The Transformational Approach to Program Development

Alberto Pettorossi[1], Maurizio Proietti[2], and Valerio Senni[1]

[1] DISP, University of Rome Tor Vergata, Via del Politecnico 1, I-00133 Rome, Italy
{pettorossi,senni}@disp.uniroma2.it
[2] IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy
proietti@iasi.cnr.it

Abstract. We present an overview of the program transformation techniques which have been proposed over the past twenty-five years in the context of logic programming. We consider the approach based on rules and strategies. First, we present the transformation rules and we address the issue of their correctness. Then, we present the transformation strategies and, through some examples, we illustrate their use for improving program efficiency via the elimination of unnecessary variables, the reduction of nondeterminism, and the use of program specialization. We also describe the use of the transformation methodology for the synthesis of logic programs from first-order specifications. Finally, we illustrate some transformational techniques for verifying first-order properties of logic programs and their application to model checking for finite and infinite state concurrent systems.

1 Introduction

When deriving programs from specifications there are, among others, two main objectives to achieve: (i) program correctness, and (ii) program efficiency. Unfortunately, these two objectives are often in contrast with each other. Efficient programs may be rather intricate and their correctness proofs may be quite complex and long.

In order to overcome this difficulty, one can use the so called *program transformation* methodology by which starting from the given formal specifications, one derives efficient programs by applying a sequence of *transformation rules*, each of which preserves correctness. The transformation methodology is particularly appealing when programs are written in a declarative language such as a functional language or a logic language. In those cases, in fact, (i) the formal specifications are formulas which can easily be translated into an initial program which is, thus, correct by construction, and (ii) the transformation rules can be viewed as correctness preserving deduction rules in a suitable logic.

In order to get final programs which are more efficient than the initial ones, we need to apply the transformation rules according to suitable *transformation strategies*. This particular approach to program transformation, called the *rules + strategies* approach, has been first advocated in the seminal paper by Burstall and Darlington [17] in the case of functional programs. Then, as we will indicate at the beginning of the next section, it

has been adapted to logic programs [31,64], constraint logic programs [22,40], and the so-called functional-logic languages [1].

The program transformation methodology can also be used for performing *program synthesis* (see, for instance, [41] and also [5] for a recent survey). In that case the initial program is the declarative specification of a problem and the derived, transformed program is the encoding of an efficient algorithm for solving that problem.

In recent years program transformation has also been used as a technique for *program verification*. It has been shown that via program transformation, one can prove properties of programs [47] and also perform *model checking* for *finite* or *infinite* state systems [25].

In this paper we will focus our attention on the use of the program transformation methodology for the development of logic programs and we will mainly refer to the contributions coming from that area. In Section 2 we will present the most popular transformation rules, such as *unfolding* and *folding*, and we will mention some correctness results for those rules in various logic languages. In Section 3 we will describe some of the strategies that can be used to guide the application of the transformation rules for improving program efficiency. In Sections 4 and 5 we will present some transformational methods for program synthesis and program verification. Finally, in Section 6 we will discuss some future research directions in program transformation.

2 Transformation Rules

Various sets of program transformation rules have been proposed in the literature for several declarative programming languages. In their landmark paper [64] Tamaki and Sato considered definite logic programs and presented a set of transformation rules, including *definition*, *unfolding*, *folding*, *goal replacement*, and *clause deletion*. Under suitable restrictions, these rules are *correct* w.r.t. the least Herbrand model semantics [64]. Indeed, if from program P_0 we derive program P_n by several applications of the transformation rules, then under certain conditions the least Herbrand model is preserved, that is, $M(P_0) = M(P_n)$, where by $M(P)$ we denote the least Herbrand model of the program P. In the subsequent years, Tamaki and Sato's approach has been extended in several directions as we now indicate.

(1) Transformation rules for other logic-based programming languages, besides definite logic programs, have been considered. For instance, various rules have been presented for transforming: (i) general logic programs with *negation* [58], (ii) constraint logic programs [22,26,40], (iii) concurrent constraint logic programs [23,24], (iv) constraint handling rules [62], and functional-logic programs [1].

(2) The correctness of the transformation rules w.r.t. various semantics of logic languages has been proved. In particular, it has been shown that, under suitable conditions, the unfolding and folding transformation rules preserve: (i) the set of answer substitutions computed by SLD-resolution [6], (ii) the sequence of answer substitutions computed according to the Prolog operational semantics [49], (iii) termination properties such as finite failure [58] and left-termination [11], universal termination [7], and acyclicity [12], (iv) various semantics of general logic programs, such as the Clark completion [30], the perfect models of stratified programs [40,58], the stable models [57],

the well-founded models [59], and Kunen's and Fitting's three-valued models [10]. Systematic approaches for proving the correctness of the transformation rules based on the notions of *semantic kernel* and *argumentation semantics*, have been proposed in [4] and [65], respectively.

(3) The set of transformation rules has been extended either by adding extra rules such as *negative unfolding* and *negative folding* [26,60], and *simultaneous replacement* [10], or by relaxing the conditions under which we can apply the usual rules [48,53].

Now we present a set of transformation rules for locally stratified programs [40,45,60]. We will use these rules in the program transformations described in Sections 3, 4, and 5.

Given a locally stratified program P, throughout the paper by $M(P)$ we denote the perfect model of P [2], which is equal to the least Herbrand model in the case of definite logic programs. Given any conjunction C of one or more literals, by $vars(C)$ we denote the set of variables occurring in C. A similar notation will also be used for sets of conjunctions of literals. When applying the transformation rules we will feel free to rewrite clauses by: (i) renaming their variables, and (ii) rearranging the order and removing repeated occurrences of literals occurring in their bodies.

The transformation rules are used to construct a sequence $P_0, \ldots, P_n$ of programs, called a *transformation sequence*. The construction of that sequence is done as follows. Suppose that we have constructed the transformation sequence $P_0, \ldots, P_k$, for $0 \leq k \leq n-1$. Then the next program P_{k+1} in the transformation sequence is derived from program P_k by the application of a transformation rule among the following rules R1–R9.

Rule R1 is the *definition introduction* rule which is applied for introducing a new predicate definition by one or more clauses.

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

$\delta_1 : newp(X_1, \ldots, X_h) \leftarrow B_1, \ \ldots, \ \delta_m : newp(X_1, \ldots, X_h) \leftarrow B_m$

where: (i) $newp$ is a predicate symbol not occurring in $\{P_0, \ldots, P_k\}$, (ii) $X_1, \ldots, X_h$ are distinct variables occurring in $\{B_1, \ldots, B_m\}$, (iii) every predicate symbol occurring in $\{B_1, \ldots, B_m\}$ also occurs in P_0. The set $\{\delta_1, \ldots, \delta_m\}$ of clauses is called the *definition* of $newp$.
By *definition introduction* from program P_k we derive the program $P_{k+1} = P_k \cup \{\delta_1, \ldots, \delta_m\}$. For $k \geq 0$, Defs_k denotes the set of clauses introduced by the definition rule during the transformation sequence $P_0, \ldots, P_k$. In particular, $\mathit{Defs}_0 = \{\}$.

The *unfolding* rule consists in: (i) replacing an atom A occurring in the body of a clause by a suitable instance of the disjunction of the bodies of the clauses whose heads unify with A, and (ii) applying suitable boolean laws for deriving clauses. There are two unfolding rules: (1) the *positive unfolding*, and (2) the *negative unfolding*, corresponding to the case where A occurs positively or negatively, respectively, in the body of the clause to be unfolded.

R2. Positive Unfolding. Let $\gamma : \ H \leftarrow G_L \wedge A \wedge G_R$ be a clause in program P_k and let P'_k be a variant of P_k without variables in common with γ. Let

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

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

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

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

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

be all clauses of program P'_k such that A is unifiable with $K_1, \ldots, K_m$, with most general unifiers $\vartheta_1, \ldots, \vartheta_m$, respectively. Assume that:

1. $A = K_1\vartheta_1 = \cdots = K_m\vartheta_m$, that is, for $i = 1, \ldots, m$, A is an instance of K_i,
2. for $i = 1, \ldots, m$, γ_i has no existential variables, and
3. from $G_L \wedge \neg(B_1\vartheta_1 \vee \ldots \vee B_m\vartheta_m) \wedge G_R$ we get a logically 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* γ *w.r.t.* $\neg A$ we derive the clauses $\eta_1, \ldots, \eta_r$, where for $i = 1, \ldots, r$, η_i is $H \leftarrow Q_i$. From P_k we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_r\}$.

The *folding* rule consists in replacing instances of the bodies of the clauses which are the definition of a predicate by the corresponding head. As for unfolding, we have both the positive folding rule and the negative folding rule, depending on whether folding is applied to positive or negative occurrences of (conjunctions of) literals. Note that by the positive folding rule we may replace $m\,(\geq 1)$ clauses by one clause only.

