Fundamenta Informaticae 127 (2013) 1–20
DOI 10.3233/FI-2013-890
IOS Press

Proving Theorems by Program Transformation

Fabio Fioravanti*
University of Chieti-Pescara, Pescara, Italy
fioravanti@unich.it

Alberto Pettorossi
University of Rome Tor Vergata, Via del Politecnico 1, 00133 Rome, Italy
pettorossi@disp.uniroma2.it

Maurizio Proietti
IASI-CNR, Viale Manzoni 30, 00185 Rome, Italy
maurizio.proietti@iasi.cnr.it

Valerio Senni
IMT, Institute for Advanced Studies, Lucca, Italy
valerio.senni@imtlucca.it

Abstract. In this paper we present an overview of the *unfold/fold proof method*, a method for proving theorems about programs, based on program transformation. As a metalanguage for specifying programs and program properties we adopt constraint logic programming (CLP), and we present a set of transformation rules (including the familiar *unfolding* and *folding* rules) which preserve the semantics of CLP programs. Then, we show how program transformation strategies can be used, similarly to theorem proving tactics, for guiding the application of the transformation rules and inferring the properties to be proved. We work out three examples: (i) the proof of predicate equivalences, applied to the verification of equality between CCS processes, (ii) the proof of first order formulas via an extension of the quantifier elimination method, and (iii) the proof of temporal properties of infinite state concurrent systems, by using a transformation strategy that performs program specialization.

Keywords: Automated theorem proving, program transformation, constraint logic programming, program specialization, bisimilarity, quantifier elimination, temporal logics.

*Address for correspondence: DEC, University of Chieti-Pescara, Viale Pindaro 42, 65127 Pescara, Italy

1. Introduction

Program transformation is a methodology that allows the programmer to separate the correctness concern and the efficiency concern when developing programs [4]. An initial, maybe inefficient, program whose correctness with respect to a given specification can easily be proved, is transformed, possibly in several steps, into an efficient program by applying correctness preserving transformations.

Although its main objective is the improvement of efficiency, it has long been recognized that program transformation can also be used as a methodology for proving program properties and, more in general, for proving theorems. Indeed, in the case of functional or logic programming, programs can be regarded as theories consisting of sets of equations and logical implications, respectively, which are associated with models defined by a suitable program semantics (either least or greatest models). Thus, transforming programs can be regarded as an activity by which one deduces consequences of theories, that is, theorems which hold in the models defined by the given semantics. In this setting, the elementary transformation steps, often called *transformation rules*, can be regarded as inference rules, and composite transformations, often called *transformation strategies*, can be regarded as theorem proving tactics.

The view of program transformation as a theorem proving activity was first suggested in the seminal paper by Burstall and Darlington [4], where some equivalences between functions defined by recursive equations are proved by applying *unfolding* and *folding* transformations. Given a function definition $f(x) = D[x]$, the unfolding rule consists in replacing a function call $f(t)$ occurring in the right hand side of a program equation by the expression $D[t]$. The folding rule is the inverse of the unfolding rule, and consists in replacing an occurrence of the expression $D[t]$ by the function call $f(t)$. In order to prove the equivalence of two functions, say f and g, Burstall and Darlington proposed a method, which we will call the *unfold/fold proof method*, based on program transformations: by applying the unfolding and folding rules, the definitions of f and g are transformed into two syntactically identical sets of equations (modulo the function and variable names) and, additionally, the *termination* of the derived set of equations is proved, to avoid that $f(x)$ and $g(x)$ differ for values of x where the function defined by the new set of equations fails to terminate. (Essentially, this proof method is a transformational version of McCarthy's induction principle [21].)

Burstall and Darlington's unfold/fold proof method for functional programs has been further refined in several papers (see, for instance, [6, 17]). In particular, Kott [17] proposed a method to avoid the termination check, which is hard to automate in general. Kott's method guarantees the soundness of the unfold/fold method by a suitable bookkeeping of the applications of the unfolding and folding rules performed during the proof. Obviously, since program equivalence is undecidable and not even semidecidable, the unfold/fold proof method is necessarily incomplete. However, completeness results for some classes of programs (including equational definitions of regular sets of trees) were presented in [6].

Tamaki and Sato extended the unfold/fold transformation methodology to logic programs in [38]. After their landmark paper, a lot of work has been done to prove the correctness of the transformation rules with respect to the various semantics of logic programs, and to devise strategies of application of the rules which have the objective of improving program efficiency (see [25] for a survey of early work in the area).

Also the unfold/fold proof method has been extended to logic programming to prove equivalences of predicates, instead of functions, that is, first order formulas of the form $\forall X(p(X) \leftrightarrow q(X))$ [26]. This method has been shown to be effective for several verification tasks, such as the verification of properties of parameterized concurrent systems [33]. Moreover, by using the Lloyd-Topor transformation [19],

any first order logic formula can be translated into a logic program with negation, thereby extending the applicability of the unfold/fold proof method to prove any first order formula, not only equivalences [27].

In the context of first order theorem proving, for reasons of efficiency it is often useful to employ specialized theorem provers for specific theories. This is why *Constraint Logic Programming* (CLP) is a very attractive paradigm [15], as it combines general purpose, resolution-based logical reasoning, with dedicated theorem provers (called *solvers* in this framework) for restricted theories of *constraints* (for instance, linear equalities and inequalities over the integers, or the rationals, or the reals, and formulas over the booleans or finite domains). The unfold/fold proof method has also been developed in the case of CLP programs, thereby combining rules and strategies for transforming logic programs with theorem proving techniques that exploit properties of the specific constraint domain [28].

Many *non-classical logics*, such as temporal logics, can be encoded into (constraint) logic programming and, by this encoding, the unfold/fold proof method can be used for proving theorems in those logics. This observation has lead to the design of transformational techniques for proving temporal properties of infinite state concurrent systems [10, 13, 18, 23].

The large variety of contexts where the unfold/fold proof method can be applied witnesses its great generality and flexibility. Besides this, we would like to stress the main technical point that motivates exploring the connections between program transformation and theorem proving: many automated transformation strategies which have been developed with the goal of improving program efficiency can be turned into proof tactics. One notable example is the strategy for eliminating existential variables, whose initial motivation was to avoid the construction of unnecessary data structures when computing with logic programs [31]. The same strategy can also be regarded as a technique for proving theorems by *quantifier elimination* (see, for instance, [32]).

In this paper we overview the unfold/fold proof method in the case of constraint logic programming, and we illustrate the method by means of examples.

2. Transformation Rules for Constraint Logic Programs

In this section we briefly recall the basic notions about constraint logic programs [15] and we present the rules we use for transforming those programs (see also [9, 12, 29, 34, 35]).

2.1. Constraint Logic Programs

We will consider constraint logic programs with linear constraints over the set $\mathbb{R}$ of the real numbers. Note, however, that most of the notions and techniques extend to other constraint domains in a straightforward way.

Constraints are defined as follows. If p_1 and p_2 are linear polynomials with real variables, then $p_1 \geq p_2$ and $p_1 > p_2$ are *atomic constraints*. We will also use the equality '$=$' and the inequalities '$\leq$' and '$<$' defined in terms of '$\geq$' and '$>$' as usual. A *constraint* is either *true*, or *false*, or an atomic constraint, or a *conjunction* of constraints.

An *atom* is an atomic formula of the form $p(t_1, \ldots, t_m)$, where p is a predicate symbol not in $\{\geq, >\}$ and $t_1, \ldots, t_m$, with $m \geq 0$, are terms. A *literal* is either an atom or a negated atom. A *goal* is a (possibly empty) conjunction of literals. A *constrained goal* $c \wedge G$ is a conjunction of a constraint c and a goal G. A CLP *program* is a finite set of *clauses* of the form $A \leftarrow c \wedge G$, where A is an atom and $c \wedge G$ is a

constrained goal. Given a clause $A \leftarrow c \wedge G$, A is the *head* of the clause and $c \wedge G$ is the *body* of the clause. Without loss of generality, we assume that all terms denoting real numbers and occurring in the head of a clause are distinct variables.

The *definition* $Defs(p, P)$ of a predicate p in a program P is the set of all clauses of P whose head predicate is p. A predicate p *depends on* a predicate q in a program P if either in P there is a clause $p(\ldots) \leftarrow c \wedge G$ such that q occurs in G, or there exists a predicate r such that p depends on r in P and r depends on q in P. The *extended definition* $Defs^*(p, P)$ of a predicate p in a program P is the set containing the definition of p and the definitions of all those predicates on which p depends in P.

Given a constraint (or a goal or a constrained goal) φ, by $vars(\varphi)$ we denote the set of variables occurring in φ. Given a clause $\gamma : H \leftarrow c \wedge G$, by $evars(\gamma)$ we denote the set of the *existential variables* of γ, that is, $vars(c \wedge G) - vars(H)$. By $\forall(\varphi)$ we denote the universal closure $\forall X_1 \ldots \forall X_n \, \varphi$, where $vars(\varphi) = \{X_1, \ldots, X_n\}$. Similarly, by $\exists(\varphi)$ we denote the existential closure $\exists X_1 \ldots \exists X_n \, \varphi$.

A *stratification* is a function σ from the set of predicate symbols to the non-negative integers. A stratification σ extends to literals by taking $\sigma(p(\ldots)) =_{def} \sigma(p)$ and $\sigma(\neg A) =_{def} \sigma(A) + 1$. A clause $A \leftarrow c \wedge G$ is *stratified with respect to* σ if for every literal L in G, $\sigma(A) \geq \sigma(L)$. A program P is stratified with respect to σ if every clause of P is. Finally, a program is *stratified* if it is stratified with respect to some stratification function.

Let $T_{\mathbb{R}}$ denote the set of ground terms built from $\mathbb{R}$ and from the function symbols in the language of P. An $\mathbb{R}$-*interpretation* is an interpretation which: (i) has universe $T_{\mathbb{R}}$, (ii) assigns to $+, \times, >, \geq$ the usual meaning in $\mathbb{R}$, and (iii) is the standard Herbrand interpretation [19] for function and predicate symbols different from $+, \times, >, \geq$. We can identify an $\mathbb{R}$-interpretation I with the set of ground atoms (with arguments in $T_{\mathbb{R}}$) which are true in I. We write $\mathbb{R} \models \varphi$ if φ is true in every $\mathbb{R}$-interpretation. A constraint c is *satisfiable* if $\mathbb{R} \models \exists(c)$. A constraint c *entails* a constraint d, denoted $c \sqsubseteq d$, if $\mathbb{R} \models \forall(c \rightarrow d)$.

An $\mathbb{R}$-*model* of a CLP program P is an $\mathbb{R}$-interpretation that makes true every clause of P. Every stratified CLP program P has a unique *perfect model*, denoted $M(P)$, which is constructed as follows (see [3] for a similar definition). Let us consider any stratification σ such that P is stratified with respect to σ. Let $S_0, \ldots, S_n$ be a sequence of programs such that: (i) $\bigcup_{0 \leq k \leq n} S_k = P$, and (ii) for $k = 0, \ldots, n$, S_k is the set of clauses $A \leftarrow c \wedge G$ in P such that $\sigma(A) = k$. We define a sequence of $\mathbb{R}$-interpretations as follows: (i) M_0 is the least $\mathbb{R}$-model of S_0 (note that no negative literals occur in S_0), and (ii) for $0 \leq k < n$, M_{k+1} is the least $\mathbb{R}$-model of S_{k+1} which contains M_k. The $\mathbb{R}$-interpretation M_n is the perfect model of P.

