the Examples 4 and 5 below.

Example 3. Specification of List Maximum. Let us consider the following *List-Membership* program:

1. $\mathit{list}([\,]) \leftarrow$
2. $\mathit{list}([A|As]) \leftarrow \mathit{list}(As)$
3. $\mathit{member}(X, [A|As]) \leftarrow X = A$
4. $\mathit{member}(X, [A|As]) \leftarrow \mathit{member}(X, As)$

and $=$ and $\leq$ are basic predicates denoting, respectively, the equality predicate and a given total order predicate over the given domain. For brevity, we do not show the clauses defining these two basic predicates. The maximum M of a list L of items may be specified by the following formula:

$max(L, M) \leftrightarrow (list(L),\ member(M, L),\ \forall X\,(member(X, L) \rightarrow X \leq M))$ (Φ)

By our synthesis method we want to derive an efficient program *Max* which defines the predicate *max* such that:

$M(ListMembership \wedge Max) \models \forall L, M\,(max(L, M) \leftrightarrow \varphi_{max})$

where φ_{max} denotes the right hand side of formula (Φ) above. □

In the rest of this section, we illustrate a synthesis method, called the *unfold/fold synthesis method*, which we now introduce.

The Unfold/Fold Synthesis Method.

Given a locally stratified program P and a specification formula of the form: $g(X_1, \ldots, X_n) \leftrightarrow \varphi$, this method consists of two steps as follows.

Step 1. We apply a variant of the Lloyd-Topor transformation [46], and we derive a conjunction $Cls(g, \varphi)$ of clauses such that $P \wedge Cls(g, \varphi)$ is a locally stratified program and, for all ground terms $t_1, \ldots, t_n$,

(1) $M(P) \models \varphi\{X_1/t_1, \ldots, X_n/t_n\}$ iff $M(P \wedge Cls(g, \varphi)) \models g(t_1, \ldots, t_n)$

Step 2. From the program P, the conjunction $Cls(g, \varphi)$ of clauses, and a set of equivalences to be used for rule R9, by applying the unfold/fold transformation strategy of Section 2.3, we derive a program T such that, for all ground terms $t_1, \ldots, t_n$,

(2) $M(P \wedge Cls(g, \varphi)) \models g(t_1, \ldots, t_n)$ iff $M(T) \models g(t_1, \ldots, t_n)$

and thus, the above relation (*Synth*) holds.

As already mentioned, our unfold/fold synthesis method is a generalization of the two step transformation method presented in the previous Section 2.3, because here we consider a first order formula φ, instead of a goal G. Notice also that, similarly to the transformation method of Section 2.3, the program $P \wedge Cls(g, \varphi)$ itself is a particular program satisfying (2), but usually we have to discard this trivial solution because we look for an efficient program T satisfying (2).

We now illustrate the variant of the method proposed by Lloyd and Topor in [46] which we use for constructing the conjunction of clauses $Cls(g, \varphi)$ starting from the given specification formula $g(X_1, \ldots, X_n) \leftrightarrow \varphi$ according to the requirements indicated in Step 1 above.

We need to consider a class of formulas, called *statements* [46], each of which is of the form: $A \leftarrow \beta$, where A is an atom and β, called the *body* of the statement, is a first order logic formula. We write $C[\gamma]$ to denote a first order formula where the subformula γ occurs as an *outermost conjunct*, 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 $s \geq 0$. We will say that the formula $C[\gamma]$ is transformed into the formula $C[\delta]$ when $C[\delta]$ is obtained from $C[\gamma]$ by replacing the conjunct γ by the new conjunct δ.

The LT transformation.

Given a conjunction of statements, perform the following transformations.
(A) Eliminate from the body of every statement the occurrences of logical constants, connectives, and quantifiers other than *true*, $\neg$, $\wedge$, $\vee$, and $\exists$.

(B) Repeatedly apply the following rules until a conjunction of clauses is generated:

(1) $A \leftarrow C[\neg \mathit{true}]$ is deleted.

(2) $A \leftarrow C[\neg\neg\gamma]$ is transformed into $A \leftarrow C[\gamma]$.

(3) $A \leftarrow C[\neg(\gamma \wedge \delta)]$ is transformed into

$A \leftarrow C[\neg \mathit{newp}(Y_1, \ldots, Y_k)] \ \wedge \ \mathit{newp}(Y_1, \ldots, Y_k) \leftarrow \gamma \wedge \delta$

where *newp* is a new non-basic predicate and $\{Y_1, \ldots, Y_k\} = \mathit{freevars}(\gamma \wedge \delta)$.

(4) $A \leftarrow C[\neg(\gamma \vee \delta)]$ is transformed into $A \leftarrow C[\neg\gamma] \ \wedge \ A \leftarrow C[\neg\delta]$.

(5) $A \leftarrow C[\neg\exists X\, \gamma]$ is transformed into

$A \leftarrow C[\neg \mathit{newp}(Y_1, \ldots, Y_k)] \ \wedge \ \mathit{newp}(Y_1, \ldots, Y_k) \leftarrow \gamma$

where *newp* is a new non-basic predicate and $\{Y_1, \ldots, Y_k\} = \mathit{freevars}(\exists X\, \gamma)$.

(6) $A \leftarrow C[\neg p(t_1, \ldots, t_m)]$ is transformed into

$A \leftarrow C[\neg \mathit{newp}(Y_1, \ldots, Y_k)] \ \wedge \ \mathit{newp}(Y_1, \ldots, Y_k) \leftarrow p(t_1, \ldots, t_m)$

where p is a non-basic predicate which depends on existential variables in P, *newp* is a new non-basic predicate, and $\{Y_1, \ldots, Y_k\} = \mathit{vars}(p(t_1, \ldots, t_m))$.