R4. Positive Folding. Let $\gamma_1, \ldots, \gamma_m$, with $m \geq 1$, be clauses in P_k and let *Defs*$'_k$ be a variant of *Defs*$_k$ without variables in common with $\gamma_1, \ldots, \gamma_m$. Let the definition of a predicate in *Defs*$'_k$ consist of the m clauses

$\delta_1: K \leftarrow B_1, \ \ldots, \ \delta_m: K \leftarrow B_m$

where, for $i = 1, \ldots, m$, B_i is a non-empty conjunction of literals. Suppose that there exists a substitution ϑ such that, for $i = 1, \ldots, m$, clause γ_i is of the form $H \leftarrow G_L \wedge B_i\vartheta \wedge G_R$ and, for every variable $X \in vars(B_i) - vars(K)$, the following conditions hold: (i) $X\vartheta$ is a variable not occurring in $\{H, G_L, G_R\}$, and (ii) $X\vartheta$ does not occur in the term $Y\vartheta$, for any variable Y occurring in B_i and different from X.
By *folding* $\gamma_1, \ldots, \gamma_m$ *using* $\delta_1, \ldots, \delta_m$ we derive the clause η: $H \leftarrow G_L \wedge K\vartheta \wedge G_R$. From P_k we derive the program $P_{k+1} = (P_k - \{\gamma_1, \ldots, \gamma_m\}) \cup \{\eta\}$.

R5. Negative Folding. Let γ be a clause in P_k and let *Defs*$'_k$ be a variant of *Defs*$_k$ without variables in common with γ. Suppose that there exists a predicate in *Defs*$'_k$ whose definition consists of a single clause $\delta: K \leftarrow A$, where A is an atom. Suppose also that there exists a substitution ϑ such that clause γ is of the form: $H \leftarrow G_L \wedge \neg A\vartheta \wedge G_R$ and $vars(K) = vars(A)$.

By *folding* γ *using* δ we derive the clause η: $H \leftarrow G_L \wedge \neg K\vartheta \wedge G_R$. From P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

The following *clause deletion* rule allows us to remove from P_k a redundant clause γ, that is, a clause γ such that $M(P_k) = M(P_k - \{\gamma\})$. Since the problem of testing whether or not $M(P_k) = M(P_k - \{\gamma\})$ is undecidable, we will consider some sufficient conditions based on decidable properties. These sufficient conditions are based on the notions of *subsumed* clause, clause *with false body*, and *useless* clause, which we now define.

A clause γ is *subsumed* by a clause of the form $H \leftarrow G_1$ if γ is of the form $(H \leftarrow G_1 \wedge G_2)\vartheta$ for some substitution ϑ and conjunction of literals G_2. A clause *has a false body* if it is of the form $H \leftarrow G_1 \wedge A \wedge \neg A \wedge G_2$.

The set of *useless predicates* in a program P is the maximal set U of predicates occurring in P such that a predicate p is in U iff every clause γ with head predicated p is of the form $p(\ldots) \leftarrow G_1 \wedge q(\ldots) \wedge G_2$ for some q in U. A clause in a program P is *useless* if the predicate of its head is useless in P. For example, in the following program:

$p(X) \leftarrow q(X) \wedge \neg r(X)$
$q(X) \leftarrow p(X)$
$r(a) \leftarrow$

p and q are useless predicates, while r is not useless.

R6. Clause Deletion. Let γ be a clause in P_k. By *clause deletion* we derive the program $P_{k+1} = P_k - \{\gamma\}$ if one of the following three cases occurs:

R6s. γ is subsumed by a clause in $P_k - \{\gamma\}$;

R6f. γ has a false body;

R6u. γ is useless in P_k.

The following *goal replacement* rule allows us to replace a conjunction of literals occurring in the body of a clause by an equivalent conjunction of literals.

R7. Goal Replacement. Let γ: $H \leftarrow G_1 \wedge Q \wedge G_2$ be a clause in P_k. Suppose that for some conjunction R of literals we have:

$M(P_0) \models \forall X_1 \ldots \forall X_u \, (\exists Y_1 \ldots \exists Y_v \, Q \leftrightarrow \exists Z_1 \ldots \exists Z_w \, R)$

where: (i) $\{X_1, \ldots, X_u\} = vars(\{H, G_1, G_2\})$, (ii) $\{Y_1, \ldots, Y_v\} = vars(Q) - \{X_1, \ldots, X_u\}$, and (iii) $\{Z_1, \ldots, Z_w\} = vars(R) - \{X_1, \ldots, X_u\}$.
Then by *goal replacement* from γ we derive the clause η: $H \leftarrow G_1 \wedge R \wedge G_2$. From P_k we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

The following *equality introduction* rule R8i allows us to substitute a variable for a term occurring in a clause, by adding an equality in the body of the clause. The *equality elimination* rule R8e can be viewed as the inverse of rule R8i.

R8. Equality Introduction and Elimination. Let γ be a clause of the form $(H \leftarrow Body)\{X/t\}$, such that the variable X does not occur in t and let δ be the clause: $H \leftarrow X = t \wedge Body$.
R8i. By *equality introduction* we derive clause δ from clause γ. If γ occurs in P_k then we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\delta\}$.
R8e. By *equality elimination* we derive clause γ from clause δ. If δ occurs in P_k then we derive the new program $P_{k+1} = (P_k - \{\delta\}) \cup \{\gamma\}$.

The *clause splitting* rule allows us to reason by cases according to the truth value of a given atom.

R9. Clause Splitting. Let $\gamma: \ H \leftarrow G$ be a clause in P_k and A be an atom. Then from clause γ we derive the two clauses γ_1: $H \leftarrow A \wedge G$ and γ_2: $H \leftarrow \neg A \wedge G$. From P_k we derive the new program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\gamma_1, \gamma_2\}$.

We say that a transformation sequence $P_0, \ldots, P_n$ is *correct* (w.r.t. the perfect model semantics), if $P_0 \cup \mathit{Defs}_n$ and P_n are locally stratified and $M(P_0 \cup \mathit{Defs}_n) = M(P_n)$. Note that, since we can introduce new predicate symbols by using rule R1, it may be the case that for a correct transformation sequence we have $M(P_0) \neq M(P_n)$.

Transformation sequences constructed by an unrestricted use of the transformation rules may not be correct. Consider, for instance, the program:

P_0: $\quad p \leftarrow q \qquad q \leftarrow$

The perfect model of P_0 is $M(P_0) = \{p, q\}$ and $M(P_0) \models p \leftrightarrow q$. Thus, we may apply the goal replacement rule R7 and replace q by p in $p \leftarrow q$. We derive the new program:

P_1: $\quad p \leftarrow p \qquad q \leftarrow$

The transformation sequence P_0, P_1 is *not* correct, because $M(P_1) = \{q\}$ and, thus, $M(P_0) \neq M(P_1)$. Indeed, P_0 succeeds for the goal p, while P_1 *does not terminate* for the goal p.

One can show that the correctness of a transformation sequence is guaranteed if termination is preserved, that is, if the initial program terminates then also the final program terminates. Now we will state a sufficient condition for the correctness of the transformation rules R1–R9 based on the notion of *left termination* [3]. An *LDNF derivation* is an SLDNF derivation constructed by using the *leftmost selection rule* [3].

Definition 1. *A program P is called* left terminating *if all LDNF derivations of P starting from a ground goal, are finite.*

The following Theorem 1 which follows from results presented in [3,9], states that if we consider a transformation sequence of locally stratified, non-floundering [3,39] programs, then the preservation of left termination guarantees the preservation of the perfect model.

Theorem 1 (Correctness of the Transformation Rules). *Let $P_0, \ldots, P_n$ be a transformation sequence such that, for $k = 0, \ldots, n$, program P_k is locally stratified, non-floundering, and left terminating. Then $M(P_0 \cup \mathit{Defs}_n) = M(P_n)$.*

In Theorem 1 we referred to the notion of left termination. However, weaker notions of termination may be considered and in [36], for instance, there is a correctness result for definite programs based on *existential termination*.

Theorem 1 is theoretically relevant because it relates the correctness of a transformation sequence and the preservation of left termination. However, this result is of limited use in practice for two reasons: (1) left termination is an undecidable property (as well as the properties of being locally stratified and non-floundering), and (2) left termination (or other notions of termination) may be too restrictive, especially in the cases where logic programs are used as specifications.

In Section 5 we will show some examples of transformation of nonterminating programs in the context of program verification and model checking. Correctness results w.r.t. the perfect model semantics which do not make explicit use of termination properties can be found in [26,40,52,58,60]. For lack of space we do not report those results here.

3 Transformation Strategies

In order to construct a transformation sequence $P_0, \ldots, P_n$ such that the final program P_n is more efficient than the initial program P_0, we need to apply suitable procedures, called *transformation strategies*.