2.2. Transformation Rules for CLP Programs

A *transformation sequence* is a sequence $P_0, \ldots, P_n$ of programs constructed by applying the transformation rules defined below. Without loss of generality, when applying the transformation rules we will feel free to rewrite clauses by: (i) renaming their variables apart (so that distinct clauses have no variables in common), and (ii) rearranging the order and removing repeated occurrences of literals in their bodies. 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 sequence is derived from program P_k by the application of one the following rules R1–R7.

Rule R1 is applied for introducing a new predicate definition.

R1. Definition Introduction. Let us consider a clause of the form: δ: $newp(X_1, \ldots, X_h) \leftarrow c \wedge G$, 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 $c \wedge G$, (iii) every predicate symbol occurring in G also occurs in P_0. Clause δ is called the *definition* of *newp*. By *definition introduction* from program P_k we derive the program $P_{k+1} = P_k \cup \{\delta\}$. 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, *Defs*$_0 = \emptyset$.

The (*positive* or *negative*) *unfolding* rules consist in: (i) replacing an atom A occurring in the body of a clause by the corresponding instance of the disjunction of the bodies of the clauses whose heads unify with A, and (ii) applying suitable boolean laws for deriving clauses.

R2. Positive Unfolding. Let γ: $H \leftarrow c \wedge G_L \wedge A \wedge G_R$ be a clause in program P_k and let γ_1: $K_1 \leftarrow d_1 \wedge B_1$ ..., γ_m: $K_m \leftarrow d_m \wedge B_m$ ($m \geq 0$) be all (renamed apart) 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 $\eta_1, \ldots, \eta_m$ ($m \geq 0$), where for $i = 1, \ldots, m$, η_i is $(H \leftarrow c \wedge d_i \wedge G_L \wedge B_i \wedge G_R)\vartheta_i$. From P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta_1, \ldots, \eta_m\}$.

R3. Negative Unfolding. Let γ: $H \leftarrow c \wedge G_L \wedge \neg A \wedge G_R$ be a clause in program P_k and let γ_1: $K_1 \leftarrow d_1 \wedge B_1$, ..., γ_m: $K_m \leftarrow d_m \wedge B_m$ ($m \geq 0$) be all (renamed apart) clauses in 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: (i) $A = K_1\vartheta_1 = \ldots = K_m\vartheta_m$, that is, for $i = 1, \ldots, m$, A is an instance of K_i, and (ii) for $i = 1, \ldots, m$, *evars*$(\gamma_i) = \emptyset$. From $G_L \wedge \neg((d_1 \wedge B_1)\vartheta_1 \vee \ldots \vee (d_m \wedge B_m)\vartheta_m) \wedge G_R$ we get an equivalent disjunction $Q_1 \vee \ldots \vee Q_r$ of constrained goals, with $r \geq 0$, by first moving $\neg$ inward using De Morgan's law, then replacing every negated atomic constraint of the form $\neg(p_1 \geq p_2)$ by $p_1 < p_2$ and replacing every negated atomic constraint of the form $\neg(p_1 < p_2)$ by $p_1 \geq p_2$, and finally moving $\vee$ outward using distributivity. 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 an instance of the body of the definition of a predicate by the corresponding head.

R4. Positive Folding. Let γ be a clause in P_k and let δ: $K \leftarrow d \wedge B$, where B is a non-empty conjunction of literals, be a (renamed apart) definition in *Defs*$_k$. Suppose that there exists a substitution ϑ such that: (i) γ is of the form $H \leftarrow c \wedge d\vartheta \wedge G_L \wedge B\vartheta \wedge G_R$, and (ii) for every variable $X \in$ *evars*(δ), the following conditions hold: (ii.1) $X\vartheta$ is a variable not occurring in $\{H, c, G_L, G_R\}$, and (ii.2) $X\vartheta$ does not occur in the term $Y\vartheta$, for any variable Y occurring in $d \wedge B$ and different from X. By *folding* γ *using the definition* δ we derive the clause η: $H \leftarrow c \wedge G_L \wedge K\vartheta \wedge G_R$. From P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

R5. Negative Folding. Let γ be a clause in P_k and let δ: $K \leftarrow d \wedge A$, where A is an atom, be a (renamed apart) definition in *Defs*$_k$. Suppose also that there exists a substitution ϑ such that: (i) γ is of the form: $H \leftarrow c \wedge G_L \wedge \neg A\vartheta \wedge G_R$, (ii) $c \sqsubseteq d\vartheta$, and (iii) $vars(A) = vars(K)$. By *folding* γ *using the definition* δ we derive the clause η: $H \leftarrow c \wedge G_L \wedge \neg K\vartheta \wedge G_R$. From P_k we derive the program $P_{k+1} = (P_k - \{\gamma\}) \cup \{\eta\}$.

The *goal replacement rule* allows us to replace a *constrained goal* $c_1 \wedge G_1$ in the body of a clause by a constrained goal $c_2 \wedge G_2$, such that $c_1 \wedge G_1$ and $c_2 \wedge G_2$ are equivalent in the perfect model of P_k.

R6. Goal Replacement. Let γ: $H \leftarrow c \wedge c_1 \wedge G_L \wedge G_1 \wedge G_R$ be a clause in program P_k and assume we have that $M(P_k) \models \forall X\, (\exists Y\, c_1 \wedge G_1 \leftrightarrow \exists Z\, c_2 \wedge G_2)$, where $X = vars(\{H, c, G_L, G_R\})$, $Y =$ *vars*$(c_1 \wedge G_1) - X$, and $Z =$ *vars*$(c_2 \wedge G_2) - X$. By *replacing* $c_1 \wedge G_1$ with $c_2 \wedge G_2$, from γ we derive δ: $H \leftarrow c \wedge c_2 \wedge G_L \wedge G_2 \wedge G_R$, and from P_k we derive $P_{k+1} = (P_k - \{\gamma\}) \cup \{\delta\}$.

The *clause deletion* rule R7 allows us to remove from P_k a redundant clause γ, that is, a clause γ such that $M(P_k) = M(P_k - \{\gamma\})$. In Rule R7 we use the following notions. A clause γ is *subsumed* by a clause of the form $H \leftarrow c \wedge G_1$ if γ is of the form $(H \leftarrow d \wedge G_1 \wedge G_2)\vartheta$, for some substitution ϑ, and $d\vartheta \sqsubseteq c$. A clause *has a false body* if it is of the form $H \leftarrow c \wedge G$ and either $\mathbb{R} \models \neg\exists(c)$ or G has a subconjunction of the form $A \wedge \neg A$. The set of *useless predicates* in a program P is the maximal set U of predicates occurring in P such that p is in U iff every clause γ with head predicated p is of the form $p(\ldots) \leftarrow c \wedge G_1 \wedge q(\ldots) \wedge G_2$ for some q in U. A clause in a program P is *useless* if the predicate of its head is useless in P.

R7. 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:
R7s. γ is subsumed by a clause in $P_k - \{\gamma\}$; **R7f**. γ has a false body; **R7u**. γ is useless in P_k.

A transformation sequence $P_0, \ldots, P_n$ is *correct* if the following conditions hold: (i) $P_0 \cup \mathit{Defs}_n$ and P_n are stratified, and (ii) $M(P_0 \cup \mathit{Defs}_n) = M(P_n)$. Transformation sequences constructed by an unrestricted use of the transformation rules may be not correct. Now we present a correctness result for the class of the so-called *admissible* transformation sequences.

Definition 2.1. (Admissible Transformation Sequence)
1. An atom A in a conjunction G is *σ-maximal* if, for every literal L in G, we have $\sigma(A) \geq \sigma(L)$.
2. A clause δ: $H \leftarrow c \wedge G$ is *σ-tight* if G is of the form $G_1 \wedge A \wedge G_2$, for some σ-maximal atom A and subgoals G_1 and G_2, and $\sigma(H) = \sigma(A)$.
3. A clause η is said to be a *descendant* of a clause γ if *either* η is γ itself *or* there exists a clause δ such that η is *derived from* δ by using a rule in $\{\text{R2}, \text{R3}, \text{R4}, \text{R5}, \text{R6}\}$, and δ is a descendant of γ.
4. Let P_0 be a stratified program and let σ be a stratification for P_0. A transformation sequence $P_0, \ldots, P_n$, with $n \geq 0$, is said to be *admissible* if, for $k = 1, \ldots, n$:
(a) every clause in Defs_k is σ-tight,
(b) if P_k is derived from P_{k-1} by goal replacement (R6) and $c_1 \wedge G_1$ is replaced with $c_2 \wedge G_2$ in the clause $H \leftarrow c \wedge B$, then $\sigma(H) > \sigma(L)$ for every L in G_2, and
(c) if P_k is derived from P_{k-1} by positive folding (R4) of clause γ using clause δ, then: *either* (c.1) the head predicate of γ occurs in P_0, *or* (c.2) γ is a descendant of a clause β in P_j, with $0 < j \leq k-1$, such that β has been derived by positive unfolding of a clause α in P_{j-1} w.r.t. an atom which is σ-maximal in the body of α and whose predicate occurs in P_0.

Theorem 2.2. Every admissible transformation sequence is correct.

This theorem extends to CLP programs the result presented in [29] for *locally stratified* logic programs over the domain of infinite lists. Recall that a program is locally stratified if there exists a function σ from the set of ground atoms to the set of non-negative integers such that, for all ground instances $H \leftarrow c \wedge B$ of a program clause, for all literals L in B, $\sigma(H) \geq \sigma(L)$ (where for all ground atoms A, $\sigma(\neg A) =_{def} \sigma(A)+1$) [3]. For the sake of conciseness, here we have made the more restrictive assumption that programs are stratified. However, Theorem 2.2 can be extended to locally stratified CLP programs in a straightforward way.