(7) $A \leftarrow C[\gamma \vee \delta]$ is transformed into $A \leftarrow C[\gamma] \ \wedge \ A \leftarrow C[\delta]$.

(8) $A \leftarrow C[\exists X\, \gamma]$ is transformed into $A \leftarrow C[\gamma\{X/Y\}]$, where Y does not occur in $A \leftarrow C[\exists X\, \gamma]$.

Given a locally stratified program P and a specification $g(X_1, \ldots, X_n) \leftrightarrow \varphi$, we denote by $\mathit{Cls}(g, \varphi)$ the conjunction of the clauses derived by applying the LT transformation to the statement $g(X_1, \ldots, X_n) \leftarrow \varphi$.

Example 4. LT transformation of the List Maximum specification. Let us consider the program *ListMembership* and the specification formula (Φ) of Example 3. By applying the LT transformation to the statement $\mathit{max}(L, M) \leftarrow \mathit{list}(L), \mathit{member}(M, L), \forall X\, (\mathit{member}(X, L) \rightarrow X \leq M)$ we derive the conjunction $\mathit{Cls}(\mathit{max}, \varphi_{\mathit{max}})$ consisting of the following two clauses:

5. $\mathit{max}(L, M) \leftarrow \mathit{list}(L), \mathit{member}(M, L), \neg \mathit{new1}(L, M)$
6. $\mathit{new1}(L, M) \leftarrow \mathit{member}(X, L), \neg X \leq M$

The program *ListMembership* $\wedge$ $\mathit{Cls}(\mathit{max}, \varphi_{\mathit{max}})$ is a very inefficient, generate-and-test program: it works by nondeterministically generating a member M of the list L and then testing whether or not M is the maximum member of L. $\square$

The following result states that the LT transformation is correct w.r.t. the perfect model semantics [46,55].

Theorem 2. [Correctness of LT Transformation w.r.t. Perfect Models] *Let P be a locally stratified program and $g(X_1, \ldots, X_n) \leftrightarrow \varphi$ be a specification. If $Cls(g, \varphi)$ is obtained from $g(X_1, \ldots, X_n) \leftarrow \varphi$ by the LT transformation, then* (i) *$P \wedge Cls(g, \varphi)$ is a locally stratified program and* (ii)*, for all ground terms $t_1, \ldots, t_n$, $M(P) \models \varphi\{X_1/t_1, \ldots, X_n/t_n\}$ iff $M(P \wedge Cls(g, \varphi)) \models g(t_1, \ldots, t_n)$.*

Step 2 of our unfold/fold synthesis method makes use, as already said, of the unfold/fold transformation strategy presented in Section 2.3, starting from program P, the conjunction $Cls(g, \varphi)$ of clauses, instead of $Cls(g, G)$, and a set of equivalences to be used for the application of rule R9.

The partition of $Cls(g, \varphi)$ into levels can be constructed similarly to the partition of $Cls(g, G)$ in Section 2.3. Indeed, by construction, there exists a level mapping of $Cls(g, \varphi)$ such that: (1) $Cls(g, \varphi)$ can be partitioned into K subconjunctions $D^1, \ldots, D^K$, such that $Cls(g, \varphi) = D^1 \wedge \ldots \wedge D^K$, and for $i = 1, \ldots, K$, the subconjunction D^i consists of all clauses in $Cls(g, \varphi)$ whose head predicates are at level i, (2) for $i = 1, \ldots, K$ and for each clause $p(\ldots) \leftarrow B$ in D^i the level of every predicate symbol in the goal B is strictly smaller than the level of p, (3) the predicate g is at the highest level K, and (4) all predicates of $Cls(g, \varphi)$ which occur in P, are at level 0.

The reader may notice that for all $K \geq 0$ there exists a formula ψ and a predicate g such that K is the highest value of the level mapping of $Cls(g, \psi)$.

Example 5. Synthesis of the List Maximum program. Let us consider again the program *ListMembership* and the formula Φ of Example 3. Let us also consider the conjunction $Cls(max, \varphi_{max})$ consisting of clauses 5 and 6 of Example 4 which define the predicates *max* and *new*1. We may choose the level mapping so that the levels of *list*, *member*, $\leq$, $=$ are all 0, the level of *new*1 is 1, and the level of *max* is 2. Thus, the highest level K is 2, $D^1 = \{\text{clause } 6\}$, and $D^2 = \{\text{clause } 5\}$.

We apply our unfold/fold transformation strategy as follows.

Level 1. Initially program T is *ListMembership*. We start off by adding clause 6 to T. Both *Defs* and *InDefs* consist of clause 6 only. Then we execute the body of the WHILE loop as follows.

UNFOLD. We unfold clause 6 w.r.t. $member(X, L)$ and we get:

7. $new1([A|As], M) \leftarrow X = A, \neg X \leq M$
8. $new1([A|As], M) \leftarrow member(X, As), \neg X \leq M$

TAUTOLOGY-REPLACE. From clause 7, by applying the goal replacement rule (using the equivalence $\forall A, M\, (\exists X\, (X = A, \neg X \leq M) \leftrightarrow \neg A \leq M)$) we derive:

9. $new1([A|As], M) \leftarrow \neg A \leq M$

DEFINE-FOLD. By folding clause 8 using clause 6 we derive the clause:

10. $new1([A|As], M) \leftarrow new1(As, M)$

No new definition has been introduced. Thus, *InDefs* is empty and the transformation strategy terminates for level 1. At this point program T is made out of clauses 1, 2, 3, 4, 9, and 10.

Level 2. We start off the transformation strategy for this level, by adding clause 5 to T. Both *Defs* and *InDefs* consist of clause 5 only. Then we iterate twice the body of the WHILE loop as follows.