In this section we will describe some of the strategies which have been proposed in the literature. In particular, we will present: (i) a strategy for *eliminating unnecessary variables* [50], (ii) a strategy for *reducing nondeterminism* [26], and (iii) a strategy for performing *program specialization* [46].

Several other strategies for transforming logic programs have been proposed. For instance, (i) the strategy for deriving *tail recursive* programs [20], (ii) the strategy for *compiling control* [13], and (iii) the strategy for *changing data representations* and, in particular, for replacing ordinary lists by *difference-lists* [68].

3.1 Eliminating Unnecessary Variables

Logic programs written in a declarative style often make use of *existential variables* (see Section 2) and *multiple variables*, that is, variables with multiple occurrences in the body of a clause. Existential variables and multiple variables are collectively called *unnecessary variables*. In the practice of logic programming, multiple occurrences of existential variables are often used for storing intermediate results, while multiple occurrences of non-existential variables are often used for defining predicates which perform multiple traversals of the input data structure.

The strategy presented in [50] has the objective of eliminating unnecessary variables, thereby avoiding both the construction of intermediate results and the multiple traversal of data structures. This strategy is related to the *deforestation* [67] and the *tupling* [43] strategies, which were introduced for the case of functional programs, and it is also related to *conjunctive partial deduction* [19] which is a technique for eliminating unnecessary variables that follows the *partial deduction* [37] approach, instead of the *rules + strategies* approach.

Now we show an example of application of the strategy for eliminating unnecessary variables.

Example 1 (Two Players Impartial Game). Consider two players sitting at a table. On the table there is a heap of matches. The two players play alternate moves and each move consists in taking away either one (move 1) or two matches (move 2) from the table. A player wins if after the opponent's move, he finds no matches on the table. Let us introduce the predicate $win(N, M)$ which holds iff either $N = 0$ or there are N matches on the table and the player who has to move, wins by making move M.

Given a natural number N, the following program $Game$ computes a move M, if it exists, such that $win(N, M)$ holds.

1. $win(N, M) \leftarrow nat(N) \wedge move(M) \wedge w(N, M)$
2. $w(0, M) \leftarrow$
3. $w(s(N), 1) \leftarrow \neg w(N, 1) \wedge \neg w(N, 2)$
4. $w(s(s(N)), 2) \leftarrow \neg w(N, 1) \wedge \neg w(N, 2)$
5. $nat(0) \leftarrow$
6. $nat(s(N)) \leftarrow nat(N)$
7. $move(1) \leftarrow$
8. $move(2) \leftarrow$

The variable M occurs twice in the body of clause 1. Likewise, the variable N occurs twice in the body of clauses 1, 3, and 4. In particular, the multiple occurrences of N in clauses 3 and 4 leads to a computation with $O(2^n)$ time complexity for any query $win(n, M)$, where n is a natural number and M is a variable. We want to improve the efficiency of the above program *Game* by eliminating the multiple occurrences of variables. The strategy which allows us to do so consists in the iteration of the following two phases (see [50] for details).

Unfold phase: We apply the unfolding rule one or more times starting from clause 1, thereby deriving a set U of clauses;

Define-Fold phase: For each clause γ in U with multiple occurrences of variables in its body, we introduce a suitable new clause δ by rule R1, and we fold γ using δ so that the derived clause η has no multiple occurrences of variables in its body.

For each new clause introduced during the *Define-Fold* phase, we perform one more iteration of the *Unfold* and *Define-Fold* phases. We store in a set, called *Defs*, all clauses introduced during every *Define-Fold* phase and we introduce a new clause δ only if we cannot apply the folding rule by using a clause already belonging to the set *Defs*.

Let us see this strategy for eliminating the multiple occurrences of variables in action in our example.

First Iteration.

Unfold. We apply the positive unfolding rule to clause 1 w.r.t. the leftmost atom in its body and we derive the following two clauses:

9. $win(0, M) \leftarrow move(M) \wedge w(0, M)$
10. $win(s(N), M) \leftarrow nat(N) \wedge move(M) \wedge w(s(N), M)$

By several applications of the positive unfolding rule, from clauses 9 and 10 we derive:

11. $win(0, M) \leftarrow move(M)$
12. $win(s(N), 1) \leftarrow nat(N) \wedge \neg w(N, 1) \wedge \neg w(N, 2)$
13. $win(s(N), 2) \leftarrow nat(N) \wedge w(s(N), 2)$

Define-Fold. We eliminate the multiple occurrences of the variable N from the bodies of clauses 12 and 13 by applying the definition introduction rule R1 and the positive folding rule R4 as follows. By rule R1 we introduce the following two clauses:

14. $new1(N) \leftarrow nat(N) \wedge \neg w(N, 1) \wedge \neg w(N, 2)$
15. $new2(N) \leftarrow nat(N) \wedge w(s(N), 2)$

and by folding clauses 12 and 13 using clauses 14 and 15, respectively, we derive:

16. $win(s(N), 1) \leftarrow new1(N)$
17. $win(s(N), 2) \leftarrow new2(N)$

without multiple occurrences of variables in their bodies. However, in the bodies of clauses 14 and 15 there are multiple occurrences of variables and, in order to eliminate them, we have to perform one more iteration of the *Unfold* and *Define-Fold* phases starting from those two clauses.

Second Iteration.

Unfold. By unfolding clause 14 w.r.t. the leftmost atom in its body, we derive:

18. $new1(0) \leftarrow \neg w(0,1) \wedge \neg w(0,2)$
19. $new1(s(N)) \leftarrow nat(N) \wedge \neg w(s(N),1) \wedge \neg w(s(N),2)$

By negative unfolding, clause 18 is deleted because $w(0,1)$ (and also $w(0,2)$) holds (see clause 2). From clause 19, by negative unfolding w.r.t. $\neg w(s(N),1)$, we derive:

20. $new1(s(N)) \leftarrow nat(N) \wedge w(N,1) \wedge \neg w(s(N),2)$
21. $new1(s(N)) \leftarrow nat(N) \wedge w(N,2) \wedge \neg w(s(N),2)$

Define-Fold. By applying rule R1, we introduce the following two clauses:

22. $new3(N) \leftarrow nat(N) \wedge w(N,1) \wedge \neg w(s(N),2)$
23. $new4(N) \leftarrow nat(N) \wedge w(N,2) \wedge \neg w(s(N),2)$

By folding clauses 20 and 21 using clauses 22 and 23, respectively, we derive:

24. $new1(s(N)) \leftarrow new3(N)$
25. $new1(s(N)) \leftarrow new4(N)$

without multiple occurrences of variables in their bodies. Since in the clauses 22 and 23 introduced by rule R1, there are multiple occurrences of variables, we continue the execution of the strategy starting from these two clauses as we have done above starting from clauses 14 and 15. After some more iterations of the *Unfold* and *Define-Fold* phases we derive the following final program $Game_F$ without multiple occurrences of variables.

11. $win(0,N) \leftarrow move(N)$
16. $win(s(N),1) \leftarrow new1(N)$
17. $win(s(N),2) \leftarrow new2(N)$
24. $new1(s(N)) \leftarrow new3(N)$
25. $new1(s(N)) \leftarrow new4(N)$
26. $new2(s(N)) \leftarrow new1(N)$
27. $new3(0) \leftarrow$
28. $new4(0) \leftarrow$
29. $new4(s(N)) \leftarrow new5(N)$
30. $new5(s(N)) \leftarrow new1(N)$

It can be verified that for the program derivation we have now completed, the local stratification, non-floundering, and left termination conditions of Theorem 1 are all satisfied. In particular, the final program $Game_L$ is a left terminating, *definite* program (and, hence, locally stratified and non-floundering). Thus, $M(Game) = M(Game_L)$.

Program $Game_L$ runs in nondeterministic $O(n)$ time for any query of the form $win(n,M)$. In the next section we will present the transformation from program $Game_L$ into a program running in deterministic $O(n)$ time.

3.2 Reducing Nondeterminism

In this section we will present the *Determinization strategy* [26] which can be applied for improving the efficiency of logic programs by reducing the nondeterminism of their computations. We will see this strategy in action by applying it to the program $Game_L$ we have derived at the end of the previous section.

Example 2 (Two Players Impartial Game, Continued). The program $Game_L$ is nondeterministic because, for any given query $win(n, M)$, where n is a ground term denoting a natural number, SLD-resolution may generate a call which is unifiable with the head of more than one program clause. For instance, if $n > 0$, the initial call $win(n, M)$ unifies with the heads of both clause 16 and clause 17. In other terms, these two clauses are *not mutually exclusive* with respect to calls of the form $win(n, M)$, where n is a ground term.