Example 2.3. Let us consider the program P_0 made out of the following clauses:

1. $\mathit{prop} \leftarrow \mathit{even}(X) \wedge \neg \mathit{odd}(X+1)$
2. $\mathit{even}(X) \leftarrow X = 0$
3. $\mathit{even}(X) \leftarrow X \geq 2 \wedge \mathit{even}(X-2)$
4. $\mathit{odd}(X) \leftarrow X = 1$
5. $\mathit{odd}(X) \leftarrow X \geq 2 \wedge \mathit{odd}(X-2)$

We take the stratification function σ such that $\sigma(\mathit{prop}) = \sigma(\mathit{even}) > \sigma(\mathit{odd})$. Predicate *prop* holds iff there exists an even number whose successor is not odd. We will now prove that *prop* is false by constructing a suitable transformation sequence starting from P_0. By rule R1 we introduce the clause:

6. $\mathit{newp}(X) \leftarrow \mathit{even}(X) \wedge \neg\, \mathit{odd}(X+1)$

and we derive $P_1 = P_0 \cup \{6\}$. We take $\sigma(\mathit{newp}) = \sigma(\mathit{even})$. Thus, clause 6 is σ-tight and $\mathit{even}(X)$ is a σ-maximal atom in its body. By using rule R2, we unfold clause 6 w.r.t. $\mathit{even}(X)$ and we derive $P_2 = P_0 \cup \{7, 8\}$, where:

7. $\mathit{newp}(X) \leftarrow X = 0 \wedge \neg\, \mathit{odd}(X+1)$
8. $\mathit{newp}(X) \leftarrow X \geq 2 \wedge \mathit{even}(X-2) \wedge \neg\, \mathit{odd}(X+1)$

By applying rule R3, we unfold clause 7 w.r.t. $\neg\, \mathit{odd}(X+1)$, and we get $P_3 = P_0 \cup \{8, 9, 10, 11, 12\}$, where:

9. $\mathit{newp}(X) \leftarrow X = 0 \wedge X+1 < 1 \wedge X+1 < 2$
10. $\mathit{newp}(X) \leftarrow X = 0 \wedge X+1 < 1 \wedge \neg\, \mathit{odd}((X+1)-2)$
11. $\mathit{newp}(X) \leftarrow X = 0 \wedge X+1 > 1 \wedge X+1 < 2$
12. $\mathit{newp}(X) \leftarrow X = 0 \wedge X+1 > 1 \wedge \neg\, \mathit{odd}((X+1)-2)$

Now, clauses 9–12 all have an unsatisfiable conjunction of constraints in their body. Thus, by applying the clause deletion rule R7f, we remove them all and we derive $P_4 = P_0 \cup \{8\}$. Then, by unfolding clause 8 w.r.t. $\neg\, \mathit{odd}(X+1)$ and deleting the clauses with unsatisfiable constraints, we derive $P_4 = P_0 \cup \{13\}$, where:

13. $\mathit{newp}(X) \leftarrow X \geq 2 \wedge X+1 > 1 \wedge \mathit{even}(X-2) \wedge \neg\, \mathit{odd}((X+1)-2)$

By rule R6, we replace the constrained goal $X \geq 2 \wedge X+1 > 1 \wedge \neg\, \mathit{odd}((X+1)-2)$ by the constrained goal $X \geq 2 \wedge \neg\, \mathit{odd}((X-2)+1)$, and we derive $P_5 = P_0 \cup \{14\}$, where:

14. $\mathit{newp}(X) \leftarrow X \geq 2 \wedge \mathit{even}(X-2) \wedge \neg\, \mathit{odd}((X-2)+1)$

By applying rule R4 twice, we fold clauses 1 and 14 using definition 6 and we derive the program $P_6 = \{2, 3, 4, 5, 15, 16\}$, where:

15. $\mathit{prop} \leftarrow \mathit{newp}(X)$
16. $\mathit{newp}(X) \leftarrow X \geq 2 \wedge \mathit{newp}(X-2)$

Finally, clauses 15 and 16 are useless and, by applying rule R7u, can be deleted. Thus, we derive $P_7 = \{2, 3, 4, 5\}$.

The transformation sequence $P_0, \ldots, P_7$ is admissible, and hence by Theorem 2.2 it is correct. In particular, the two applications of rule R4 satisfy Condition (4) of Definition 2.1 because: (i) clause 6 is σ-tight, (ii.1) the head predicate of clause 1 occurs in P_0, and (ii.2) clause 14 is a descendant of clause 8 which has been derived by unfolding w.r.t. a σ-maximal atom whose predicate occurs in P_0. Since the definition of *prop* is the empty set of clauses, we have that *prop* is false in $M(P_7)$. By the correctness of the transformation sequence, we have proved that *prop* is false in $M(P_0)$.

Variants of the above rules have been presented in several papers and correctness results have been proved with respect to various semantics of logic programs and constraint logic programs (see [25] for a survey of early results, and [9, 30, 35] for more recent work). In this section we have presented only the rules that are used in the examples presented in the following sections.

3. Proving Equivalence of CCS Terms

In this section we show the correctness of a mutual exclusion protocol by using the unfold/fold proof method. First, we formalize the operational semantics of the protocol within the Calculus for Communicating Systems (CCS) [22] and we express that semantics using a logic program. Then, we show that the protocol satisfies the mutual exclusion property by showing the equivalence of two predicates.

Let us start by introducing the basic notions of the fragment of CCS we need. For the notions not presented here the reader may refer to [22]. We consider the following sets.
(i) The infinite set A of *names*. For every name $\ell \in A$ we assume that there exists a *co-name*, denoted by $\overline{\ell}$. The set of all co-names is denoted by $\overline{A}$. We assume that for any $\ell \in A$, $\overline{\overline{\ell}} = \ell$. (ii) The set *Act* of *actions*, which is $A \cup \overline{A} \cup \{\tau\}$, where τ is a distinguished element. $\alpha, \beta, \ldots$ range over *Act*. (iii) The set *Id* of *identifiers* which are introduced by *definitions* (see Point (v) below). (iv) The set $\mathcal{P}$ of *processes*, also called *terms*, ranged over by $p, q, p', q', \ldots$, possibly with subscripts, whose syntax is as follows:

$p \in \mathcal{P} \qquad p ::= 0 \mid \alpha.p \mid p_1 + p_2 \mid p_1 \,|\, p_2 \mid p \backslash L \mid P$

where: 0 is a distinguished process, α is an action in *Act*, $L \subseteq A$ is a set of names, and P is a process identifier in *Id*. (v) The set of *definitions* of the form: $P =_{def} p$, where every occurrence of an identifier in p is within a subterm of the form $\alpha.p'$, with α different from τ. We will write $\sum_{i \in I} p_i$ to denote the term $p_1 + (p_2 + (\ldots + p_n)\ldots)$, for $I = \{1, \ldots, n\}$. Every subterm p_i is called a *summand*.

We define the operational semantics of processes by introducing the binary relation $\xrightarrow{\alpha} \subseteq \mathcal{P} \times \mathcal{P}$, for every $\alpha \in Act$. That relation is defined by the following rules of inference.

Prefix: $\alpha.p \xrightarrow{\alpha} p$ Sum: $\dfrac{p_j \xrightarrow{\alpha} q}{\sum_{i \in I} p_i \xrightarrow{\alpha} q}$ if $j \in I$

Parallel Composition: $\dfrac{p_1 \xrightarrow{\alpha} p_1'}{p_1 \,|\, p_2 \xrightarrow{\alpha} p_1' \,|\, p_2}$ $\dfrac{p_2 \xrightarrow{\alpha} p_2'}{p_1 \,|\, p_2 \xrightarrow{\alpha} p_1 \,|\, p_2'}$ (†) $\dfrac{p_1 \xrightarrow{\ell} p_1' \quad p_2 \xrightarrow{\overline{\ell}} p_2'}{p_1 \,|\, p_2 \xrightarrow{\tau} p_1' \,|\, p_2'}$ for any $\ell \in A$

Restriction: $\dfrac{p \xrightarrow{\alpha} q}{p \backslash B \xrightarrow{\alpha} q \backslash B}$ if $\alpha \notin B \cup \overline{B}$, for any set $B \subseteq A$ of names

Identifier: $\dfrac{p \xrightarrow{\alpha} q}{P \xrightarrow{\alpha} q}$ where $P =_{def} p$

From these rules it follows that the parallel composition '$|$' is associative and commutative. If $p \xrightarrow{\alpha} q$, we say that q is an α-*derivative* of p. We have that: (i) $\alpha.p \,|\, \overline{\alpha}.q \xrightarrow{\alpha} p \,|\, \overline{\alpha}.q$, (ii) $\alpha.p \,|\, \overline{\alpha}.q \xrightarrow{\overline{\alpha}} \alpha.p \,|\, q$, and (iii) $\alpha.p \,|\, \overline{\alpha}.q \xrightarrow{\tau} p \,|\, q$. However, due to the restriction '$\backslash\{\alpha\}$', we have that: $(\alpha.p \,|\, \overline{\alpha}.q)\backslash\{\alpha\} \xrightarrow{\tau} p \,|\, q$, and neither an α-derivative nor an $\overline{\alpha}$-derivative exists for $(\alpha.p \,|\, \overline{\alpha}.q)\backslash\{\alpha\}$.

Now we will define the relation $= \subseteq \mathcal{P} \times \mathcal{P}$, called *equality*. It requires the definition of the relation $\approx \subseteq \mathcal{P} \times \mathcal{P}$, called *bisimilarity*, which in turn requires the definition of the relations $\stackrel{\alpha}{\Longrightarrow}$ and $\stackrel{\widehat{\alpha}}{\Longrightarrow}$, for any action $\alpha \in Act$. Let $(\xrightarrow{\tau})^*$ denote the reflexive, transitive closure of $\xrightarrow{\tau}$. Let ε denote the empty sequence of actions in Act^*. We define $\widehat{\tau}$ to be ε, and for any action α different from τ, we define $\widehat{\alpha}$ to be α itself. Then, we define the following two relations which are subsets of $\mathcal{P} \times \mathcal{P}$:
(i) $p \stackrel{\varepsilon}{\Longrightarrow} q$ iff $p\,(\xrightarrow{\tau})^*\,q$ (in particular, for every process p, $p \stackrel{\varepsilon}{\Longrightarrow} p$), and
(ii) $p \stackrel{\alpha}{\Longrightarrow} q$ iff $p\,(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^*\,q$.

For any action $\alpha \in Act$, the definition of $\stackrel{\widehat{\alpha}}{\Longrightarrow}$ follows from Points (i) and (ii) and the definition of $\widehat{\alpha}$.

Definition 3.1. The relation $\approx$ is the *largest* relation such that, for all processes p and q,

$p \approx q$ iff $\forall \alpha \in Act$ (i) $\forall p'$ if $p \stackrel{\alpha}{\longrightarrow} p'$ then ($\exists q'\ q \stackrel{\widehat{\alpha}}{\Longrightarrow} q'$ and $p' \approx q'$) and

(ii) $\forall q'$ if $q \stackrel{\alpha}{\longrightarrow} q'$ then ($\exists p'\ p \stackrel{\widehat{\alpha}}{\Longrightarrow} p'$ and $p' \approx q'$).

If $p \approx q$, we say that p and q are *bisimilar*. The relation $=$ is the relation such that, for all processes p and q,

$p = q$ iff $\forall \alpha \in Act$ (i) $\forall p'$ if $p \stackrel{\alpha}{\longrightarrow} p'$ then ($\exists q'\ q \stackrel{\alpha}{\Longrightarrow} q'$ and $p' \approx q'$) and

(ii) $\forall q'$ if $q \stackrel{\alpha}{\longrightarrow} q'$ then ($\exists p'\ p \stackrel{\alpha}{\Longrightarrow} p'$ and $p' \approx q'$).

If $p = q$, we say that p and q are *equal*.