First Iteration.
UNFOLD. By some unfolding steps, from clause 5 in *InDefs* we derive:

11. $max([A|As], M) \leftarrow list(As),\ M = A,\ A \leq M,\ \neg new1(As, M)$
12. $max([A|As], M) \leftarrow list(As),\ member(M, As),\ A \leq M,\ \neg new1(As, M)$

TAUTOLOGY-REPLACE. By applying the goal replacement rule, from clause 11 we derive:

13. $max([A|As], M) \leftarrow list(As),\ M = A,\ \neg new1(As, M)$

DEFINE-FOLD. The definition of predicate max, consisting of clauses 12 and 13 is nondeterministic, because an atom of the form $max(l, M)$, where l is a ground, nonempty list, is unifiable with the head of both clauses. We may derive a more efficient, deterministic definition for max by introducing the new predicate $new2$ as follows:

14. $new2(A, As, M) \leftarrow list(As),\ M = A,\ \neg new1(As, M)$
15. $new2(A, As, M) \leftarrow list(As),\ member(M, As),\ A \leq M,\ \neg new1(As, M)$

and then folding clauses 12 and 13 using clauses 14 and 15, as follows:

16. $max([A|As], M) \leftarrow new2(A, As, M)$

Now, (i) T consists of clauses 1, 2, 3, 4, 9, 10, and 16, (ii) *Defs* consists of clauses 6, 14, and 15, and (iii) *InDefs* consists of clauses 14 and 15 only.

Second Iteration.
UNFOLD. By positive and negative unfolding, from clauses 14 and 15 in *InDefs* we get:

17. $new2(A, [\,], M) \leftarrow M = A$
18. $new2(A, [B|As], M) \leftarrow list(As),\ M = A,\ B \leq M,\ \neg new1(As, M)$
19. $new2(A, [B|As], M) \leftarrow list(As),\ M = B,\ A \leq M,\ B \leq M,\ \neg new1(As, M)$
20. $new2(A, [B|As], M) \leftarrow list(As),\ member(M, As),\ A \leq M,\ B \leq M,$
$\neg new1(As, M)$

TAUTOLOGY-REPLACE. By applying the basic goal replacement rule to clauses 18, 19, and 20, and in particular, by using the equivalence $M(ListMembership) \models true \leftrightarrow B \leq A \vee A \leq B$ (recall that $\leq$ is a total order), we get:

18.1. $new2(A, [B|As], M) \leftarrow B \leq A,\ list(As),\ M = A,\ \neg new1(As, M)$
19.1. $new2(A, [B|As], M) \leftarrow A \leq B,\ list(As),\ M = B,\ \neg new1(As, M)$
20.1. $new2(A, [B|As], M) \leftarrow B \leq A,\ list(As),\ member(M, As),\ A \leq M,$
$\neg new1(As, M)$
20.2. $new2(A, [B|As], M) \leftarrow A \leq B,\ list(As),\ member(M, As),\ B \leq M,$
$\neg new1(As, M)$

DEFINE-FOLD. Now we fold clauses 18.1 and 20.1 using clauses 14 and 15, and we also fold clauses 19.1 and 20.2 using clauses 14 and 15. We obtain:

21. $new2(A, [B|As], M) \leftarrow B \leq A,\ new2(A, As, M)$
22. $new2(A, [B|As], M) \leftarrow A \leq B,\ new2(B, As, M)$

No new definition has been introduced during the second iteration. Thus, *InDefs* is empty and we terminate our unfold/fold transformation strategy also for the highest level 2. We finally eliminate all predicate definitions on which max does not depend, and we derive our final program:

16. $max([A|As], M) \leftarrow new2(A, As, M)$
17. $new2(A, [\,], M) \leftarrow M = A$
21. $new2(A, [B|As], M) \leftarrow B \leq A,\ new2(A, As, M)$
22. $new2(A, [B|As], M) \leftarrow A \leq B,\ new2(B, As, M)$

This final program deterministically computes the answers to queries of the form: $max(l, M)$ where l is a ground list. Indeed, while traversing the given list l, the first argument of the predicate $new2$ holds the maximal item encountered so far (see clauses 21 and 22) and, at the end of the traversal, the value of this argument is returned as an answer (see clause 17). □

4 Program Verification via Transformation Rules and Strategies

In this section we show that the transformation rules and the strategy we have presented in Sections 2.2 and 2.3, can also be used for program verification. In particular, we can prove a property φ of a given locally stratified logic program P by applying the unfold/fold synthesis method of Section 3. For program verification purposes, instead of starting from a specification formula where free variables may occur, the unfold/fold synthesis method is applied starting from the *closed* specification formula $g \leftrightarrow \varphi$, where $\mathit{freevars}(\varphi) = \emptyset$ and g is a predicate symbol of arity 0.

Our method for verifying whether or not φ holds in the perfect model of the program P is specified as follows.

The Unfold/Fold Verification Method.
Given a locally stratified program P and a closed formula φ, we can check whether or not $M(P) \models \varphi$ holds by performing the following two steps.

Step 1. We introduce a new predicate symbol g of arity 0, not occurring in $\{P, \varphi\}$ and, by using the LT transformation we transform the statement $g \leftarrow \varphi$, into a conjunction $\mathit{Cls}(g, \varphi)$ of clauses, such that $M(P) \models \varphi$ iff $M(P \wedge \mathit{Cls}(g, \varphi)) \models g$.

Step 2. From program P, the conjunction $\mathit{Cls}(g, \varphi)$ of clauses, and a set of equivalences to be used for rule R9, by applying the unfold/fold transformation strategy of Section 2.3, we derive a program T such that

$$M(P \wedge \mathit{Cls}(g, \varphi)) \models g \quad \text{iff} \quad M(T) \models g$$

Thus, if T is the program consisting of the clause $g \leftarrow$ only, then $M(P) \models \varphi$, and if T is the empty program, then $M(P) \not\models \varphi$.