Non-mutually exclusive clauses can be avoided by transforming program $Game_L$ as follows. By the equality introduction rule R8i, from clauses 16 and 17 we derive:

31. $win(s(N), M) \leftarrow M = 1 \wedge new1(N)$
32. $win(s(N), M) \leftarrow M = 2 \wedge new2(N)$

By applying the definition introduction rule, we introduce the following two clauses:

33. $new6(N, M) \leftarrow M = 1 \wedge new1(N)$
34. $new6(N, M) \leftarrow M = 2 \wedge new2(N)$

By folding clauses 31 and 32 using clauses 33 and 34 we derive:

35. $win(s(N), M) \leftarrow new6(N, M)$

The predicate win is defined by the two clauses 11 and 35 which are mutually exclusive w.r.t. calls of the form $win(n, M)$. Indeed, for any given ground term n, there is at most one clause in $\{11, 35\}$ whose head is unifiable with $win(n, M)$.

Now we are left with the problem of transforming the two clauses 33 and 34 introduced by rule R1, into a set of mutually exclusive clauses (w.r.t. calls of the form $new6(n, M)$, where n is a ground term). The Determinization strategy proceeds similarly to the strategy for eliminating unnecessary variables presented in Section 3.1, by iterating an *Unfold* phase followed by a *Define-Fold* phase. During the *Define-Fold* phase we derive mutually exclusive clauses by introducing new predicates possibly defined by *more than one clause* (while in the strategy for eliminating unnecessary variables each new predicate is defined by precisely one clause).

Let us now see how the Determinization strategy proceeds in action in our example. For lack of space, we present the first iteration only.

First Iteration.

Unfold. By positive unfolding, from clauses 33 and 34 we derive:

36. $new6(s(N), M) \leftarrow M = 1 \wedge new3(N)$
37. $new6(s(N), M) \leftarrow M = 1 \wedge new4(N)$
38. $new6(s(N), M) \leftarrow M = 2 \wedge new1(N)$

Define-Fold. Clauses 36, 37, and 38 are *not mutually exclusive*. By the definition introduction rule we introduce the following three clauses:

39. $new7(N, M) \leftarrow M = 1 \wedge new3(N)$
40. $new7(N, M) \leftarrow M = 1 \wedge new4(N)$
41. $new7(N, M) \leftarrow M = 2 \wedge new1(N)$

By folding clauses 36, 37, and 38 using clauses 39, 40, and 41 we derive:

42. $new6(s(N), M) \leftarrow new7(N, M)$

Clause 42 constitutes a set of mutually exclusive clauses for $new6$ (because it is one clause only). In order to transform the newly introduced clauses 39, 40, and 41 into mutually exclusive clauses, we continue the execution of the Determinization strategy and, after several iterations we derive the following program $Game_D$:

11. $win(0, M) \leftarrow move(M)$
35. $win(s(N), M) \leftarrow new6(N, M)$
42. $new6(s(N), M) \leftarrow new7(N, M)$
43. $new7(0, M) \leftarrow M = 1$
44. $new7(s(N), M) \leftarrow new8(N, M)$
45. $new8(0, M) \leftarrow M = 2$
46. $new8(s(N), M) \leftarrow new9(N, M)$
47. $new9(s(N), M) \leftarrow new7(N, M)$

Program $Game_D$ is left terminating and all conditions of Theorem 1 are satisfied. Thus, $M(Game) = M(Game_D)$. Moreover, program $Game_D$ is a set of mutually exclusive clauses and computes the winning move, for any natural number n, in $O(n)$ deterministic time.

3.3 Program Specialization

Programs are often written in a parametric form so that they can be reused in different contexts, and when a parametric program is reused, one may want to improve its performance by taking advantage of the new context of use. This improvement can often be realized by applying a transformation methodology, called *program specialization* (see [29,32,37] for introductions).

The most used technique for program specialization is *partial evaluation*, also called *partial deduction* in the case of logic programs, where it has been first proposed by [33] (see also [14,15,28,38,55,61,63,66] for early work on this subject). Essentially, partial deduction can be performed by applying the transformation rules R1 (definition introduction), R2 (positive unfolding), R4 (positive folding), and R5 (negative folding) presented in Section 2 with the following restriction: by rule R1 we can introduce a new clause of the form $newp(X_1, \ldots, X_h) \leftarrow A$, where A is an atom and $X_1, \ldots, X_h$ are the variables occurring in A. This restriction limits also folding, as rules R4 and R5 are applied using clauses introduced by rule R1.

Program specialization techniques which make use of more powerful rules, such as unrestricted definition introduction (and, hence, unrestricted folding) and goal replacement have been first proposed in [8]. Here we will present an example of application of the specialization strategy introduced in [46], which extends partial deduction by also eliminating unnecessary variables and reducing nondeterminism. In our example we will derive a specialized pattern matcher for a given pattern, starting from a given parametric pattern matcher. In this example we will use constraint logic programs. As already mentioned, the extension of the transformation rules to the case of constraint logic programs has been studied in [22,26,40].

Example 3 (Constrained Matching). We define a matching relation between two strings of numbers called, respectively, the *pattern* P and the *string* S. We say that the pattern P *matches* the string S, and we write $m(P, S)$, iff $P = [p_1, \ldots, p_n]$ and in S there is a substring $Q = [q_1, \ldots, q_n]$ such that for $i = 1, \ldots, n$, $p_i \leq q_i$. (Much more complex matchers can be considered by allowing a matching relation which can be defined by any constraint logic program.)

The following constraint logic program *Match* can be taken as the specification of our parametric pattern matcher for the pattern P:

1. $m(P, S) \leftarrow app(B, C, S) \wedge app(A, Q, B) \wedge leq(P, Q)$
2. $app([\,], Ys, Ys) \leftarrow$
3. $app([X|Xs], Ys, [X|Zs]) \leftarrow app(Xs, Ys, Zs)$
4. $leq([\,], [\,]) \leftarrow$
5. $leq([X|Xs], [Y|Ys]) \leftarrow X \leq Y \wedge leq(Xs, Ys)$

Suppose that we want to specialize this pattern matcher to the specific pattern $P = [1,0,2]$. The specialization strategy we now apply has the same structure as the strategies presented in Sections 3.1 and 3.2. The improvements gained through the application of the specialization strategy are due to the fact that this strategy: (i) makes some precalculations which depend on the specific pattern $P = [1,0,2]$, (ii) eliminates unnecessary variables, and (iii) reduces nondeterminism. As already mentioned, these improvements are possible because we use more powerful transformation rules with respect to partial deduction (which would only perform the precalculations of Point (i)).

The specialization strategy starts off by introducing the following clause which defines the specialized matching relation m_{sp}:

6. $m_{sp}(S) \leftarrow m([1,0,2], S)$

Now we iterate *Unfold* and *Define-Fold* phases. The main difference with the applications of the strategies presented in Sections 3.1 and 3.2 will be that, in order to get mutually exclusive clauses, before applying the definition introduction rule and the folding rule, we will apply the clause splitting rule R9 whenever needed.

First Iteration

Unfold. We unfold clause 6 w.r.t. the atom $m([1,0,2],S)$. We derive:

7. $m_{sp}(S) \leftarrow app(B, C, S) \wedge app(A, Q, B) \wedge leq([1,0,2], Q)$

Define-Fold. In order to fold clause 7, we introduce the following definition:

8. $new1(S) \leftarrow app(B, C, S) \wedge app(A, Q, B) \wedge leq([1,0,2], Q)$

Then we fold clause 7 and we derive:

9. $m_{sp}(S) \leftarrow new1(S)$

Now the strategy continues by transforming the newly introduced clause 8.

Second Iteration

Unfold. We unfold clause 8 w.r.t. the atoms *app* and *leq* and we get:

10. $new1([X|Xs]) \leftarrow 1 \leq X \wedge app(Q, C, Xs) \wedge leq([0,2], Q)$
11. $new1([X|Xs]) \leftarrow app(B, C, Xs) \wedge app(A, Q, B) \wedge leq([1,0,2], Q)$

Clause Splitting. In order to derive mutually exclusive clauses, thereby reducing nondeterminism, we apply the clause splitting rule to clause 11, by separating the cases when $1 \leq X$ and when $1 > X$ (that is, $\neg(1 \leq X)$). We get:

12. $new1([X|Xs]) \leftarrow 1 \leq X \wedge app(B, C, Xs) \wedge app(A, Q, B) \wedge leq([1,0,2], Q)$
13. $new1([X|Xs]) \leftarrow 1 > X \wedge app(B, C, Xs) \wedge app(A, Q, B) \wedge leq([1,0,2], Q)$

Define-Fold. In order to fold clauses 10 and 12 we introduce the following two clauses defining the predicate $new2$:

14. $new2(Xs) \leftarrow app(Q, C, Xs) \wedge leq([0, 2], Q)$
15. $new2(Xs) \leftarrow app(B,C,Xs) \wedge app(A,Q,B) \wedge leq([1,0,2],Q)$

Then we fold clauses 10 and 12 by using the two clauses 14 and 15 and we also fold clause 13 by using clause 8. We derive the following clauses:

16. $new1([X|Xs]) \leftarrow 1 \leq X \wedge new2(Xs)$
17. $new1([X|Xs]) \leftarrow 1 > X \wedge new1(Xs)$

Note that these two clauses: (i) are specialized w.r.t. the information that the first element of the pattern is 1, (ii) have no unnecessary variables, and (iii) are mutually exclusive because of the constraints $1 \leq X$ and $1 > X$.