Let a *context* $\mathcal{C}[\_]$ be a CCS term $\mathcal{C}$ without a subterm. For instance, $([\_] \mid p) + q$ is a context. In general, given any equivalence relation $\sim$, we say that it is a *congruence* iff for all p, q, if $p \sim q$, then for all contexts $\mathcal{C}[\_]$, $\mathcal{C}[p] \sim \mathcal{C}[q]$. We have the following result [22].

Fact 3.1. The relation $\approx$ is an equivalence relation and it is not a congruence. The relation $=$ is an equivalence relation and it is the largest congruence contained in $\approx$.

The following are sound axioms for establishing equality between processes: for all p, q, $r \in \mathcal{P}$, for all actions α, α_i, β_j, for all sets $B \subseteq Act - \{\tau\}$,

1. $p + (q + r) = (p + q) + r$ 2. $p + q = q + p$ 3. $p + p = p$ 4. $p + 0 = p$
5. $\alpha.\tau.p = \alpha.p$ 6. $p + \tau.p = \tau.p$ 7. $\alpha.(p + \tau.q) = \alpha.(p + \tau.q) + \alpha.q$
8. $0\backslash B = 0$ 9. $(p + q)\backslash B = p\backslash B + q\backslash B$ 10. $(\alpha.p)\backslash B =$ *if* $\alpha \in B \cup \overline{B}$ *then* 0 *else* $\alpha.(p\backslash B)$

Let p be $\sum_{i \in I}\ \alpha_i.p_i$ and q be $\sum_{j \in J}\ \beta_j.q_j$. Then,

11. $p \mid q = \sum_{i \in I} \alpha_i.(p_i \mid q) + \sum_{j \in J} \beta_j.(p \mid q_j) + \sum_{i \in I,\, j \in J,\, \alpha_i = \overline{\beta}_j} \tau.(p_i \mid q_j)$.

As a consequence of Axioms 8–11, we get the following equality, called *Expansion Theorem*. For $i = 1, \ldots, m$, let p_i be a process of the form $\sum_{j \in J}\ \alpha_{ij}.p'_{ij}$. Then,

$$(p_1 | \ldots | p_m)\backslash B = \sum\ \alpha_{ij}.(p_1 | \ldots | p'_{ij} | \ldots | p_m)\backslash B\ +\ \sum\ \tau.(p_1 | \ldots | p'_{rh} | \ldots | p'_{sk} | \ldots | p_m)\backslash B$$

where: (i) the left summation is over all $i \in \{1, \ldots, m\}$ and all summands $\alpha_{ij}.p'_{ij}$ of p_i with $\alpha_{ij} \notin B \cup \overline{B}$, and (ii) the right summation is over all distinct $r, s \in \{1, \ldots, m\}$, all summands $\alpha.p'_{rh}$ of p_r, and all summands $\overline{\alpha}.p'_{sk}$ of p_s with $\alpha \in B \cup \overline{B}$.

We define the semantics of any given process p to be a (finite or infinite) tree t, called a *behaviour tree*, which has the following syntax: $t ::= 0 \mid \alpha.t \mid t_1 + t_2 \mid \bot$

where: (i) 0 is the *empty behaviour tree*, (ii) for every $\alpha \in Act$, $\alpha.\_$ is a unary constructor, (iii) $\_ + \_$ is a binary constructor, which is assumed to be associative, commutative, idempotent, with identity 0, and (iv) $\bot$ is the *undefined behaviour tree*. The semantics of a process p of the form either 0, or $\alpha.p'$, or $p_1 + p_2$, is the process itself, when viewed as a behaviour tree. The semantics of a process p involving parallel composition and restriction is the semantics of the process obtained from p by applying the Expansion Theorem (which replaces $\mid$ in favour of $+$). The semantics of a process identifier P, defined by $P =_{def} p$, is the semantics of p, and thus the semantics of a recursively defined process is, in general, an infinite, 'periodic' behaviour tree. For instance, (i) the semantics of $\alpha.0$ is $\alpha.0$, (ii) the semantics of P

defined by $P =_{def} \alpha.Q$ and $Q =_{def} \beta.0$, is $\alpha.\beta.0$, and (iii) the semantics of R defined by $R =_{def} \alpha.\beta.R$, is the infinite tree $\alpha.\beta.\alpha.\beta.\ldots$

The behaviour tree $\bot$ has been introduced to avoid the explicit reference to infinite behaviour trees, as we now explain. First, we need the following definition. An *approximation* of a behaviour tree t is either t itself or a tree obtained from t by replacing one or more of its subtrees by $\bot$.

Then, for any process p and any behaviour tree t, we introduce the predicate $b(p,t)$ which holds if and only if t is a *finite* approximation of the behaviour tree of process p. We list below (see clauses 1–5.6) some of the clauses that define $b(p,t)$. Clauses 1–3 refer to processes involving 0, $\alpha._$, and $\_+\_$ only. Clauses 4.1–4.3 refer to parallel composition of processes. Clauses 5.1–5.6 refer to process identifiers. In clauses 1–5.6 we assume that: (i) B is any subset of *Act*$-\{\tau\}$ and $\widetilde{B}$ denotes the set $B \cup \overline{B}$, (ii) actions α, β, and γ are pairwise distinct, and (iii) process $\mathit{id}(P)$ is defined by $P =_{def} p$ and, for $i = 1,2,3$, process $\mathit{id}(P_i)$ is defined by $P_i =_{def} p_i$.

1. $b(0,0) \leftarrow$ 2. $b(\alpha.P,\ \alpha.T) \leftarrow b(P,T)$ 3. $b(P_1{+}P_2,\ T_1{+}T_2) \leftarrow b(P_1,T_1)\ \wedge\ b(P_2,T_2)$

4.1 $b((\overline{\alpha}.P_1 \mid \overline{\alpha}.P_2 \mid \alpha.P_3)\backslash B,\ \tau.T_1{+}\tau.T_2) \leftarrow$
$b((P_1 \mid \overline{\alpha}.P_2 \mid P_3)\backslash B,\ T_1) \wedge b((\overline{\alpha}.P_1 \mid P_2 \mid P_3)\backslash B,\ T_2)$ for all $\alpha \in \widetilde{B}$

4.2 $b((\gamma.P_1 \mid \overline{\alpha}.P_2 \mid \beta.P_3)\backslash B,\ \gamma.T) \leftarrow b((P_1 \mid \overline{\alpha}.P_2 \mid \beta.P_3)\backslash B,\ T)$ for all $\alpha, \beta \in \widetilde{B}$ and $\gamma \notin \widetilde{B}$

4.3 $b((\overline{\alpha}.P_1 \mid \gamma.P_2 \mid \beta.P_3)\backslash B,\ \gamma.T) \leftarrow b((\overline{\alpha}.P_1 \mid P_2 \mid \beta.P_3)\backslash B,\ T)$ for all $\alpha, \beta \in \widetilde{B}$ and $\gamma \notin \widetilde{B}$

5.1 $b(\mathit{id}(P), \bot) \leftarrow$ 5.2 $b(\mathit{id}(P), T) \leftarrow b(p,T)$

5.3 $b((\mathit{id}(P_1) \mid P_2 \mid \mathit{id}(P_3))\backslash B,\ \bot) \leftarrow$ 5.4 $b((\mathit{id}(P_1) \mid P_2 \mid \mathit{id}(P_3))\backslash B,\ T) \leftarrow b((p_1 \mid P_2 \mid p_3)\backslash B,\ T)$

5.5 $b((P_1 \mid \mathit{id}(P_2) \mid \mathit{id}(P_3))\backslash B,\ \bot) \leftarrow$ 5.6 $b((P_1 \mid \mathit{id}(P_2) \mid \mathit{id}(P_3))\backslash B,\ T) \leftarrow b((P_1 \mid p_2 \mid p_3)\backslash B,\ T)$

Note that Clauses 4.1–4.3 and Clauses 5.3–5.6 are particular instances of more general clauses that one can introduce for defining the semantics of parallel composition of processes and process identifiers. We considered these instances because they allow a shorter proof in the example we will present below.

We also introduce, for any two behaviour trees t_1 and t_2, the predicate $\mathit{eq}(t_1,t_2)$ which holds iff the equality $t_1 = t_2$ follows from Axioms 1–7 (which, among Axioms 1–11, are the ones involving $\alpha._$ and $\_+\_$ only) by considering behaviour trees rather than processes. (Note that for our unfold/fold proof, when applying the goal replacement rule, we need to know only some valid equivalences holding for *eq*, not the clauses defining *eq*.) We have the following Fact (A): for all processes p and q, if for all finite behaviour trees t, $\exists t_1(b(p,t_1) \wedge \mathit{eq}(t_1,t))$ iff $\exists t_2(b(q,t_2) \wedge \mathit{eq}(t_2,t))$, then $p = q$ (in the sense of Definition 3.1).

Now, by using the unfold/fold proof method, we will prove the correctness of a simple locking protocol for mutual exclusion [14]. We consider the predicate $b(\mathit{Sys}, T_1)$ that defines the operational semantics of the protocol (denoted by the CCS term *Sys*), and the predicate $b(\mathit{Mutex}, T_2)$ that defines the mutual exclusion property (denoted by the CCS term *Mutex*). Then, we consider the predicates *new*1(T) and *new*2(T) defined by the clauses *new*1$(T) \leftarrow b(\mathit{Sys}, T_1) \wedge \mathit{eq}(T_1, T)$ and *new*2$(T) \leftarrow b(\mathit{Mutex}, T_2) \wedge \mathit{eq}(T_2, T)$, respectively. By constructing an admissible transformation sequence using the program transformation rules of Section 2, we will derive for *new*1 and *new*2 two identical sets of clauses (modulo the name of the predicates), and hence $M(\mathit{Beq}) \models \exists T_1(b(\mathit{Sys}, T_1) \wedge \mathit{eq}(T_1, T)) \leftrightarrow \exists T_2(b(\mathit{Mutex}, T_2) \wedge \mathit{eq}(T_2, T))$, where $M(\mathit{Beq})$ is the perfect model of the program *Beq* made out of the clauses for b and *eq*. Thus, by Fact (A) above, we have that *Sys* $=$ *Mutex*, and this proves mutual exclusion.

In the protocol we consider, we have two processes, a reader process R and a writer process W, which want to access a common store. They are defined as follows: $R =_{def} r_1.r_2.R$ and $W =_{def} w_1.w_2.W$.

The purpose of the protocol is to ensure mutual exclusion, that is, in every sequence of actions neither action w_1 nor action w_2 should occur in between the two actions r_1 and r_2, and symmetrically, neither r_1 nor r_2 should occur in between w_1 and w_2. The parallel composition $(R \,|\, W)$ does *not* ensure mutual exclusion. Indeed, for instance, we have that: $(R \,|\, W) \xrightarrow{r_1} \xrightarrow{w_1} \xrightarrow{r_2} (R \,|\, W)$. In order to ensure mutual exclusion, (i) we consider the extra process: $L =_{def} l.u.L$, (where l stands for *lock* and u for *unlock*), and (ii) we modify the processes R and W by requiring that they should get the lock from L before their actions (by performing the action $\overline{l}$) and should release it to L afterwards (by performing the action $\overline{u}$). Thus, we get the following two modified processes: $R' =_{def} \overline{l}.r_1.r_2.\overline{u}.R'$ and $W' =_{def} \overline{l}.w_1.w_2.\overline{u}.W'$.