Let us now see an example of program verification.

Example 6. The Yale Shooting Problem. This problem has been often presented in the literature on temporal and nonmonotonic reasoning. It can be formulated as follows. Let us consider a person and a gun and three possible *events*: (e1) a *load* event in which the gun is loaded, (e2) a *shoot* event in which the gun shoots, and (e3) a *wait* event in which nothing happens. These events are represented by clauses 6, 7, and 8 of the program *YSP* below. A *situation* is (the result of) a sequence of events. This sequence is represented as a list which, so to speak, grows to the left as time progresses. In any situation, at least one of the following three facts *holds*: (f1) the person is *alive*, (f2) the person is *dead*, and (f3) the gun is *loaded*. These facts are represented by clauses 9, 10, and 11 below. We have the following statements:

(s1) In the initial situation, represented by the empty list $[\,]$, the person is *alive*.
(s2) After a *load* event the gun is *loaded*.
(s3) If the gun is *loaded*, then after a *shoot* event the person is *dead*.
(s4) If the gun is *loaded*, then it is *abnormal* that after a *shoot* event the person is *alive*.
(s5) If a fact F holds in a situation S and it is not abnormal that F holds after the event E following S, then F holds also after the event E. This statement is often called the *inertia axiom*.

The following locally stratified program, called *YSP*, formalizes the above statements, and in particular, clauses 1–5 correspond to statements (s1)–(s5), respectively. Our *YSP* program is similar to the one of Apt and Bezem [1].

Program *YSP*

1. $holds(alive, [\,]) \leftarrow$
2. $holds(loaded, [load|S]) \leftarrow$
3. $holds(dead, [shoot|S]) \leftarrow holds(loaded, S)$
4. $ab(alive, shoot, S) \leftarrow holds(loaded, S)$
5. $holds(F, [E|S]) \leftarrow fact(F),\ event(E),\ holds(F, S),\ \neg ab(F, E, S)$
6. $event(load) \leftarrow$
7. $event(shoot) \leftarrow$
8. $event(wait) \leftarrow$
9. $fact(alive) \leftarrow$
10. $fact(dead) \leftarrow$
11. $fact(loaded) \leftarrow$
12. $append([\,], Y, Y) \leftarrow$
13. $append([A|X], Y, [A|Z]) \leftarrow append(X, Y, Z)$

Apt and Bezem showed that $M(YSP) \models holds(dead, [shoot, wait, load])$ can be derived in a straightforward way by applying SLDNF-resolution. Let us now consider the following stronger property σ:

$$\forall S\,(holds(dead, S) \rightarrow \exists S1, S2, S3, S4\,(append(S1, [shoot|S2], S4),\ append(S4, [load|S3], S)))$$

meaning that the person may be *dead* in the current situation only if a *load* event occurred in the past and that event was followed, maybe not immediately,

by a *shoot* event. We would like to prove that $M(YSP) \models \sigma$. Our two step verification method works as follows.

Step 1. We apply the LT transformation starting from the statement $g \leftarrow \sigma$ and we derive $Cls(g, \sigma)$ which consists of the following three clauses:

14. $g \leftarrow \neg new1$
15. $new1 \leftarrow holds(dead, S), \neg new2(S)$
16. $new2(S) \leftarrow append(S1, [shoot|S2], S4), append(S4, [load|S3], S)$

The level of $new2$ is 1, the level of $new1$ is 2, and the level of g is 3. The level of all other predicates is 0.

Step 2. We now apply the unfold/fold transformation strategy of Section 2.3, starting from the program *YSP*, the conjunction of clauses $Cls(g, \sigma)$, and an empty set of equivalences (rule R9 will not be applied). We have that $K = 3$, $D^1 = \{\text{clause } 16\}$, $D^2 = \{\text{clause } 15\}$, and $D^3 = \{\text{clause } 14\}$.

Level 1. Initially program T is *YSP*. We start off by applying the definition introduction rule and adding clause 16 to T. Both *Defs* and *InDefs* consist of clause 16 only. Then we iterate the execution of the body of the WHILE loop of the unfold/fold transformation strategy as follows.

First Iteration.
UNFOLD. By unfolding, from clause 16 we derive:

17. $new2([shoot|S]) \leftarrow append(S4, [load|S3], S)$
18. $new2([E|S]) \leftarrow append(S1, [shoot|S2], S4), append(S4, [load|S3], S)$

DEFINE-FOLD. We introduce the following new predicate definition:

19. $new3(A) \leftarrow append(B, [load|C], A)$

and we fold clauses 17 and 18 using clauses 19 and 16, respectively:

20. $new2([shoot|S]) \leftarrow new3(S)$
21. $new2([E|S]) \leftarrow new2(S)$

At this point (i) program T consists of clauses 20 and 21 together with clauses 1–13, (ii) *Defs* consists of clauses 16 and 19, and (iii) *InDefs* consists of clause 19.

Second Iteration.
UNFOLD. By unfolding clause 19 we derive:

22. $new3([load|S]) \leftarrow$
23. $new3([E|S]) \leftarrow append(S4, [load|S3], S)$

DEFINE-FOLD. By folding clause 23 using clause 19 we derive:

22. $new3([load|S]) \leftarrow$
24. $new3([E|S]) \leftarrow new3(S)$

We need not introduce any new clause for folding. Thus, *InDefs* is empty and the WHILE loop terminates for level 1. At this point program T consists of the following clauses:

20. $new2([shoot|S]) \leftarrow new3(S)$

21. $new2([E|S]) \leftarrow new2(S)$
22. $new3([load|S]) \leftarrow$
24. $new3([E|S]) \leftarrow new3(S)$

together with clauses 1–13.