Now the program transformation strategy continues by transforming clauses 14 and 15, which define predicate $new2$. After a few more iterations of the *Unfold*, *Clause Splitting*, and *Define-Fold* phases, we derive the following specialized program $Match_{sp}$:

9. $m_{sp}(S) \leftarrow new1(S)$
16. $new1([X|Xs]) \leftarrow 1 \leq X \wedge new2(Xs)$
17. $new1([X|Xs]) \leftarrow 1 > X \wedge new1(Xs)$
18. $new2([X|Xs]) \leftarrow 1 \leq X \wedge new3(Xs)$
19. $new2([X|Xs]) \leftarrow 0 \leq X \wedge 1 > X \wedge new4(Xs)$
20. $new2([X|Xs]) \leftarrow 0 > X \wedge new1(Xs)$
21. $new3([X|Xs]) \leftarrow 2 \leq X \wedge new5(Xs)$
22. $new3([X|Xs]) \leftarrow 1 \leq X \wedge 2 > X \wedge new3(Xs)$
23. $new3([X|Xs]) \leftarrow 0 \leq X \wedge 1 > X \wedge new4(Xs)$
24. $new3([X|Xs]) \leftarrow 0 > X \wedge new1(Xs)$
25. $new4([X|Xs]) \leftarrow 2 \leq X \wedge new6(Xs)$
26. $new4([X|Xs]) \leftarrow 1 \leq X \wedge 2 > X \wedge new2(Xs)$
27. $new4([X|Xs]) \leftarrow 1 > X \wedge new1(Xs)$
28. $new5([X|Xs]) \leftarrow$
29. $new6([X|Xs]) \leftarrow$

This final program $Match_{sp}$ has no occurrences of unnecessary variables and is deterministic in the sense that at most one clause can be applied during the evaluation of any ground goal. The efficiency of $Match_{sp}$ is very high because it behaves like a deterministic finite automaton (see Figure 1) as the Knuth-Morris-Pratt matcher.

4 Program Synthesis

Program synthesis is a technique for the automatic derivation of programs from their formal specifications (see, for instance, [41] for the derivation of functional programs and [16,27,31] for the derivation of logic programs from first-order logic specifications).

In this section we present a transformational approach to program synthesis [26,56]. By following this approach, the synthesis of an *efficient* logic program from a first order logic specification can be performed in two steps: first (1) we translate the specification into a possibly inefficient logic program by applying the *Lloyd-Topor transformation* [39], and then (2) we derive an efficient program by applying the transformation rules and strategies described in Sections 2 and 3.

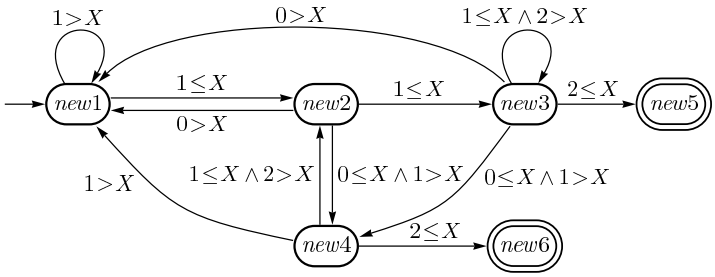

Fig. 1. The finite automaton corresponding to the program $Match_{sp}$ made out of clauses 9 and 16–29. The initial state is $new1$ and the final states are $new5$ and $new6$.

The transformational program synthesis approach will be presented through the N-queens example. This example also illustrates that powerful programming techniques such as recursion and backtracking, which are often presented in the literature for solving the N-queens problem, can indeed be automatically derived by transformation.

Example 4 (N-queens). We are required to place $N\,(\geq 0)$ queens on an $N \times N$ chess board, so that no two queens attack each other, that is, they do not lie on the same row, column, or diagonal. By using the fact that no two queens should lie on the same column, the positions of the N queens on the chess board can be denoted by the list $L = [i_1, \ldots, i_N]$ such that, for $1 \leq k \leq N$, i_k is the row where the queen on column k is placed.

A specification of the solution L for the N-queens problem is given by the following first-order formula:

$$\begin{aligned} \mathit{board}(N, L) =_{\mathit{def}}\ & \mathit{nat}(N) \wedge \mathit{nat\_list}(L) \wedge \mathit{length}(L, N) \wedge \\ & \forall X\,(\mathit{member}(X, L) \rightarrow \mathit{in}(X, 1, N)) \wedge \\ & \forall A\,\forall B\,\forall K\,\forall M \\ & \qquad ((1 \leq K \wedge K < M \wedge \mathit{occurs}(A, K, L) \wedge \mathit{occurs}(B, M, L)) \\ & \qquad\qquad \rightarrow (A \neq B \wedge A - B \neq M - K \wedge B - A \neq M - K)) \end{aligned}$$

where the various predicates that occur in $\mathit{board}(N, L)$, are defined by the following constraint logic program P:

$$\begin{aligned} & \mathit{nat}(0) \leftarrow \\ & \mathit{nat}(N) \leftarrow N = M + 1 \wedge M \geq 0 \wedge \mathit{nat}(M) \\ & \mathit{nat\_list}([\,]) \leftarrow \\ & \mathit{nat\_list}([H|T]) \leftarrow \mathit{nat}(H) \wedge \mathit{nat\_list}(T) \\ & \mathit{length}([\,], 0) \leftarrow \\ & \mathit{length}([H|T], N) \leftarrow N = M + 1 \wedge M \geq 0 \wedge \mathit{length}(T, M) \\ & \mathit{member}(X, [H|T]) \leftarrow X = H \\ & \mathit{member}(X, [H|T]) \leftarrow \mathit{member}(X, T) \\ & \mathit{in}(X, M, N) \leftarrow X = N \wedge M \leq N \\ & \mathit{in}(X, M, N) \leftarrow N = K + 1 \wedge M \leq K \wedge \mathit{in}(X, M, K) \\ & \mathit{occurs}(X, I, [H|T]) \leftarrow I = 1 \wedge X = H \\ & \mathit{occurs}(X, J, [H|T]) \leftarrow J = I + 1 \wedge I \geq 1 \wedge \mathit{occurs}(X, I, T) \end{aligned}$$

In this program P we have that: (i) $in(X, M, N)$ iff $M \leq X \leq N$, and (ii) $occurs(X, I, [a_1, \ldots, a_n])$ iff $X = a_i$ and $I = i$. Now, we would like to synthesize a constraint logic program R which computes a predicate $queens(N, L)$ such that, for every N and L, the following property holds:

$$M(R) \models queens(N, L) \text{ iff } M(P) \models board(N, L) \qquad (\alpha)$$

where by $M(R)$ and $M(P)$ we denote the perfect model of the programs R and P, respectively. By applying the technique presented in [26], we start off from the formula $queens(N, L) \leftarrow board(N, L)$ (where $board(N, L)$ is the first order formula defined above) and, by applying a variant of the Lloyd-Topor transformation, we derive the following stratified program F:

$queens(N, L) \leftarrow nat(N) \wedge nat\_list(L) \wedge length(L, N) \wedge \neg aux1(L, N) \wedge \neg aux2(L)$
$aux1(L, N) \leftarrow member(X, L) \wedge \neg in(X, 1, N)$
$aux2(L) \leftarrow 1 \leq K \wedge K < M \wedge \neg(A \neq B \wedge A - B \neq M - K \wedge B - A \neq M - K) \wedge$
$\quad occurs(A, K, L) \wedge occurs(B, M, L)$

It can be shown that this variant of the Lloyd-Topor transformation preserves the perfect model semantics and, thus, we have that, for every N and L:

$$M(P \cup F) \models queens(N, L) \text{ iff } M(P) \models board(N, L).$$

The derived program $P \cup F$ is not satisfactory from a computational point of view, when using LDNF resolution. Indeed, for a query of the form $queens(n, L)$, where n is a nonnegative integer and L is a variable, program $P \cup F$ works by first generating a value l for the list L and then testing whether or not $length(l, n) \wedge \neg aux1(l, n) \wedge \neg aux2(l)$ holds. This generate-and-test behavior is very inefficient and it may also lead to nontermination. Thus, the process of program synthesis proceeds by applying the definition, unfolding, folding, and goal replacement transformation rules, according to a strategy similar to the ones we have described in Section 3, with the objective of deriving a more efficient program. We derive the following definite program R:

$queens(N, L) \leftarrow new2(N, L, 0)$
$new2(N, [\,], K) \leftarrow N = K$
$new2(N, [H|T], K) \leftarrow N \geq K + 1 \wedge new2(N, T, K + 1) \wedge new3(H, T, N, 0)$
$new3(A, [\,], N, M) \leftarrow in(A, 1, N) \wedge nat(A)$
$new3(A, [B|T], N, M) \leftarrow A \neq B \wedge A - B \neq M + 1 \wedge B - A \neq M + 1 \wedge nat(B) \wedge$
$\quad new3(A, T, N, M + 1)$

together with the clauses listed above which define the predicates in and nat.