Now, processes R' and W', when composed in parallel with process L, can access the common store in a mutually exclusive way only. Indeed, in particular, if the process L wants to perform the action l, then by the parallel composition rule $(\dagger)$, *only one* of the two processes R' and W' can engage in that action with process L by performing $\overline{l}$. We will formally prove that mutual exclusion is ensured by showing that the process, called *Sys*, which is the term $(\overline{l}.r_1.r_2.\overline{u}.R' \mid \overline{l}.w_1.w_2.\overline{u}.W' \mid l.u.L)\backslash\{l, u\}$ is equal (in the sense of the equality relation $=$ of Definition 3.1) to the following process specifying the desired mutually exclusive access to the store: *Mutex* $=_{def} \tau.r_1.r_2.$*Mutex* $+ \tau.w_1.w_2.$*Mutex*. Note that, in contrast to the process *Mutex*1 defined as: *Mutex*1 $=_{def} r_1.r_2.$*Mutex*1 $+ w_1.w_2.$*Mutex*1, our process *Mutex* gives to the store the extra possibility of deciding 'of its own volition' to give access either to the reader or to the writer.

The unfold/fold proof starts off by introducing, using rule R1, the following two predicates *new*1(T) and *new*2(T):

1. *new*1$(T) \leftarrow b((\overline{l}.r_1.r_2.\overline{u}.$*id*$(R') \mid \overline{l}.w_1.w_2.\overline{u}.$*id*$(W') \mid l.u.$*id*$(L))\backslash\{l, u\},\ T_1) \wedge$ *eq*(T_1, T)
2. *new*2$(T) \leftarrow b(\tau.r_1.r_2.$*id*$($*Mutex*$) + \tau.w_1.w_2.$*id*$($*Mutex*$),\ T_2) \wedge$ *eq*(T_2, T)

The bodies of clauses 1 and 2 define, indeed, the system *Sys* and the property *Mutex*, respectively. As a consequence of its definition, the predicate *eq* satisfies the following equivalences, which we need below in the unfold/fold proof (free variables are assumed to be universally quantified at the front):

E1. for all $\alpha \in$ *Act*, *eq*$(\alpha.\tau.T_1, T) \leftrightarrow$ *eq*$(\alpha.T_1, T)$ (see Axiom 5 above)

E2. for all $u, v \in$ *Act*$^+$, *eq*$(u.T_1 + v.T_2, T) \leftrightarrow \exists U_1 \exists U_2$ (*eq*$(T_1, U_1) \wedge$ *eq*$(T_2, U_2) \wedge$ *eq*$(u.U_1 + v.U_2, T))$

together with the equivalences which axiomatize the fact that *eq* is a congruence. We deal with predicate *new*1 first. By applying rule R2 we unfold clause 1 using the definitions of R', W', and L. We get:

1.1 *new*1$(T) \leftarrow b((\overline{l}.r_1.r_2.\overline{u}.$*id*$(R') \mid \overline{l}.w_1.w_2.\overline{u}.$*id*$(W') \mid l.u.$*id*$(L))\backslash C,\ T_1) \wedge$ *eq*$(T_1,\ T)$

where C denotes the set $\{l, u\}$. By unfolding clause 1.1 using clause 4.1, we get:

1.2 *new*1$(T) \leftarrow b((r_1.r_2.\overline{u}.$*id*$(R') \mid \overline{l}.w_1.w_2.\overline{u}.$*id*$(W') \mid u.$*id*$(L))\backslash C,\ T_{11})$
$\wedge\ b((\overline{l}.r_1.r_2.\overline{u}.$*id*$(R') \mid w_1.w_2.\overline{u}.$*id*$(W') \mid u.$*id*$(L))\backslash C,\ T_{12}) \wedge$ *eq*$(\tau.T_{11} + \tau.T_{12},\ T)$

After a few unfolding steps using clauses 4.1, 4.2, and 4.3, from clause 1.2 we get:

1.3 *new*1$(T) \leftarrow b((id(R') \mid \overline{l}.w_1.w_2.\overline{u}.$*id*$(W') \mid id(L))\backslash C,\ T_{11})$
$\wedge\ b((\overline{l}.r_1.r_2.\overline{u}.$*id*$(R') \mid id(W') \mid id(L))\backslash C,\ T_{12}) \wedge$ *eq*$(\tau.r_1.r_2.\tau.T_{11} + \tau.w_1.w_2.\tau.T_{12},\ T)$

By applying the goal replacement rule R6 based on (E1) and on the congruence axioms for *eq*, we get:

1.4 *new*1$(T) \leftarrow b((id(R') \mid \overline{l}.w_1.w_2.\overline{u}.$*id*$(W') \mid id(L))\backslash C,\ T_{11})$
$\wedge\ b((\overline{l}.r_1.r_2.\overline{u}.$*id*$(R') \mid id(W') \mid id(L))\backslash C,\ T_{12}) \wedge$ *eq*$(\tau.r_1.r_2.T_{11} + \tau.w_1.w_2.T_{12},\ T)$

Then, by a few more unfolding steps from clause 1.4 using clauses 5.3–5.6, we get:

1.5 *new*1$(T) \leftarrow$ *eq*$(\tau.r_1.r_2.\bot + \tau.w_1.w_2.\bot,\ T)$ $(*)$

1.6 *new*1$(T) \leftarrow b((\overline{l}.r_1.r_2.\overline{u}.$*id*$(R') \mid \overline{l}.w_1.w_2.\overline{u}.$*id*$(W') \mid l.u.$*id*$(L))\backslash C, T_{12})$
$\wedge$ *eq*$(\tau.r_1.r_2.\bot + \tau.w_1.w_2.T_{12},\ T)$

1.7 $\mathit{new}1(T) \leftarrow b((\overline{l}.r_1.r_2.\overline{u}.\mathit{id}(R') \mid \overline{l}.w_1.w_2.\overline{u}.\mathit{id}(W') \mid l.u.\mathit{id}(L))\backslash C, T_{11})$
$\wedge\ \mathit{eq}(\tau.r_1.r_2.T_{11}{+}\tau.w_1.w_2.\bot,\ T)$
1.8 $\mathit{new}1(T) \leftarrow b((\overline{l}.r_1.r_2.\overline{u}.\mathit{id}(R') \mid \overline{l}.w_1.w_2.\overline{u}.\mathit{id}(W') \mid l.u.\mathit{id}(L))\backslash C, T_{11})$
$\wedge\ b((\overline{l}.r_1.r_2.\overline{u}.\mathit{id}(R') \mid \overline{l}.w_1.w_2.\overline{u}.\mathit{id}(W') \mid l.u.\mathit{id}(L))\backslash C, T_{12})$
$\wedge\ \mathit{eq}(\tau.r_1.r_2.T_{11}{+}\tau.w_1.w_2.T_{12},\ T)$

By applying the goal replacement rule based on (E2) (and instances of it) to clauses 1.6–1.8, we get:

1.6r $\mathit{new}1(T) \leftarrow b((\overline{l}.r_1.r_2.\overline{u}.\mathit{id}(R') \mid \overline{l}.w_1.w_2.\overline{u}.\mathit{id}(W') \mid l.u.\mathit{id}(L))\backslash C, T_{12}) \wedge \mathit{eq}(T_{12}, U_{12})$
$\wedge\ \mathit{eq}(\tau.r_1.r_2.\bot{+}\tau.w_1.w_2.U_{12},\ T)$
1.7r $\mathit{new}1(T) \leftarrow b((\overline{l}.r_1.r_2.\overline{u}.\mathit{id}(R') \mid \overline{l}.w_1.w_2.\overline{u}.\mathit{id}(W') \mid l.u.\mathit{id}(L))\backslash C, T_{11}) \wedge \mathit{eq}(T_{11}, U_{11})$
$\wedge\ \mathit{eq}(\tau.r_1.r_2.U_{11}{+}\tau.w_1.w_2.\bot,\ T)$
1.8r $\mathit{new}1(T) \leftarrow b((\overline{l}.r_1.r_2.\overline{u}.\mathit{id}(R') \mid \overline{l}.w_1.w_2.\overline{u}.\mathit{id}(W') \mid l.u.\mathit{id}(L))\backslash C, T_{11}) \wedge \mathit{eq}(T_{11}, U_{11})$
$\wedge\ b((\overline{l}.r_1.r_2.\overline{u}.\mathit{id}(R') \mid \overline{l}.w_1.w_2.\overline{u}.\mathit{id}(W') \mid l.u.\mathit{id}(L))\backslash C, T_{12}) \wedge \mathit{eq}(T_{12}, U_{12})$
$\wedge\ \mathit{eq}(\tau.r_1.r_2.U_{11}{+}\tau.w_1.w_2.U_{12},\ T)$

By applying rule R4 and folding clauses 1.6r–1.8r using clause 1, we get:

1.6f $\mathit{new}1(T) \leftarrow \mathit{new}1(U_{12}) \wedge \mathit{eq}(\tau.r_1.r_2.\bot{+}\tau.w_1.w_2.U_{12},\ T)$ $(*)$
1.7f $\mathit{new}1(T) \leftarrow \mathit{new}1(U_{11}) \wedge \mathit{eq}(\tau.r_1.r_2.U_{11}{+}\tau.w_1.w_2.\bot,\ T)$ $(*)$
1.8f $\mathit{new}1(T) \leftarrow \mathit{new}1(U_{11}) \wedge \mathit{new}1(U_{12}) \wedge \mathit{eq}(\tau.r_1.r_2.U_{11}{+}\tau.w_1.w_2.U_{12},\ T)$ $(*)$

Now we deal with predicate *new*2. Starting from clause 2 we perform a derivation similar to the one we have performed starting from clause 1. By unfolding clause 2, we get:

2.1 $\mathit{new}2(T) \leftarrow b(\tau.r_1.r_2.\mathit{id}(\mathit{Mutex}),\ T_{21}) \wedge b(\tau.w_1.w_2.\mathit{id}(\mathit{Mutex}),\ T_{22}) \wedge \mathit{eq}(T_{21}{+}T_{22},\ T)$

After a few unfolding steps, from clause 2.1 we get the following clauses:

2.2 $\mathit{new}2(T) \leftarrow \mathit{eq}(\tau.r_1.r_2.\bot{+}\tau.w_1.w_2.\bot,\ T)$ $(**)$
2.3 $\mathit{new}2(T) \leftarrow b(\tau.r_1.r_2.\mathit{id}(\mathit{Mutex}) + \tau.w_1.w_2.\mathit{id}(\mathit{Mutex}), T_{22}) \wedge \mathit{eq}(\tau.r_1.r_2.\bot{+}\tau.w_1.w_2.T_{22},\ T)$
2.4 $\mathit{new}2(T) \leftarrow b(\tau.r_1.r_2.\mathit{id}(\mathit{Mutex}) + \tau.w_1.w_2.\mathit{id}(\mathit{Mutex}), T_{21}) \wedge \mathit{eq}(\tau.r_1.r_2.T_{21}{+}\tau.w_1.w_2.\bot,\ T)$
2.5 $\mathit{new}2(T) \leftarrow b(\tau.r_1.r_2.\mathit{id}(\mathit{Mutex}) + \tau.w_1.w_2.\mathit{id}(\mathit{Mutex}), T_{21})$
$\wedge\ b(\tau.r_1.r_2.\mathit{id}(\mathit{Mutex}) + \tau.w_1.w_2.\mathit{id}(\mathit{Mutex}), T_{22}) \wedge \mathit{eq}(\tau.r_1.r_2.T_{21}{+}\tau.w_1.w_2.T_{22},\ T)$

Then, by applying the goal replacement rule based on (E2) (and instances of it) to clauses 2.3–2.5, and by folding the derived clauses using clause 2, we get:

2.3f $\mathit{new}2(T) \leftarrow \mathit{new}2(U_{22}) \wedge \mathit{eq}(\tau.r_1.r_2.\bot{+}\tau.w_1.w_2.U_{22},\ T)$ $(**)$
2.4f $\mathit{new}2(T) \leftarrow \mathit{new}2(U_{21}) \wedge \mathit{eq}(\tau.r_1.r_2.U_{21}{+}\tau.w_1.w_2.\bot,\ T)$ $(**)$
2.5f $\mathit{new}2(T) \leftarrow \mathit{new}2(U_{21}) \wedge \mathit{new}2(U_{22}) \wedge \mathit{eq}(\tau.r_1.r_2.U_{21}{+}\tau.w_1.w_2.U_{22},\ T)$ $(**)$

The transformation sequence from clause 1 to the clauses marked with $(*)$ and the transformation sequence from clause 2 to the clauses marked with $(**)$, constructed by applying the transformation rules of Section 2, are admissible. Indeed, by taking $\sigma(\mathit{new}1) = \sigma(\mathit{new}2) = \sigma(b) > \sigma(\mathit{eq})$, Condition 4 of Definition 2.1 is satisfied. Thus, those sequences are correct.

Since the clauses marked with $(*)$ are equal (modulo the names *new*1 and *new*2) to the clauses marked with $(**)$, we conclude that the given system *Sys* satisfies the mutual exclusion property.

4. Proving First-Order Formulas

In this section we illustrate a transformation technique for checking whether or not a first order property φ holds in the perfect model $M(P)$ of a stratified CLP program P, that is, whether or not $M(P) \models \varphi$ holds.

In particular, we show how unfold/fold program transformations can be used to extend to CLP programs the quantifier elimination technique used for proving theorems in first order logic [32].

The basic idea of our technique is to transform a given formula φ with quantified variables into CLP clauses with existential variables (that is, variables occurring in the body of a clause and not in its head), and then to apply the Existential Variable Elimination strategy (EVE) [28, 31] to eliminate those variables, hence deriving a propositional program Q. Then we can check whether or not $M(P) \models \varphi$ holds by constructing the perfect model of Q. Since $M(P) \models \varphi$ is in general undecidable, the EVE strategy may not terminate. The EVE strategy requires that the theory of constraints occurring in a stratified CLP program admits quantifier elimination and, indeed, this is the case for the theory of constraints introduced in Section 2, also known as the theory of Linear Real Arithmetic (LRA) [20].

Given a stratified program P with no existential variables and a closed first order formula φ, our method for proving whether or not $M(P) \models \varphi$ holds consists of the following two steps.

Step 1. We transform the formula $p \leftarrow \varphi$, where p is a predicate symbol not occurring in P and φ, into a set $D(p, \varphi)$ of clauses such that $M(P) \models \varphi$ iff $M(P \cup D(p, \varphi)) \models p$. This step is done by applying a variant of the Lloyd-Topor transformation [19].

Step 2. We derive from $P \cup D(p, \varphi)$ a *propositional*, stratified logic program Q such that $M(P \cup D(p, \varphi)) \models p$ iff $M(Q) \models p$. This step is done by applying the transformation rules of Section 2 according to the EVE strategy.

If Step 2 terminates, the perfect model of Q is a finite set that can be constructed in finite time, and thus in finite time we can check whether or not $M(Q) \models p$ holds by checking whether or not $p \in M(Q)$.

The EVE strategy is an extension to CLP programs of the UDF strategy for logic programs [31] (further details can be found in [28]). The set $D(p, \varphi)$ constructed at the end of Step 1 is a set $\{D_1, \ldots, D_n\}$ of clauses such that, for $i = 1, \ldots, n$, (i) the head predicate of D_i does not occur in $P \cup \{D_1, \ldots, D_{i-1}\}$, and (ii) every predicate symbol in the body of D_i occurs in $P \cup \{D_1, \ldots, D_{i-1}\}$.

The Existential Variable Elimination Strategy EVE.

Input: A stratified program P and the set $D(p, \varphi) = \{D_1, \ldots, D_n\}$ of definitions generated by the Lloyd-Topor transformation from $p \leftarrow \varphi$.

Output: A propositional program Q such that $M(P \cup D(p, \varphi)) \models p$ iff $M(Q) \models p$.

$T := P$;
FOR $i = 1, \ldots, n$ DO
  Defs $:= \{D_i\}$; *InDefs* $:= \{D_i\}$;
  WHILE *InDefs* $\neq \emptyset$ DO LET $D \in$ *InDefs* IN
    IF *evars*$(D) \neq \emptyset$ THEN *Unfold*(D, T, Us); *Simplify*(Us, Ss); *Define-Fold*$(Ss, Defs, NewDefs, Fs)$
                ELSE *Fs* $:= \{D\}$; *NewDefs* $= \emptyset$ FI;
    $T := T \cup$ *Fs*; *Defs* $:=$ *Defs* $\cup$ *NewDefs*; *InDefs* $:= ($*InDefs* $- \{D\}) \cup$ *NewDefs*
  OD
OD;
$Q =$ *Defs*$^*(p, T)$

For each definition D_i, the strategy iteratively applies the following three procedures: *Unfold*, *Simplify*, and *Define-Fold*.

■ *Unfold*. A definition clause D containing existential variables is unfolded w.r.t. each (positive or negative) literal in the body using program T (initially T is the input program P), thereby deriving a set *Us* of clauses. When unfolding, we use the rules R2 and R3. Note that, as indicated in [28], Condition (i)

of rule R3 (requiring a suitable instantiation of negative literals) can be satisfied by imposing syntactic restrictions on programs and by using a specific unfolding strategy. Condition (ii) of rule R3 (requiring the absence of existential variables in the body of the clauses used for unfolding) is satisfied because the initial program P has no existential variables and the program T derived at the end of each iteration of the WHILE-loop has no existential variables either.

■ *Simplify*. The set *Us* of clauses obtained by unfolding is simplified, thereby deriving a new set *Ss* of clauses as follows: (1) we apply rule R6 and exploit the fact that LRA admits quantifier elimination, for removing implied subconstraints and existential variables occurring in constraints only, and (2) we apply rules R7s and R7f for deleting subsumed clauses and clauses with a false body.

■ *Define-Fold*. By using rule R4 each clause $\gamma \in Ss$ that contains an existential variable is folded so that the derived clause has no existential variables. Folding is performed by using: (1) either a previously introduced definition in the set *Defs*, (2) or a suitable new definition introduced by applying rule R1. Note that, in Case (2), Conditions (i) and (ii) of rule R4 are satisfied by introducing a new definition whose body is the smallest constrained subgoal of the body of γ containing all occurrences of the existential variables of γ (see [28] for details). The new definitions and the folded clauses are collected in the sets *NewDefs* and *Fs*, respectively. Note that the new definitions that are introduced by the *Define-Fold* procedure are added to the set *InDefs*, and thus require further iterations of the body of the WHILE-loop of the EVE strategy.

Each iteration of the body of the WHILE-loop terminates and produces a program T with no existential variables, at the expense of possibly introducing new predicate definitions. The strategy terminates when no new definitions are introduced. It may be the case that an unbounded number of new definitions has to be introduced and the strategy does not terminate.

Let T^0 be program P and, for $i = 1, \ldots, n$, let T^i and Defs^i denote, respectively, the program T and the set *Defs* at the end of the i-th iteration of the FOR-loop. The proof of correctness of the EVE strategy is based on the fact that, for $i = 1, \ldots, n$, program T^i has been obtained from program T^{i-1} via an admissible transformation sequence, and hence, by Theorem 2.2, $M(T^{i-1} \cup \mathit{Defs}^i) = M(T^i)$.

The last step of the EVE strategy consists in deriving the program Q by taking the clauses in the extended definition Defs^* of p in T.

Now we illustrate how the EVE strategy works by means of an example. Let us consider the theory of finite, ordered lists of real numbers, defined by the following clauses:

1. $\mathit{ord}([\,]) \leftarrow$
2. $\mathit{ord}([A|L]) \leftarrow \mathit{ord}_1(A, L)$
3. $\mathit{ord}_1(A, [\,]) \leftarrow$
4. $\mathit{ord}_1(A, [B|L]) \leftarrow A \leq B \wedge \mathit{ord}_1(B, L)$
5. $\mathit{nth}([A|L], P, E) \leftarrow P = 0 \wedge A = E$
6. $\mathit{nth}([A|L], P, E) \leftarrow P > 0 \wedge \mathit{nth}(L, P-1, E)$
7. $\mathit{el}([\,], [\,]) \leftarrow$
8. $\mathit{el}([A|L], [B|M]) \leftarrow \mathit{el}(L, M)$

where: (i) $\mathit{ord}(L)$ holds the list L is ordered, (ii) $\mathit{nth}(L, P, E)$ holds iff in the list L the element at position P is E, and (iii) $\mathit{el}(L, M)$ holds iff the lists L and M have equal length. Let us consider the following property φ: given two ordered lists X and Y of equal length, the list Z obtained by element-wise sum of the elements in X and Y, is ordered. Thus, $\varphi \equiv \forall X \, \forall Y \, \forall Z \; \mathit{el}(X, Y) \wedge \mathit{el}(X, Z) \wedge \psi(X, Y, Z) \wedge \mathit{ord}(X) \wedge \mathit{ord}(Y) \rightarrow \mathit{ord}(Z)$, where the subformula ψ defines Z in terms of element-wise sum of X and Y, that is, $\psi(X, Y, Z) \equiv \forall P \, \forall E \, \forall F \; \mathit{nth}(X, P, E) \wedge \mathit{nth}(Y, P, F) \rightarrow \mathit{nth}(Z, P, E+F)$.