Level 2. We apply the definition introduction rule and we add clause 15 to T. Both *Defs* and *InDefs* consist of clause 15 only. Then we iterate the execution of the body of the WHILE loop as follows.

First Iteration.
UNFOLD. By unfolding, from clause 15 we derive:

25. $new1 \leftarrow holds(loaded, S), \neg new3(S), \neg new2(S)$
26. $new1 \leftarrow holds(dead, S), \neg new2(S)$
27. $new1 \leftarrow holds(dead, S), \neg new3(S), \neg new2(S)$
28. $new1 \leftarrow holds(dead, S), \neg new2(S)$

TAUTOLOGY-REPLACE. Clauses 27 and 28 are subsumed by clause 26 and they can be deleted.

DEFINE-FOLD. We introduce the following new predicate:

29. $new4 \leftarrow holds(loaded, S), \neg new3(S), \neg new2(S)$

and we fold clauses 25 and 28 using clauses 29 and 15, respectively. We get:

30. $new1 \leftarrow new4$
31. $new1 \leftarrow new1$

Now (i) T is made out of clauses 1–13, 20–24, and 30–31, (ii) *Defs* consists of clauses 15 and 29, and (iii) *InDefs* consists of clause 29. Since *InDefs* is not the empty conjunction, we proceed by a second execution of the body of the WHILE loop of the unfold/fold transformation strategy.

Second Iteration.
UNFOLD. By unfolding, from clause 29 we derive:

32. $new4 \leftarrow holds(loaded, S), \neg new3(S), \neg new3(S), \neg new2(S)$
33. $new4 \leftarrow holds(loaded, S), \neg new3(S), \neg new2(S)$

TAUTOLOGY-REPLACE. Clause 32 is deleted because it is subsumed by clause 33.
DEFINE-FOLD. We fold clause 32 using clause 29, and we derive:

34. $new4 \leftarrow new4$

No new clause is added by the definition introduction rule. Thus, *InDefs* is the empty conjunction and the WHILE loop terminates for level 2. Now, predicates $new1$ and $new4$ are useless and their definitions, that is, clauses 30, 31, and 34, are deleted.

Thus, at the end of the transformation strategy for level 2, the derived program T consists of clauses 1–13 and 20–24.

Level 3. We add clause 14 to program T. By unfolding clause 14 we derive:

35. $g \leftarrow$

Our transformation strategy terminates by applying the definition elimination rule and deleting all definitions of predicates upon which g does not depend. Thus our final program consists of clause 35 only, and we have proved that $M(YSP \wedge Cls(g, \sigma)) \models g$ and thus, $M(YSP) \models \sigma$.

The reader may check that g cannot be derived from $YSP \wedge Cls(g, \sigma)$ using SLDNF-resolution, because an SLDNF-refutation of g would require the construction of a finitely failed SLDNF-tree for $new1$ and no such a finite tree exists. Indeed, g may be derived by using SLS-resolution, that is, resolution augmented with the *negation as (finite or infinite) failure* rule. However, the applicability conditions of the negation as infinite failure rule are, in general, not decidable and even not semi-decidable. On the contrary, in our approach we use a set of transformation rules which have decidable applicability conditions, assuming that the equivalence of basic goals is decidable (see the goal replacement rule R9). □

5 Related Work

The idea of program development as a deductive activity in a formal theory has been very fertile in the field of programming methodologies. Early results on this topic are reported, for instance, in [10,11,12,21,32,39,49]. Here we would like to mention some of the contributions to this field, focusing on logic program transformation. In the pioneering work by Hogger [32] program transformation was intended as a particular form of deduction in first order logic. Later, the approach based on the unfold/fold transformations proposed by Burstall and Darlington [10] for functional languages, was adapted to logic languages by Tamaki and Sato [74]. These authors proposed a set of rules for transforming definite logic programs and proved their correctness w.r.t. the least Herbrand model semantics. Since then, several researchers have investigated various aspects of the unfold/fold transformation approach. They also considered its extension to deal with negation [6,29,48,63,70,71], disjunctive programs [30], constraints [4,22], and concurrency [23].

In this chapter we have essentially followed the approach of Tamaki and Sato where the correctness of the transformations is ensured by conditions on the sequence of the transformation rules which are applied during program derivation [74]. The main novelty w.r.t. other papers which follow a similar approach and deal with general logic programs (see, for instance, [63,70,71]) is that our set of rules includes the negative unfolding (R4), the negative folding (R6), and the basic goal replacement rules (R9) which are very useful for the program derivation examples we have presented.

Together with the formalization and the study of the properties of the transformation rules, various strategies for the application of these rules have been considered in the literature. Among others, for case of logic programs we recall: (i) the strategies for deriving *tail recursive* programs [3,17], (ii) the *promotion* strategy for reducing nondeterminism within generate-and-test programs [72],

(iii) the strategy for *eliminating unnecessary variables* and thus, avoiding multiple traversals and intermediate data structures [58], and (iv) the strategy for *reducing nondeterminism* during program specialization [56].

The general unfold/fold transformation strategy we have presented in Section 2.3, extends the above mentioned strategies to the case of programs with locally stratified negation. The interesting fact to notice is that the same general strategy can be refined in different ways so to realize not only program transformation, but also program synthesis and program verification. However, in order to be effective in practice, our general strategy requires some information concerning specific computation domains and classes of programs. For instance, information on the computation domains is needed for the application of the goal replacement rule. The merit of a general purpose transformation strategy rests upon the fact that it provides a uniform guideline for performing program derivation in different computation domains.