Since the transformation rules preserve the perfect model semantics, for every N and L, we have that, $M(R) \models queens(N, L)$ iff $M(P \cup F) \models queens(N, L)$ and, thus, Property (α) holds. It can be shown that program R terminates for all queries of the form $queens(n, L)$. Program R computes a solution for the N-queens problem in a clever way: each time a new queen is placed on the board, program R tests whether or not that queen attacks any other queen already placed on the board.

5 Program Verification

Proofs of program properties are often needed during program development for checking the correctness of software components with respect to their specifications. It has

been shown that the transformation rules introduced in [17,64] can be used for proving several kinds of program properties, such as equivalences of functions defined by recursive equation programs [34], equivalences of predicates defined by logic programs [44], first-order properties of predicates defined by constraint logic programs [47], and temporal properties of concurrent systems [25,54].

In this section we see the use of program transformation for proving program properties specified either by first-order logic formulas or by temporal logic formulas.

5.1 The Unfold/Fold Proof Method

Through a simple example taken from [47], now we illustrate a method, called *unfold/fold proof method*, which uses the program transformation methodology for proving first-order properties of constraint logic programs. Consider the following constraint logic program *Member* which defines the membership relation between an element and a list of elements:

$member(X, [Y|L]) \leftarrow X = Y$ $\qquad$ $list([\,]) \leftarrow$
$member(X, [Y|L]) \leftarrow member(X, L)$ $\qquad$ $list([H|T]) \leftarrow list(T)$

Suppose we want to show that every finite list of numbers has an upper bound, that is, we want to prove the following formula:

$$\forall L\,(list(L) \rightarrow \exists U\,\forall X\,(member(X, L) \rightarrow X \leq U)) \qquad (\beta)$$

The unfold/fold proof method works in two steps, which are similar to the two steps of the transformational synthesis approach presented in Section 4. In the first step, the formula β is transformed into a set of clauses by applying a variant of the Lloyd-Topor transformation, thereby deriving the following program:

$P1$: $prop \leftarrow \neg p$
$p \leftarrow list(L) \wedge \neg q(L)$
$q(L) \leftarrow list(L) \wedge \neg r(L, U)$
$r(L, U) \leftarrow X > U \wedge list(L) \wedge member(X, L)$

The predicate $prop$ is equivalent to β in the sense that $M(Member) \models \beta$ iff $M(Member \cup P1) \models prop$. The correctness of this transformation can be checked by realizing that $M(Member) \models \beta \leftrightarrow \neg\exists L(list(L) \wedge \neg(\exists U(list(L) \wedge \neg(\exists X\,(X > U \wedge list(L) \wedge member(X, L))))))$.

In the second step, we eliminate the *existential variables* occurring in $P1$ (see Section 2 for a definition) by applying the transformation strategy for eliminating unnecessary variables presented in Section 3.1. We derive the following program $P2$ which defines the predicate *prop*:

$P2$: $prop \leftarrow \neg p$ $\qquad$ $p \leftarrow p_1$ $\qquad$ $p_1 \leftarrow p_1$

Now, $P2$ is a propositional program and has a *finite* perfect model, which is $\{prop\}$. Since it can be shown that all transformations we have performed preserve the perfect model, we have that $M(Member) \models \beta$ iff $M(P2) \models prop$ and, therefore, we have completed the proof of β because $prop$ belongs to $M(P2)$.

The expert reader will note that the unfold/fold proof method we have now illustrated, can be viewed as an extension to constraint logic programs of the *quantifier elimination* method, which has well-known applications in the field of automated theorem proving (see [51] for a brief survey).

5.2 Infinite-State Model Checking

As indicated in [18], the behavior of a concurrent system that evolves over time according to a given protocol can be modeled as a *state transition system*, that is, (i) a set S of *states*, (ii) an *initial state* $s_0 \in S$, and (iii) a *transition relation* $t \subseteq S \times S$. We assume that the transition relation t is *total*, that is, for every state $s \in S$ there exists at least one state $s' \in S$, called a *successor state* of s, such that $t(s, s')$ holds. A *computation path* starting from a state s_1 (not necessarily, the initial state) is an *infinite* sequence of states $s_1\, s_2 \ldots$ such that, for every $i \geq 1$, there is a transition from s_i to s_{i+1}, that is, $t(s_i, s_{i+1})$ holds.

The properties of the evolution over time, that is, the computation paths, of a concurrent system can be specified by using a formula of a temporal logic called *Computation Tree Logic* (or CTL, for short [18]). The formulas of CTL are built from a given set of *elementary properties*, each of which may or may not hold in a particular state, by using: (i) the connectives: *not* and *and*, (ii) the quantifiers along a computation path: g ('for all states on the path' or 'globally'), f ('there exists a state on the path' or 'in the future'), x ('next time'), and u ('until'), and (iii) the quantifiers over computation paths: a ('for all paths') and e ('there exists a path'). Quantified formulas are written in a compact form and, for instance, we will write $ef(F)$ and $ag(F)$, instead of $e(f(F))$ and $a(g(F))$, respectively.

Very efficient algorithms and tools exist for verifying temporal properties of *finite state transition systems*, that is, systems where the set S of states is finite [18]. However, many concurrent systems cannot be modeled by finite state transition systems. The problem of verifying CTL properties of *infinite* state transition systems is, unfortunately, undecidable and, thus, it cannot be tackled by traditional model checking techniques. For this reason various methods based on automated theorem proving have been proposed for extending model checking so to deal with infinite state systems (see [21] for a method based on constraint logic programming). Due to the above mentioned undecidability limitation, all these methods are necessarily incomplete.

Now we present a method for verifying temporal properties of (finite or infinite) state transition systems which is based on transformation techniques for constraint logic programs [25]. As an example we consider the *Bakery* protocol [35] and we verify that it satisfies the *mutual exclusion* and *starvation freedom* properties.

Let us consider two agents A and B which want to access a shared resource in a mutually exclusive way by using the Bakery protocol. The state of the agent A is represented by a pair $\langle A1, A2\rangle$, where $A1$, called the *control state*, is an element of the set $\{t, w, u\}$ (where t, w, and u stand for *think*, *wait*, and *use*, respectively) and $A2$, called the *counter*, is a natural number. Analogously, the state of agent B is represented by a pair $\langle B1, B2\rangle$. The *state* of the system consisting of the two agents A and B, whose states are $\langle A1, A2\rangle$ and $\langle B1, B2\rangle$, respectively, is represented by the 4-tuple $\langle A1, A2, B1, B2\rangle$. The transition relation t of the two agent system from an old state $OldS$ to a new state $NewS$, is defined as follows:

$t(OldS,\ NewS) \leftarrow t_A(OldS,\ NewS)$
$t(OldS,\ NewS) \leftarrow t_B(OldS,\ NewS)$

where the transition relation t_A for the agent A is given by the following clauses whose bodies are conjunctions of constraints (see also Figure 2):

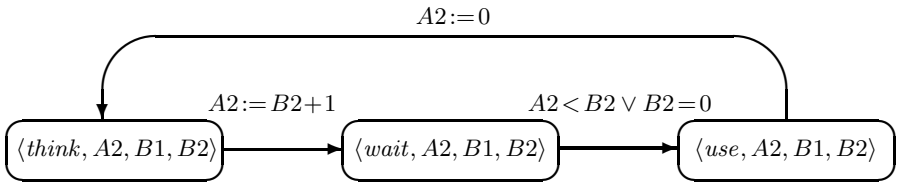

Fig. 2. The Bakery protocol: a graphical representation of the transition relation t_A for the agent A. The assignment $X := e$ on the arc from a state s_1 to a state s_2 tells us that the value of the variable X in s_2 is the value of the expression e in s_1. The boolean expression b on the arc from a state s_1 to a state s_2 tells us that the transition from s_1 to s_2 takes place iff b holds.