Step 1. By Lloyd-Topor transformation from the formula $p \leftarrow \varphi$ we derive the following set $D(p, \varphi)$ of definitions:

9. $r(X, Y, Z) \leftarrow \mathit{nth}(X, P, E) \wedge \mathit{nth}(Y, P, F) \wedge \neg \mathit{nth}(Z, P, E+F)$

10. $q \leftarrow \mathit{el}(X,Y) \wedge \mathit{el}(X,Z) \wedge \neg r(X,Y,Z) \wedge \mathit{ord}(X) \wedge \mathit{ord}(Y) \wedge \neg \mathit{ord}(Z)$
11. $p \leftarrow \neg q$

Step 2. The inputs of the EVE strategy are the program $P = \{1,2,3,4,5,6,7,8\}$ and the set $D = \{9,10,11\}$. The strategy starts off by applying to clause 9 a single step of (positive or negative) unfolding (that is, rule R2 or rule R3) w.r.t. each literal. We get:

12. $r([A|B],[C|D],[\,]) \leftarrow$
13. $r([A|B],[C|D],[E|F]) \leftarrow E \neq A+C$
14. $r([A|B],[C|D],[\,]) \leftarrow E \geq 0 \wedge \mathit{nth}(B,E,F) \wedge \mathit{nth}(D,E,G)$
15. $r([A|B],[C|D],[E|F]) \leftarrow G \geq 0 \wedge G < 0 \wedge \mathit{nth}(B,G,H) \wedge \mathit{nth}(D,G,I)$
16. $r([A|B],[C|D],[E|F]) \leftarrow G \geq 0 \wedge \mathit{nth}(B,G,H) \wedge \mathit{nth}(D,G,I) \wedge \neg \mathit{nth}(F,G,H+I)$

Clause 13 with $\neq$ in its body stands for two different clauses, one with $>$ and one with $<$. Clause 14 is deleted by applying rule R7s, because it is subsumed by clause 12. Clause 15 is deleted by applying rule R7f, since $\mathbb{R} \models \neg\,\exists(G \geq 0 \wedge G < 0)$. Clause 16 cannot be folded using clause 9, because *true* $\not\sqsubseteq G \geq 0$. Thus, the following new definition 17 is introduced and clause 16 is folded using clause 17:

17. $n_1(A,B,C) \leftarrow D \geq 0 \wedge \mathit{nth}(A,D,E) \wedge \mathit{nth}(B,D,F) \wedge \neg \mathit{nth}(C,D,E+F)$
18. $r([A|B],[C|D],[E|F]) \leftarrow n_1(B,D,F)$

The set $\{12,13,18\}$ of clauses defining predicate r has no existential variables. However, the new definition clause 17 has existential variables. Thus, the strategy proceeds by transforming clause 17. By unfolding and simplifying, we obtain:

19. $n_1([A|B],[C|D],[\,]) \leftarrow$
20. $n_1([A|B],[C|D],[E|F]) \leftarrow E \neq A+C$
21. $n_1([A|B],[C|D],[E|F]) \leftarrow G \geq 0 \wedge \mathit{nth}(B,G,H) \wedge \mathit{nth}(D,G,I) \wedge \neg \mathit{nth}(F,G,H+I)$

Now, clause 21 can be folded by using the previously introduced clause 17, thus obtaining:

22. $n_1([A|B],[C|D],[E|F]) \leftarrow n_1(B,D,F)$

The clauses 19, 20, and 22 defining n_1 in the current program do not contain any existential variable. The EVE strategy proceeds by processing the next definition in $D(p,\varphi)$, that is, clause 10. Starting from clause 10 we perform a derivation similar to the one performed starting from clause 9. In particular, during this derivation we introduce a new predicate n_2 and, by unfolding, clause deletion, and folding steps, we get the following two clauses:

23. $q \leftarrow n_2$ 24. $n_2 \leftarrow n_2$

Finally, the EVE strategy takes into consideration the last clause in $D(p,\varphi)$, that is, clause 11. Since this clause has no existential variables, it is simply added to the final program T. Thus, the EVE strategy terminates returning the propositional program $Q = \mathit{Defs}^*(p,T) = \{p \leftarrow \neg q,\ q \leftarrow n_2,\ n_2 \leftarrow n_2\}$.

The perfect model of Q is $M(Q) = \{p\}$. By the correctness of the EVE strategy, we have $M(P \cup \{9,10,11\}) \models p$, and by the correctness of the Lloyd-Topor transformation, we conclude $M(P) \models \varphi$.

Note that the EVE strategy does not depend on the specific theory of constraints and can be applied to all CLP programs whose theory of constraints admits quantifier elimination (LRA in our example). For some classes of CLP programs we can prove the termination of the EVE strategy, and thus the decidability of the theorem proving problem (for example, in [11] a similar strategy has been applied for deciding weak monadic second order logic [32]).

5. Temporal Logics and Verification of Infinite State Systems

In this section we show how to apply the unfold/fold proof method for verifying properties of infinite state systems specified by using formulas of the temporal logic CTL (*Computation Tree Logic* [5]).

A concurrent system can be modeled as a *state transition system* consisting of: (i) a (possibly infinite) set S of *states*, (ii) a set $I \subseteq S$ of *initial states*, and (iii) a *transition relation* $tr \subseteq S \times S$. We assume that, for every state $s \in S$, there exists at least one state $s' \in S$, called a *successor state* of s, such that $tr(s, s')$ holds. A *computation path* starting from a state s_1 is an *infinite* sequence of states $s_1\, s_2 \ldots$ such that, for all $i \geq 1$, $tr(s_i, s_{i+1})$ holds, that is, there is a transition from s_i to s_{i+1}.

The properties of (the computation paths of) a state transition system can be specified by using the CTL logic, whose formulas are built from a given set of *elementary properties*, by using: (i) the connectives: *not* and *and*, (ii) the quantifiers along a computation path: *g* ('for all states on the path'), *f* ('for some state on the path'), *x* ('in the successor state'), and *u* ('until'), and (iii) the quantifiers over computation paths: *a* ('for all paths') and *e* ('for some path'). For example, the formula $a(f(F))$ holds in a state s if on every computation path starting from s there exists a state s' where F holds. In what follows, for reasons of readability, we will use a compact notation and, for instance, we will write $af(F)$, instead of $a(f(F))$.

We consider the *Ticket Protocol* [2], which can be used for controlling the behaviour of two processes, say *A* and *B*, competing for the access to a shared resource. The protocol has the objective of guaranteeing both *mutual exclusion* when accessing the resource, and *starvation freedom*, which means that every request will eventually be served. The interaction between the two processes and the resource is realized by assigning tickets to processes that request access to the resource.

The *state* of process *A* is represented by a pair $\langle S_A, T_A \rangle$, where S_A, called the *control state*, is an element of the set $\{\mathtt{t}, \mathtt{w}, \mathtt{u}\}$ (where `t`, `w`, and `u` stand for *think*, *wait*, and *use*, respectively), and T_A is a non-negative number encoding the ticket assigned to process *A*. Analogously, the state of process *B* is encoded by $\langle S_B, T_B \rangle$. Thus, the state of the system resulting from the composition of the two processes *A* and *B* is represented by the term $\langle S_A, T_A,\, S_B, T_B,\, T, N \rangle$, where *T* is a non-negative number which is used for storing the value of the next ticket to be issued, and *N* is a non-negative number such that if $T_A \leq N$, then process *A* may access the shared resource (and similarly for T_B and process *B*).

A state is initial if and only if *T* and *N* have the same value and both processes are in the control state *think*. Thus, the (infinite) set of initial states can be encoded by introducing a predicate *initial*, such that $initial(X)$ holds iff X is an initial state, defined by the following CLP clause:

1. $initial(\langle \mathtt{t}, T_A,\, \mathtt{t}, T_B,\, T, N \rangle) \leftarrow T = N \wedge T \geq 0$

The transition relation is defined by a predicate $tr(X, Y)$ which holds iff Y is a successor state of X. The following set of clauses encode the transitions for process *A*, where, for brevity, we have used non-distinct variables in the head of clauses and we have omitted to write constraints of the form $T \geq 0$ and $N \geq 0$ to ensure that real variables range over non-negative numbers.

2. $tr(\langle \mathtt{t}, T_A,\, S_B, T_B,\, T, N \rangle,\ \langle \mathtt{w}, T,\, S_B, T_B,\, T_1, N \rangle) \leftarrow T_1 = T + 1$
3. $tr(\langle \mathtt{w}, T_A,\, S_B, T_B,\, T, N \rangle,\ \langle \mathtt{u}, T_A,\, S_B, T_B,\, T, N \rangle) \leftarrow T_A \leq N$
4. $tr(\langle \mathtt{u}, T_A,\, S_B, T_B,\, T, N \rangle,\ \langle \mathtt{t}, T_A,\, S_B, T_B,\, T, N_1 \rangle) \leftarrow N_1 = N + 1$

These clauses correspond to the arcs of the state transition diagram shown in Figure 1. The clauses encoding the transitions for process *B* are similar and we do not show them here. Note that, since the values of the numeric variables can increase in an unbounded way, the system has an infinite number of states.

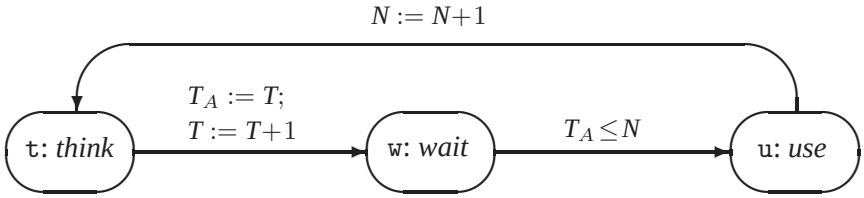

Figure 1. The Ticket Protocol: the state transition diagram for process *A*.

We define a predicate $sat(X, F)$ which holds if and only if the CTL formula F is true at state X [13] (see also [10, 18] for similar encodings). For instance, the following clauses define the predicate $sat(X, F)$ for the cases where the formula F is: (i) an elementary formula, (ii) a formula of the form $not(F)$, (iii) a formula of the form $and(F_1, F_2)$, (iv) a formula of the form $ef(F)$, and (v) a formula of the form $af(F)$.