The work on unfold/fold program transformation is tightly related to other transformation techniques. In particular, *partial evaluation* (also called *partial deduction*) and other *program specialization* techniques à la Lloyd-Shepherdson [16,27,44,47] can be rephrased in terms of a subset of the unfold/fold rules [56,67]. *Compiling control* [7] is another transformation technique which is related to the rules and strategies approach. Compiling control is based on the idea expressed by Kowalski's motto: *Algorithm = Logic + Control,* and it works as follows. Let us consider a logic program P_1 and let us assume that it is evaluated by using a given control strategy C_1. For instance, C_1 may be the Prolog left-to-right, depth-first control strategy. However, for efficiency reasons we may want to use a different control strategy, say C_2. Compiling control works by deriving from program P_1 a new program P_2 such that P_2 with control strategy C_1 is operationally equivalent to P_1 with control strategy C_2. Although the compiling control technique was not originally presented following the rules and strategies approach, the transformation of program P_1 into program P_2, may often be performed by applying a suitable unfold/fold strategy (see, for instance, [53]).

Moreover, during the last two decades there has been a fruitful interaction between unfold/fold program transformation and program synthesis. To illustrate this point, let us recall here the program synthesis methods based on derivation rules, such as the one proposed by Hogger [32] and, along similar lines, those reported in [34,35,42,68,69] which make use of derivation rules similar to the unfold/fold rules. In this regard, the specific contribution of our chapter consists in providing a method for program synthesis which ensures the correctness w.r.t. the perfect model semantics.

Also related to our rules and strategies approach, is the *proofs-as-programs* approach (see, for instance, [8,25] for its presentation in the case of logic programming) which works by extracting a program from a constructive proof of a specification formula. Thus, in the proofs-as-programs approach, programs synthesis is regarded as a theorem proving activity, whereas by using our unfold/fold method we view theorem proving as a particular case of program synthesis.

Our unfold/fold verification method is related to other methods for verifying program properties. The existence of a relation between program transformation and program verification was pointed out by Burstall and Darlington [10] and then formalized by Kott [36] and Courcelle [14] in the case of applicative program schemata. The essential idea is that, since the transformation rules preserve a given semantics, the transformation of a program P_1 into a program P_2 is also a proof of the equivalence of P_1 and P_2 w.r.t. that semantics. In [54] this idea has also been developed in the case of definite logic programs. The method presented in that paper, called *unfold/fold proof method*, allows us to prove the equivalence of conjunctions of atoms w.r.t. the least Herbrand model of a program. In [64] the unfold/fold proof method has been extended by using a more powerful folding rule and in [65,66] the extended unfold/fold proof method has been applied for the proof of properties of parametrized finite state concurrent systems.

A further extension of the unfold/fold proof method has been presented in [55]. By using the proof method described in [55] one can prove properties of the form $M(P) \models \varphi$ where P is a logic programs with locally stratified negation, $M(P)$ is its perfect model, and φ is any first order formula. In the present chapter we basically followed the presentation of [55].

In recent developments (see, for instance, [24]), it has been shown that the unfold/fold proof method can be used to perform *model checking* [13] of finite or infinite state concurrent systems. To see how this can be done, let us recall that in the model checking approach one formalizes the problem of verifying temporal properties of finite or infinite state systems as the problem of verifying the satisfaction relation $T, s \models_{CTL} F$, where (i) T is a state transition system (regarded as a *Kripke structure*), (ii) s is the initial state of the system, and (iii) F is a formula of the CTL branching time temporal logic. In [24] the problem of verifying $T, s \models_{CTL} F$ is reduced to that of verifying $M(P_T) \models sat(s, F)$, where $M(P_T)$ is the perfect model of a locally stratified program P_T defining a predicate sat which encodes the satisfaction relation $\models_{CTL}$. Thus, the unfold/fold proof method described in Section 4 can be used for performing finite or infinite state model checking starting from the program P_T and the atomic formula $sat(s, F)$. An essential point indicated in [24] is that, in order to deal with infinite sets of states, it is useful to consider logic programs extended with *constraints*.

Finally, we would like to mention that the unfold/fold proof method falls into the wide category of methods that use (constraint) logic programming for software verification. In the specific area of the verification of concurrent systems, we may briefly recall the following ones. (i) The method described in [45] uses partial deduction and *abstract interpretation* [15] of logic programs for verifying safety properties of infinite state systems. (ii) The method presented in [26] uses logic programs with linear arithmetic constraints to encode Petri nets. The least fixpoint of one such program corresponds to the reachability set of the Petri net. This method works by first applying some program transformations (different from the unfold/fold ones) to compute a Presburger formula which is a symbolic representation of the least fixpoint of the program, and then proving that a given safety property holds by proving that it is implied by that Presburger for-

mula. (iii) Similarly to [24,26], also the method presented in [18] uses constraint logic programs to represent infinite state systems. This method can be used to verify CTL properties of these systems by computing approximations of least and greatest fixpoints via abstract interpretation. (iv) The methods in [50] and [61] make use of logic programs (with and without constraints, respectively) to represent finite state systems. These two methods employ *tabulation* techniques [76] to compute fixpoints and they may be used for verifying CTL properties and *modal μ-calculus* [40,57] properties, respectively.

It is difficult to make a precise connection between the unfold/fold proof method and the verification methods listed above, because of the different formalizations and techniques which are used. However, we would like to notice that all verification methods we mentioned above, work by finding, in a more or less explicit way, properties which are *invariants* of the behaviour of a system, and within the unfold/fold proof method, the discovery of invariants is performed by the introduction of suitable predicate definitions which allow folding. This introduction of new definitions is the most creative and least mechanizable step during program transformation.