$t_A(\langle t, A2, B1, B2\rangle, \langle w, A21, B1, B2\rangle) \leftarrow A21 = B2+1$
$t_A(\langle w, A2, B1, B2\rangle, \langle u, A2, B1, B2\rangle) \leftarrow A2 < B2$
$t_A(\langle w, A2, B1, B2\rangle, \langle u, A2, B1, B2\rangle) \leftarrow B2 = 0$
$t_A(\langle u, A2, B1, B2\rangle, \langle t, A21, B1, B2\rangle) \leftarrow A21 = 0$

The following similar clauses define the transition relation t_B for the agent B:

$t_B(\langle A1, A2, t, B2\rangle, \langle A1, A2, w, B21\rangle) \leftarrow B21 = A2+1$
$t_B(\langle A1, A2, w, B2\rangle, \langle A1, A2, u, B2\rangle) \leftarrow B2 < A2$
$t_B(\langle A1, A2, w, B2\rangle, \langle A1, A2, u, B2\rangle) \leftarrow A2 = 0$
$t_B(\langle A1, A2, u, B2\rangle, \langle A1, A2, t, B21\rangle) \leftarrow B21 = 0$

Note that the system has an infinite number of states, because counters may increase in an unbounded way.

The temporal properties of a transition system are specified by defining a predicate $sat(S, P)$ which holds if and only if the temporal formula P is true at the state S. For instance, the following clauses define the predicate $sat(S, P)$ for the cases where P is: (i) an elementary formula F, (ii) a formula of the form $not(F)$, (iii) a formula of the form $and(F_1, F_2)$, and (iv) a formula of the form $ef(F)$:

$sat(S, F) \leftarrow elem(S, F)$
$sat(S, not(F)) \leftarrow \neg sat(S, F)$
$sat(X, and(F_1, F_2)) \leftarrow sat(X, F_1) \wedge sat(X, F_2)$
$sat(S, ef(F)) \leftarrow sat(S, F)$
$sat(S, ef(F)) \leftarrow t(S, T) \wedge sat(T, ef(F))$

where $elem(S, F)$ holds iff F is an elementary property which is true at state S. In particular, for the Bakery protocol we have the following clause:

$elem(\langle u, A2, u, B2\rangle, unsafe) \leftarrow$

that is, *unsafe* holds at a state where both agents A and B are in the control state u, that is, both agents use the shared resource at the same time. We have that $sat(S, ef(F))$ holds iff there exists a computation path π starting from state S and there exists a state S' on π such that F is true at S'.

The mutual exclusion property holds for the Bakery protocol if there is no computation path starting from the initial state such that at a state on this path the *unsafe* property holds. Thus, the mutual exclusion property holds if $sat(\langle t, 0, t, 0\rangle, not(ef(unsafe)))$ belongs to the perfect model $M(P_{mex})$, where: (i) $\langle t, 0, t, 0\rangle$ is the initial state of the system and (ii) P_{mex} is the program consisting of the clauses for the predicates t, t_A, t_B, *sat*, and *elem* defined above.

In order to show that $sat(\langle t, 0, t, 0\rangle, not(ef(unsafe))) \in M(P_{mex})$, we introduce a new predicate mex defined by the following clause:

$$mex \leftarrow sat(\langle t, 0, t, 0\rangle, not(ef(unsafe))) \qquad (\mu)$$

and we transform the program $P_{mex} \cup \{\mu\}$ into a new program Q which contains a clause of the form $mex \leftarrow$ (see [25] for details). This transformation is performed by applying the definition, unfolding, and folding rules according to a strategy similar to the specialization strategy presented in Section 3.3, that is, a strategy that derives specialized clauses for the evaluation of the predicate mex. From the correctness of the transformation rules we have that $mex \in M(Q)$ iff $mex \in M(P_{mex} \cup \{\mu\})$ and, hence, $sat(\langle t, 0, t, 0\rangle, not(ef(unsafe))) \in M(P_{mex})$, that is, the mutual exclusion property holds.

By applying the same methodology we can also prove the *starvation freedom* property for the Bakery protocol. This property ensures that an agent, say A, which requests the shared resource, will eventually get it. This property is expressed by the CTL formula: $ag(w_A \rightarrow af(u_A))$, which is equivalent to: $not(ef(and(w_A, not(af(u_A)))))$. The clauses defining the elementary properties w_A and u_A are:

$elem(\langle w, A2, B1, B2\rangle, w_A) \leftarrow$
$elem(\langle u, A2, B1, B2\rangle, u_A) \leftarrow$

The clauses defining the predicate $sat(S, P)$ for the case where P is a CTL formula of the form $af(F)$ are:

$sat(X, af(F)) \leftarrow sat(X, F)$
$sat(X, af(F)) \leftarrow ts(X, Ys) \wedge sat\_all(Ys, af(F))$
$sat\_all([\,], F) \leftarrow$
$sat\_all([X|Xs], F) \leftarrow sat(X, F) \wedge sat\_all(Xs, F)$

where $ts(X, Ys)$ holds iff Ys is a list of all the successor states of the state X. For instance, one of the clauses defining predicate ts in our Bakery example is:

$$ts(\langle t, A2, t, B2\rangle, [\langle w, A21, t, B2\rangle, \langle t, A2, w, B21\rangle]) \leftarrow A21 = B2+1 \wedge B21 = A2+1$$

which says that the state $\langle t, A2, t, B2\rangle$ has two successor states: $\langle w, A21, t, B2\rangle$, with $A21 = B2+1$, and $\langle t, A2, w, B21\rangle$, with $B21 = A2+1$.

Let P_{sf} denote the program obtained by adding to P_{mex} the clauses defining: (i) the elementary properties w_A and u_A, (ii) the predicate ts, (iii) the atom $sat(X, af(F))$, and (iv) the predicate sat_all. In order to verify the starvation freedom property we introduce the clause:

$$sf \leftarrow sat(\langle t, 0, t, 0\rangle, not(ef(and(w_A, not(af(u_A)))))) \qquad (\sigma)$$

and, by applying the definition, unfolding, and folding rules according to the specialization strategy, we transform the program $P_{sf} \cup \{\sigma\}$ into a new program R which contains a clause of the form $sf \leftarrow$.

Note that the derivations needed for verifying the mutual exclusion and the starvation freedom properties can be done in a fully automatic way by using the experimental constraint logic program transformation system MAP [42].

6 Conclusions and Future Directions

We have presented the program transformation methodology and we have demonstrated that it is very effective for: (i) the derivation of correct software modules from their formal specifications, and (ii) the proof of properties of programs. Since program transformation preserves correctness and improves efficiency, it is very useful for constructing software products which are provably correct and whose time and space performance is very high.

During the past twenty-five years the research community in Italy has given a very relevant contribution to the program transformation field and, more in general, to the field of logic-based program development. The extent of this contribution is witnessed by the numerous scientific papers, a small fraction of which have been mentioned in this brief survey.

The contribution of the Italian research community has also been carried out through the participation in several national and international research projects which included as an important topic the transformation methodology of logic programs. In particular, we would like to mention the following projects: (i) ESPRIT Alpes (1984–89), (ii) Compulog I and Compulog II (1989–95), (iii) the INTAS Project 'Efficient Symbolic Computing' (1994-98), (iv) the Network of Excellence on Computational Logic, (v) the Humal Capital and Mobility Project 'Logic Program Synthesis and Transformation' (1993–96), (vi) the Italian 'Progetto Finalizzato Informatica II' (1989–93), (vii) the ANATRA Project 'Strumenti per l'analisi e la trasformazione dei programmi' (1994–95), (viii) 'Programmazione Logica: Strumenti per analisi e trasformazione di programmi, Tecniche di ingegneria del software, Estensioni con vincoli, concorrenza ed oggetti' (1995–96), (ix) Progetto Speciale 'Verifica, analisi e trasformazione di programmi logici' (1998–99), and (x) 'Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software' (1998–2000). These projects were supported by the European Union, the Italian Ministry of Education, University, and Research (MIUR), and the Italian National Research Council (CNR).

All these projects gave to the research community in Italy invaluable opportunities to cooperate with other scientific groups in Europe, to strengthen their theoretical background on logic programming and to produce powerful systems and tools for logic program development, logic program analysis, knowledge representation and manipulation using logic. Research teams in Bologna, Padua, Pisa, Rome, and Venice, among others, grew considerably strong through those projects and their expertise and competence spread all over the international community and since then, their high reputation has been widely recognized.

Finally, the Italian research community has also given a very relevant contribution to the organization and the scientific success of the various meetings dedicated to the dissemination of research in logic program transformation, such as the series of Workshops and Symposia on Logic-Based Program Synthesis and Transformation (LOPSTR), held annually since 1991, and on Partial Evaluation and Semantics-Based Program Manipulation (PEPM).