5. $sat(X, F) \leftarrow elem(X, F)$
6. $sat(X, not(F)) \leftarrow \neg sat(X, F)$
7. $sat(X, and(F_1, F_2)) \leftarrow sat(X, F_1) \wedge sat(X, F_2)$
8. $sat(X, ef(F)) \leftarrow sat(X, F)$
9. $sat(X, ef(F)) \leftarrow tr(X, Y) \wedge sat(Y, ef(F))$
10. $sat(X, af(F)) \leftarrow sat(X, F)$
11. $sat(X, af(F)) \leftarrow trs(X, Ys) \wedge sat\_all(Ys, af(F))$
12. $sat\_all([\,], F) \leftarrow$
13. $sat\_all([X|Xs], F) \leftarrow sat(X, F) \wedge sat\_all(Xs, F)$

where $elem(X, F)$ holds iff F is an elementary property which is true at state X, and $trs(X, Ys)$ holds iff *Ys* is a list of all the successor states of X. For instance, we have that the elementary property $think_A$ is encoded by the clause: $elem(\langle \mathtt{t}, T_A, S_B, T_B, T, N\rangle, think_A) \leftarrow$. Similarly for the other elementary properties for processes *A* and *B*. The clauses for the predicate *trs* (when the control state of process *B* in the source state is *think*) are shown below:

14. $trs(\langle \mathtt{t}, T_A, \mathtt{t}, T_B, T, N\rangle, [\langle \mathtt{w}, T, \mathtt{t}, T_B, T_1, N\rangle, \langle \mathtt{t}, T_A, \mathtt{w}, T, T_1, N\rangle]) \leftarrow T_1 = T+1$
15. $trs(\langle \mathtt{w}, T_A, \mathtt{t}, T_B, T, N\rangle, [\langle \mathtt{u}, T_A, \mathtt{t}, T_B, T, N\rangle, \langle \mathtt{w}, T_A, \mathtt{w}, T, T_1, N\rangle]) \leftarrow T_A \leq N \wedge T_1 = T+1$
16. $trs(\langle \mathtt{w}, T_A, \mathtt{t}, T_B, T, N\rangle, [\langle \mathtt{w}, T_A, \mathtt{w}, T, T_1, N\rangle]) \leftarrow T_A > N \wedge T_1 = T+1$
17. $trs(\langle \mathtt{u}, T_A, \mathtt{t}, T_B, T, N\rangle, [\langle \mathtt{t}, T_A, \mathtt{t}, T_B, T, N_1\rangle, \langle \mathtt{u}, T_A, \mathtt{w}, T, T_1, N\rangle]) \leftarrow T_1 = T+1 \wedge N_1 = N+1$

For example, the first clause states that when both processes are in the *think* control state, there are two possible successor states where exactly one of them is in the *wait* control state. For lack of space we omit the other clauses for *trs*.

In order to verify that a CTL formula denoted by a ground term F holds for all initial states, we define a new predicate *prop*:

$prop \equiv_{def} \forall X(initial(X) \rightarrow sat(X, F))$

By using a variant of the Lloyd-Topor transformation [19] (see Section 4 for a similar transformation) and by the semantics of *not* defined by clause 7, we encode this definition by the following two clauses:

18. $prop \leftarrow \neg negprop$
19. $negprop \leftarrow initial(X) \wedge sat(X, not(F))$

Let P_F denote the constraint logic program consisting of the clauses defining the predicates *prop*, *negprop*, *initial*, *sat*, *sat_all*, *tr*, *trs*, and *elem*. The program P_F is locally stratified and, hence, it has a unique perfect model, denoted $M(P_F)$. One can show that our CLP encoding of the satisfiability of CTL formulas for state transition systems is correct [13], that is, for all states $s \in I$, the formula F holds at state s iff $prop \in M(P_F)$.

As already mentioned, the Ticket Protocol satisfies both: (i) the *mutual exclusion* property, forbidding the processes A and B to be at the same time in the control state *use*, and (ii) the *starvation freedom* property stating that, if a process, say *A*, has requested access to the resource and is waiting for accessing it (that is, the control state of process *A* is *wait*) then, whatever the system does, the

process will eventually gain access to the resource (that is, the control state of process A will be *use*). The latter property is expressed by the CTL formula *SF*: $ag(wait_A \rightarrow af(use_A))$, which is equivalent to $not(ef(and(wait_A, not(af(use_A)))))$. By using the fact that for every CTL formula F, the formula $not(not(F))$ is equivalent to F, the starvation freedom property is encoded by the following two clauses:

20. $prop \leftarrow \neg negprop$ 21. $negprop \leftarrow initial(X) \wedge sat(X, ef(and(wait_A, not(af(use_A)))))$

Now, let us consider a program, call it P_{SF}, which is like program P_F, except that clause 19 for *negprop* has been replaced by clause 21. If we transform program P_{SF} by applying the transformation rules R1–R5 and R7 according to a variant of the EVE strategy presented in Section 4, we derive a program containing a fact of the form: $prop \leftarrow$. Thus, by the correctness of the encoding of the satisfiability relation of CTL properties and by the correctness of the transformation rules, we conclude that the Ticket Protocol enjoys the starvation freedom property.

The verification of the starvation freedom property can be performed automatically by using the MAP system [1]. In [13], we have presented a transformation strategy, also based on the UDF strategy [31], that works by *specializing* program P_F with respect to any given CTL formula F and any given definitions of the predicates *initial*, *tr*, *trs*, and *elem*. In order to guarantee the termination of the transformation process, the strategy uses *generalization operators* (such as *widening* [7]) when introducing the required new definition clauses. Using that automatic strategy one can verify safety and liveness properties of several infinite-state concurrent systems, including mutual exclusion, parameterized cache coherence, and communication protocols (see [13] for details).

6. Further Developments and Conclusions

The techniques and the examples presented in this paper demonstrate that the unfold/fold proof method is very flexible and has a large variety of applications. Indeed, the method can be used for several logics (such as classical logic and temporal logic) and induction principles (such as fixpoint induction and structural induction). Moreover, it can be used for reasoning about programs written using different programming languages and different formalisms (such as concurrent process algebras, constraint logic programs, and transition systems).

In recent papers, the ability of the unfold/fold proof method to encode several induction principles has been exploited to develop techniques for reasoning about infinite structures [29, 36]. In [29] properties of programs on infinite lists are encoded using logic programs with the perfect model semantics, and those properties are proved by using transformation rules similar to the ones presented in this paper. In [36] properties of programs on infinite structures are encoded using *coinductive* logic programs (CoLP), whose semantics is defined by means of least and greatest models [37]. Then, those properties are proved by using unfolding and folding transformations that preserve the semantics of CoLP, thereby encoding a coinductive proof principle.

One key feature of the proof method presented in this paper is the use of constraint logic programming as a *metalanguage* for specifying both the programs and the logics to reason over programs. This feature makes the proof method suitable for a large number of different applications. For instance, recent papers (such as [8]) show that the unfold/fold proof method can be used to perform the analysis of imperative programs (see [16] for a survey of related techniques in the field of *software model checking*). The technique presented in [8] works as follows. Given an imperative program P and a property φ to be verified, one encodes both the interpreter of P and the property φ using a CLP program I (see [24]

for the definition of a CLP interpreter of an imperative language). Then, the property φ is proved by applying the unfold/fold proof method to program I. In particular, the transformation strategy used in this case consists in specializing I with respect to the given imperative program P.

We believe that the transformation-based methodology for theorem proving and program verification we have presented in this paper has a great potential. Some of the transformation rules and strategies presented here have been implemented in the MAP system [1] and we have obtained encouraging results. Current work is devoted to the mechanization of the verification and theorem proving techniques and to the improvement of the tools based on the transformation methodology so that they may perform well also for large scale programs.

Acknowledgments

We would like to thank the anonymous referees for their helpful comments. We also thank Dominik Ślęzak, Hung Son Nguyen, Marcin Szczuka for inviting us to contribute to this special issue dedicated to Professor Andrzej Skowron. We hope that Andrzej's enthusiasm and example may give to all his colleagues strength and joy for many years to come.

References

[1] The MAP transformation system. `http://www.iasi.cnr.it/~proietti/system.html`. Also available via WEB interface: `http://www.map.uniroma2.it/mapweb`.

[2] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

[3] K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.

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

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[6] B. Courcelle. Equivalences and transformations of regular systems – Applications to recursive program schemes and grammars. *Theoretical Computer Science*, 42:1–122, 1986.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. 5th ACM Symposium on Principles of Programming Languages, POPL'78*, 84–96. ACM Press, 1978.

[8] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs via Iterated Specialization. In *Proc. ACM SIGPLAN Workshop PEPM'13*. 43–52, ACM, New York, USA, 2013.

[9] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Comp. Sci.*, 166:101–146, 1996.

[10] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proc. ACM SIGPLAN Workshop VCL'01*, Technical Report DSSE-TR-2001-3, 85–96. University of Southampton, UK, 2001.

[11] F. Fioravanti, A. Pettorossi, and M. Proietti. Combining logic programs and monadic second order logics by program transformation. In M. Leuschel, ed., *Proc. LOPSTR'02*, Lecture Notes in Computer Science 2664, 160–181. Springer, 2003.

[12] F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, eds., *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, 292–340. Springer, 2004.

[13] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming. Special Issue 25th GULP*, 13(2):175–199, 2013.

[14] M. C. Hennessy. An Introduction to a Calculus of Communicating Systems. SRC Grant GR/A/75125, University of Edinburgh, Scotland, 1982.

[15] J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[16] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009.

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

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

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

[20] R. Loos and V. Weispfenning. Applying linear quantifier elimination. *The Comp. Jour.*, 36(5):450–462, 1993.

[21] J. McCarthy. Towards a mathematical science of computation. In C. M. Popplewell, ed., *Information Processing, Proc. IFIP'62*, 21–28, Amsterdam, North Holland, 1963.

[22] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[23] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In M. Leuschel, ed., *Proc. LOPSTR'02*, Lecture Notes in Computer Science 2664, Springer, 90–108, 2003.

[24] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In G. Levi, ed., *Proc. SAS'98*, LNCS 1503, 246–261. Springer, 1998.

[25] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19, 20:261–320, 1994.

[26] 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.

[27] A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, ed., *Proc. CL 2000*, Lecture Notes in Artificial Intelligence 1861, 613–628. Springer, 2000.

[28] A. Pettorossi, M. Proietti, and V. Senni. Proving properties of constraint logic programs by eliminating existential variables. In S. Etalle and M. Truszczyński, eds., *Proc. ICLP'06*, Lecture Notes in Computer Science 4079, 179–195. Springer, 2006.

[29] A. Pettorossi, M. Proietti, and V. Senni. Transformations of logic programs on infinite lists. *Theory and Practice of Logic Programming, Special Issue ICLP'10, Edinburgh, Scotland*, 10(4–6): 383–399, 2010.

[30] A. Pettorossi, M. Proietti, and V. Senni. Constraint-based correctness proofs for logic program transformations. *Formal Aspects of Computing*, 24:569–594, 2012.

[31] 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.

[32] M. O. Rabin. Decidable theories. In J. Barwise, ed., *Handbook of Mathematical Logic*, 595–629. North-Holland, 1977.

[33] A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *Proc. TACAS 2000*, Lecture Notes in Computer Science 1785, 172–187. Springer, 2000.

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

[35] H. Seki. On inductive and coinductive proofs via unfold/fold transformations. In D. De Schreye, ed., *Proc. LOPSTR'09*, Lecture Notes in Computer Science 6037, 82–96. Springer, 2010.

[36] H. Seki. Proving properties of co-logic programs with negation by program transformations. In E. Albert, ed., *Proc. LOPSTR'12*, Lecture Notes in Computer Science 7844, 213–227. Springer, 2013.

[37] L. Simon, A. Mallya, A. Bansal, and G. Gupta. Coinductive logic programming. In S. Etalle and M. Truszczyński, eds., *Proc. ICLP'06*, Lecture Notes in Computer Science 4079, 330–345. Springer, 2006.

[38] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, ed., *Proc. ICLP'84*, Uppsala University, Uppsala, Sweden, 127–138, 1984.