6 Conclusions

The main objective of this chapter has been to illustrate the power of the rules and strategies approach to the development of programs. This approach is particularly appealing in the case of logic programming and it allows us to separate the correctness requirement from the efficiency requirement during program development. This separation is expressed by our motto: *Program Derivation = Rules + Strategies*. It can be viewed as a variant of Kowalski's motto for program execution: *Algorithm = Logic + Control*.

More specifically, we have considered the unfold/fold transformation rules for locally stratified logic programs and we have outlined a strategy for the application of these transformation rules. As a novel contribution of this chapter we have proposed a general, two step method for performing program transformation, program synthesis, and program verification, and we have presented a powerful unfold/fold transformation strategy which allows one to perform: (1) elimination of multiple visits of data structures, program specialization, and other efficiency improving program transformations, (2) program synthesis from first order specifications, and (3) program verification.

The main advantage of developing several techniques for program derivation in a unified framework, is that we may reuse similar techniques in different contexts. For instance, the program transformation strategy for eliminating unnecessary variables [58] may be reused as a quantifier elimination technique for theorem proving [55]. Moreover, our unified view of program derivation allows us to design a general tool which may be used for machine assisted program transformation, synthesis, and verification.

It should be pointed out that, besides the many appealing features illustrated in this chapter, the transformational approach to program derivation has also

some limitations. Indeed, the problems tackled by program transformation have inherent theoretical limitations due to well-known undecidability results. Thus, in general, program derivation cannot be fully mechanical.

Now we mention some approaches by which we can face this limitation and provide techniques which are effective in practice.

(1) We may design *interactive* program transformation systems, so that many ingenious steps can be performed under the user's guidance, while the most tedious and routine tasks are automatically performed by the system. For instance, KIDS [73] is a successful representative of such interactive systems for program derivation. An important line of further development of interactive transformation systems, is the design of appropriate user interfaces and *programmable* program transformers, which allow the user to interact with the system at a very high level. In particular, in such systems the user should be able to program his own rules and strategies. There are some achievements in this direction in the related fields of term rewriting, program synthesis, and theorem proving. For instance, we recall (i) the ELAN system [5] where the user may specify his own strategy for applying rewriting rules, (ii) the Oyster/Clam system [9] where one can make a *plan* to construct a proof or synthesize a program, and (iii) the *Isabelle* generic theorem prover [51], where it is possible to specify customized deductive systems.

(2) We may consider restricted sets of transformation rules or restricted classes of programs, where certain transformation strategies can be performed in a fully mechanical, algorithmic fashion. For logic programs, a number of algorithmic transformation strategies have been developed, such as the already mentioned techniques for partial deduction, eliminating unnecessary variables, and reducing nondeterminism.

(3) We may enhance the program transformation methodology by using techniques for *global programs analysis*, such as abstract interpretation. This approach may remedy to the fact that the transformation rules are designed to make small, local changes of program code, but for their effective application sometimes we need information on the operational or denotational semantics of the whole program. Various techniques which combine program transformation and abstract interpretation have been developed, especially for the task of program specialization (see, for instance, [28,43,60] in the case of logic programs), but also for the verification of concurrent systems (see [45]). We believe that this line of research is very promising.

Finally, we would like to notice that the program derivation techniques we have described in this chapter are essentially oriented to the development of programs *in-the-small*, that is, within a single software module. We believe that one of the main challenges for logic program development is the extension of these techniques for program transformation, synthesis, and verification, to deal with programs *in-the-large*, that is, with many software modules. Some results in this direction are presented in the chapter by Lau and Ornaghi [41] where software engineering methodologies for developing logic programs in-the-large are proposed.

Acknowledgments

We would like to thank Antonis Kakas and Fariba Sadri for their kind invitation to contribute to this book in honor of Prof. Robert Kowalski. Our derivation examples were worked out by using the MAP transformation system mostly developed by Sophie Renault. We also thank the anonymous referees for their constructive comments.

References

1. K. R. Apt and M. Bezem. Acyclic programs. In D.H.D. Warren and P. Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming, Jerusalem, Israel*, pages 617–633. MIT Press, 1990.
2. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
3. N. Azibi. *TREQUASI: Un système pour la transformation automatique de programmes Prolog récursifs en quasi-itératifs.* PhD thesis, Université de Paris-Sud, Centre d'Orsay, France, 1987.
4. N. Bensaou and I. Guessarian. Transforming constraint logic programs. *Theoretical Computer Science*, 206:81–125, 1998.
5. P. Borovansky, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–95, 2001.
6. A. Bossi, N. Cocco, and S. Etalle. Transforming normal programs by replacement. In A. Pettorossi, editor, *Proceedings 3rd International Workshop on Meta-Programming in Logic, Meta '92, Uppsala, Sweden*, Lecture Notes in Computer Science 649, pages 265–279, Berlin, 1992. Springer-Verlag.
7. M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.
8. A. Bundy, A. Smaill, and G. Wiggins. The synthesis of logic programs from inductive proofs. In J. W. Lloyd, editor, *Computational Logic, Symposium Proceedings, Brussels, November 1990*, pages 135–149, Berlin, 1990. Springer-Verlag.
9. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, Germany*, Lecture Notes in Computer Science, Vol. 449, pages 647–648. Springer, 1990.
10. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
11. K. L. Clark and S. Sickel. Predicate logic: A calculus for deriving programs. In *Proceedings 5th International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, USA*, pages 419–420, 1977.
12. K. L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. In *Proceedings Information Processing '77*, pages 939–944. North-Holland, 1977.
13. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.
14. B. Courcelle. Equivalences and transformations of regular systems – applications to recursive program schemes and grammars. *Theoretical Computer Science*, 42:1–122, 1986.

15. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings 4th ACM-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, 1977.
16. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.
17. S. K. Debray. Optimizing almost-tail-recursive Prolog programs. In *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, Lecture Notes in Computer Science 201, pages 204–219. Springer-Verlag, 1985.
18. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, Lecture Notes in Computer Science 1579, pages 223–239. Springer-Verlag, 1999.
19. Y. Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.
20. Y. Deville and K.-K. Lau. Logic program synthesis. *Journal of Logic Programming*, 19, 20:321–350, 1994.
21. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewod Cliffs, N.J., 1976.
22. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
23. S. Etalle, M. Gabbrielli, and M. C. Meo. Unfold/fold transformations of CCP programs. In D. Sangiorgi and R. de Simone, editors, *Proceedings of the International Conference on Concurrency Theory, Concur98*, Lecture Notes in Computer Science 1466, pages 348–363, 1998.
24. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'01, Florence (Italy)*, Technical Report DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
25. L. Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In D. H. D. Warren and P. Szeredi, editors, *Proceedings Seventh International Conference on Logic Programming, Jerusalem, Israel, June 18-20, 1990*, pages 685–699. The MIT Press, 1990.
26. 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.
27. J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.
28. J. P. Gallagher and J. C. Peralta. Using regular approximations for generalisation during partial evalution. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00), Boston, Massachusetts, USA, January 22-23, 2000.*, pages 44–51. ACM Press, November 1999.
29. P. A. Gardner and J. C. Shepherdson. Unfold/fold transformations of logic programs. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic, Essays in Honor of Alan Robinson*, pages 565–583. MIT, 1991.
30. M. Gergatsoulis. Unfold/fold transformations for disjunctive logic programs. *Information Processing Letters*, 62:23–29, 1997.

31. C.A.R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, 583, October 1969.
32. C. J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, 1981.
33. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
34. T. Kanamori and K. Horiuchi. Construction of logic programs based on generalized unfold/fold rules. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 744–768. The MIT Press, 1987.
35. T. Kawamura. Logic program synthesis from first-order specifications. *Theoretical Computer Science*, 122:69–96, 1994.
36. 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.
37. R. A. Kowalski. Predicate logic as a programming language. In *Proceedings IFIP '74*, pages 569–574. North-Holland, 1974.
38. R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.
39. R. A. Kowalski. *Logic for Problem Solving*. North Holland, 1979.
40. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
41. K.-K. Lau and M. Ornaghi. Logic for component-based software development. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond* (*Essays in honour of Bob Kowalski, Part I*), Lecture Notes in Computer Science 2407, pages 347–373. Springer, 2002.
42. K.-K. Lau and S.D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D.H.D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming (ICLP '90)*, pages 667–684. MIT Press, 1990.
43. M. Leuschel. Program specialisation and abstract interpretation reconciled. In J. Jaffar, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming, Manchester, UK, 15-19 June 1998.*, pages 220–234. The MIT Press, 1998.
44. M. Leuschel, B. Martens, and D. de Schreye. Some achievements and prospects in partial deduction. *ACM Computing Surveys*, 30 (Electronic Section)(3es):4, 1998.
45. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proceedings of LOPSTR '99, Venice, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 1999.
46. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
47. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
48. M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.
49. Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Toplas*, 2:90–121, 1980.
50. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In J. W. Lloyd et al., editor, *First International Conference on Computational Logic, CL 2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 384–398. Springer-Verlag, 2000.

51. L. C. Paulson. The foundation of a generic theorem prover. *J. Automated Reasoning*, 5:363–397, 1989.
52. A. Pettorossi and M. Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, 1996.
53. A. Pettorossi and M. Proietti. Transformation of logic programs. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press, 1998.
54. 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.
55. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, editor, *First International Conference on Computational Logic, CL 2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 613–628. Springer, 2000.
56. A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *Proc. 24-th ACM Symposium on Principles of Programming Languages, Paris, France*, pages 414–427. ACM Press, 1997.
57. V. Pratt. A decidable μ-calculus. In *22nd Symposium on Foundations of Computer Science*, Washington (DC), 1981. IEEE Computer Society Press.
58. 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.
59. T. C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journ. of Automated Reasoning*, 5:167–205, 1989.
60. G. Puebla and M. Hermenegildo. Abstract multiple specialization and its application to program parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.
61. 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.
62. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
63. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. In P. S. Thiagarajan and R. H. C. Yap, editors, *Proceedings of ASIAN'99, 5th Asian Computing Science Conference, Phuket, Thailand, December 10-12*, Lecture Notes in Computer Science 1742, pages 322–333. Springer-Verlag, 1999.
64. 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.
65. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Proceedings of the Sixth International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2000, Berlin, Germany*, Lecture Notes in Computer Science 1785, pages 172–187. Springer, 2000.
66. A. Roychoudhury and I. V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *CAV 2001*, pages 25–37, 2001.

67. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12:7–51, 1993.
68. T. Sato and H. Tamaki. Transformational logic program synthesis. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 195–201. ICOT, 1984.
69. T. Sato and H. Tamaki. First order compiler: A deterministic logic program synthesis algorithm. *Journal of Symbolic Computation*, 8:625–627, 1989.
70. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
71. H. Seki. Unfold/fold transformation of general logic programs for well-founded semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.
72. H. Seki and K. Furukawa. Notes on transformation techniques for generate and test logic programs. In *Proceedings of the International Symposium on Logic Programming, San Francisco, USA*, pages 215–223. IEEE Press, 1987.
73. D. R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering — Special Issue on Formal Methods*, 16(9):1024–1043, September 1990.
74. 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*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.
75. D. H. D. Warren. Implementing Prolog – compiling predicate logic programs. Research Report 39 & 40, Department of Artificial Intelligence, University of Edinburgh, 1977.
76. D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, 1992.