Now, looking at the directions for future research, we would like to point out that, in order to make program transformation even more effective, we need to increase the

level of automation of the transformation strategies for program improvement, program synthesis, and program verification. Furthermore, these strategies should be incorporated into powerful tools for program development.

Another important direction for future research is the exploration of new areas of application of the transformation methodology. In this paper we have described the use of program transformation for verifying temporal properties of infinite state concurrent systems. Similar techniques could also be devised for verifying other kinds of properties and other classes of systems, such as security properties of distributed systems, safety properties of hybrid systems, and protocol conformance of multiagent systems. A more challenging issue is the fully automatic synthesis of software systems which are guaranteed to satisfy some given properties specified by the designer.

7 Acknowledgements

We would like to thank the members of GULP, the Italian Association for Logic Programming, who throughout all these years have been for us of great scientific support and encouragement. Their cooperation and friendship are very much appreciated.

Many thanks also to Agostino Dovier and Enrico Pontelli, editors of this book, for their invitation to present the contributions of the program transformation methodology in the field of logic programming.

References

1. M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. A transformation system for lazy functional logic programs. In A. Middeldorp and T. Sato, editors, *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming, FLOPS'99*, Lecture Notes in Computer Science 631, pages 147–162. Springer-Verlag, 1999.
2. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
3. K. R. Apt and D. Pedreschi. Reasoning about termination of pure logic programs. *Information and Computation*, 106:109–157, 1993.
4. C. Aravindan and P. M. Dung. On the correctness of unfold/fold transformation of normal and extended logic programs. *Journal of Logic Programming*, 24(3):201–217, 1995.
5. D. Basin, Y. Deville, P. Flener, A. Hamfelt, and J.F. Nilsson. Synthesis of programs in computational logic. In M. Bruynooghe and K.-K. Lau, editors, *Program Development in Computational Logic*. Springer-Verlag, 2004.
6. A. Bossi and N. Cocco. Basic transformation operations which preserve computed answer substitutions of logic programs. *Journal of Logic Programming*, 16(1&2):47–87, 1993.
7. A. Bossi and N. Cocco. Preserving universal termination through unfold/fold. In *Proceedings ALP '94*, Lecture Notes in Computer Science 850, pages 269–286, Berlin, 1994. Springer-Verlag.
8. A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.
9. 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.

10. A. Bossi, N. Cocco, and S. Etalle. Simultaneous replacement in normal programs. *Journal of Logic and Computation*, 6(1):79–120, 1996.
11. A. Bossi, N. Cocco, and S. Etalle. Transforming left-terminating programs: The reordering problem. In M. Proietti, editor, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '95, Arnhem, The Netherlands*, Lecture Notes in Computer Science 1048, pages 33–45, Berlin, 1996. Springer-Verlag.
12. A. Bossi and S. Etalle. Transforming acyclic programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1081–1096, July 1994.
13. M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, 6:135–162, 1989.
14. M. Bugliesi, E. Lamma, and P. Mello. Partial evaluation for hierarchies of logic theories. In S. Debray and M. Hermenegildo, editors, *Logic Programming: Proceedings of the 1990 North American Conference, Austin, Texas, October 1990*, pages 359–376. The MIT Press, 1990.
15. M. Bugliesi and F. Rossi. Partial evaluation in Prolog: Some Improvements about Cut. In E. L. Lusk and R. A. Overbeek, editors, *Logic Programming: Proceedings of the North American Conference 1989, Cleveland, Ohio, October 1989*, pages 645–660. The MIT Press, 1989.
16. 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.
17. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
18. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
19. 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.
20. 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.
21. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3):250–270, 2001.
22. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
23. S. Etalle, M. Gabbrielli, and E. Marchiori. A transformation system for CLP with dynamic scheduling and CCP. In *PEPM '97*, pages 137–150. ACM Press, 1997.
24. S. Etalle, M. Gabbrielli, and M. C. Meo. Transformations of ccp programs. *ACM Transactions on Programming Languages and Systems*, 23(3):304–395, 2001.
25. 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.
26. F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 292–340. Springer-Verlag, 2004.
27. P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalization of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, 2000.
28. J. P. Gallagher. Transforming programs by specialising interpreters. In *Proceedings Seventh European Conference on Artificial Intelligence, ECAI '86*, pages 109–122, 1986.

29. J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.
30. 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.
31. C. J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, 1981.
32. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
33. H. J. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In *Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA*, pages 255–267, 1982.
34. L. Kott. The McCarthy's induction principle: 'oldy' but 'goody'. *Calcolo*, 19(1):59–69, 1982.
35. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.
36. K.-K. Lau, M. Ornaghi, A. Pettorossi, and M. Proietti. Correctness of logic program transformation based on existential termination. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)*, pages 480–494. MIT Press, 1995.
37. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.
38. G. Levi and G. Sardu. Partial evaluation of meta programs in a multiple worlds logic language. *New Generation Computing*, 6(2&3):227–248, 1988.
39. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
40. M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.
41. Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Toplas*, 2:90–121, 1980.
42. The MAP transformation system. http://www.iasi.cnr.it/∼proietti/system.html, 1995–2010.
43. A. Pettorossi. A powerful strategy for deriving efficient programs by transformation. In *ACM Symposium on Lisp and Functional Programming*, pages 273–281. ACM Press, 1984.
44. 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.
45. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, editor, *Proceedings of the First International Conference on Computational Logic (CL 2000), London, UK, 24-28 July*, Lecture Notes in Artificial Intelligence 1861, pages 613–628. Springer-Verlag, 2000.
46. A. Pettorossi, M. Proietti, and S. Renault. Derivation of efficient logic programs by specialization and reduction of nondeterminism. *Higher-Order and Symbolic Computation*, 18(1-2):121–210, 2005.
47. A. Pettorossi, M. Proietti, and V. Senni. Proving properties of constraint logic programs by eliminating existential variables. In S. Etalle and M. Truszczyński, editors, *Proceedings of the 22nd International Conference on Logic Programming (ICLP '06)*, Lecture Notes in Computer Science 4079, pages 179–195. Springer-Verlag, 2006.
48. A. Pettorossi, M. Proietti, and V. Senni. Automatic correctness proofs for logic program transformations. In V. Dahl and I. Niemelä, editors, *Proceedings of the 23rd International Conference on Logic Programming (ICLP '07)*, Lecture Notes in Computer Science 4670, pages 364–379, 2007.

49. M. Proietti and A. Pettorossi. Semantics preserving transformation rules for Prolog. In *1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '91, Yale University, New Haven, Connecticut, USA*, pages 274–284. ACM Press, 1991.
50. 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.
51. M. O. Rabin. Decidable theories. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, 1977.
52. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. *International Journal on Foundations of Computer Science*, 13(3):387–403, 2002.
53. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM Transactions on Programming Languages and Systems*, 26:264–509, 2004.
54. 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.
55. S. Safra and E. Shapiro. Meta interpreters for real. In H. J. Kugler, editor, *Proceedings Information Processing 86*, pages 271–278. North-Holland, 1986.
56. 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.
57. H. Seki. A comparative study of the well-founded and the stable model semantics: Transformation's viewpoint. In *Proceedings of the Workshop on Logic Programming and Nonmonotonic Logic*, pages 115–123. Cornell University, 1990.
58. H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86:107–139, 1991.
59. H. Seki. Unfold/fold transformation of general logic programs for well-founded semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.
60. H. Seki. On inductive and coinductive proofs via unfold/fold transformations. In *Proceedings of the 15th International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR '09)*. Springer-Verlag, 2009.
61. L. Sterling and R. D. Beer. Incremental flavour-mixing of meta-interpreters for expert system construction. In *Proceedings 3rd International Symposium on Logic Programming, Salt Lake City, Utah, USA*, pages 20–27. IEEE Press, 1986.
62. P. Tacchella, M. Gabbrielli, and M. C. Meo. Unfolding in CHR. In *Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP '07)*, pages 179–186, 2007.
63. A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to meta-programming. In H. J. Kugler, editor, *Proceedings of Information Processing '86*, pages 415–420. North-Holland, 1986.
64. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming (ICLP'84)*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.
65. F. Toni and R. Kowalski. An argumentation-theoretic approach to logic program transformation. In M. Proietti, editor, *Logic Program Synthesis and Transformation, Proceedings LOPSTR '95, Arnhem, The Netherlands.*, Lecture Notes in Computer Science 1048, pages 61–75. Springer-Verlag, 1996.

66. R. Venken. A Prolog meta-interpretation for partial evaluation and its application to source-to-source transformation and query optimization. In T. O'Shea, editor, *Proceedings of ECAI '84*, pages 91–100. North-Holland, 1984.
67. P. L. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
68. J. Zhang and P. W. Grant. An automatic difference-list transformation algorithm for Prolog. In *Proceedings 1988 European Conference on Artificial Intelligence, ECAI '88*, pages 320–325. Pitman, 1988.