Combining Logic Programs and Monadic Second Order Logics by Program Transformation

Fabio Fioravanti[1], Alberto Pettorossi[2], Maurizio Proietti[1]

(1) IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
(2) DISP, University of Roma Tor Vergata, I-00133 Roma, Italy
{fioravanti,adp,proietti}@iasi.rm.cnr.it

Abstract We present a program synthesis method based on unfold/fold transformation rules which can be used for deriving terminating definite logic programs from formulas of the Weak Monadic Second Order theory of one successor (WS1S). This synthesis method can also be used as a proof method which is a decision procedure for closed formulas of WS1S. We apply our synthesis method for translating CLP(WS1S) programs into logic programs and we use it also as a proof method for verifying safety properties of infinite state systems.

1 Introduction

The Weak Monadic Second Order theories of k successors (WSkS) are theories of the second order predicate logic which express properties of finite sets of finite strings over a k-symbol alphabet (see [25] for a survey). Their importance relies on the fact that they are among the most expressive theories of predicate logic which are decidable. These decidability results were proved in the 1960's [4,23], but they were considered as purely theoretical results, due to the very high complexity of the automata-based decision procedures.

In recent years, however, it has been shown that some Monadic Second Order theories can, in fact, be decided by using ad-hoc, efficient techniques, such as BDD's and algorithms for finite state automata. In particular, the MONA system implements these techniques for the WS1S and WS2S theories [10].

The MONA system has been used for the verification of several non-trivial finite state systems [3,12]. However, the Monadic Second Order theories alone are not expressive enough to deal with properties of infinite state systems and, thus, for the verification of such systems alternative techniques have been used, such as those based on the embedding of the Monadic Second Order theories into more powerful logical frameworks (see, for instance, [2]).

In a previous paper of ours [7] we proposed a verification method for infinite state systems based on CLP(WSkS), which is a constraint logic programming language resulting from the embedding of WSkS into logic programs. In order to perform proofs of properties of infinite state systems in an automatic way according to the approach we have proposed, we need a system for constraint logic programming which uses a solver for WSkS formulas and, unfortunately, no such system is available yet.

In order to overcome this difficulty, in this paper we propose a method for translating CLP(WS1S) programs into logic programs. This translation is performed by a two step program synthesis method which produces terminating definite logic programs from WS1S formulas. Step 1 of our synthesis method consists in deriving a normal logic program from a WS1S formula, and it is based on a variant of the Lloyd-Topor transformation [15]. Step 2 consists in applying an unfold/fold transformation strategy to the normal logic program derived at the end of Step 1, thereby deriving a terminating definite logic program. Our synthesis method follows the general approach presented in [17,18]. We leave it for future research the translation into logic programs starting from general CLP(WSkS) programs.

The specific contributions of this paper are the following ones.

(1) We provide a synthesis strategy which is guaranteed to terminate for any given WS1S formula.

(2) We prove that, when we start from a closed WS1S formula φ, our synthesis strategy produces a program which is either (i) a unit clause of the form $f \leftarrow$, where f is a nullary predicate equivalent to the formula φ, or (ii) the empty program. Since in case (i) φ is true and in case (ii) φ is false, our strategy is also a decision procedure for WS1S formulas.

(3) We show through a non-trivial example, that our verification method based on CLP(WS1S) programs is useful for verifying properties of infinite state transition systems. In particular, we prove the safety property of a mutual exclusion protocol for a set of processes whose cardinality may change over time. Our verification method requires: (i) the encoding into WS1S formulas of both the transition relation and the elementary properties of the states of a transition system, and (ii) the encoding into a CLP(WS1S) program of the safety property under consideration. Here we perform our verification task by translating the CLP(WS1S) program into a definite logic program, thereby avoiding the use of a solver for WS1S formulas. The verification of the safety property has been performed by using a prototype tool built on top of the MAP transformation system [24].

2 The Weak Monadic Second Order Theory of One Successor

We will consider a *first order* presentation of the Weak Monadic Second Order theory of one successor (WS1S). This first order presentation consists in writing formulas of the form $n \in S$, where $\in$ is a first order predicate symbol (to be interpreted as membership of a natural number to a finite set of natural numbers), instead of formulas of the form $S(n)$, where S is a *predicate variable* (to be interpreted as ranging over finite sets of natural numbers).

We use a *typed* first order language, with the following two types: *nat*, denoting the set of natural numbers, and *set*, denoting the set of the finite sets of natural numbers (for a brief presentation of the typed first order logic the reader may look at [15]). The alphabet of WS1S consists of: (i) a set *Ivars* of *individual variables* $N, N_1, N_2, \ldots$ of type *nat*, (ii) a set *Svars* of *set variables* $S, S_1, S_2, \ldots$ of type *set*, (iii) the nullary function symbol 0 (*zero*) of type *nat*, and the unary function symbol s (*successor*) of type $nat \rightarrow nat$, and (iv) the binary predicate symbols $\leq$ of type $nat \times nat$, and $\in$ of type $nat \times set$. $Ivars \cup Svars$ is ranged over by $X, X_1, X_2, \ldots$ The syntax of WS1S is defined by the following grammar:

Individual terms: $n ::= 0 \mid N \mid s(n)$
Atomic formulas: $A ::= n_1 \leq n_2 \mid n \in S$
Formulas: $\varphi ::= A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \exists N\, \varphi \mid \exists S\, \varphi$

When writing formulas we feel free to use also the connectives $\vee$, $\rightarrow$, $\leftrightarrow$ and the universal quantifier $\forall$, as shorthands of the corresponding formulas with $\neg$, $\wedge$, and $\exists$. Given any two individual terms n_1 and n_2, we will write the formulas $n_1 = n_2$, $n_1 \neq n_2$, and $n_1 < n_2$ as shorthands of the corresponding formulas using $\leq$. Notice that, for reasons of simplicity, we have assumed that the symbol $\leq$ is primitive, although it is also possible to define it in terms of $\in$ [25].

An example of a WS1S formula is the following formula μ, with free variables N and S, which expresses that N is the maximum number in a finite set S:

$$\mu \;:\; N \in S \wedge \neg\exists N_1 (N_1 \in S \wedge \neg N_1 \leq N)$$

The semantics of WS1S formulas is defined by considering the following *typed interpretation* $\mathcal{N}$:

(i) the domain of the type *nat* is the set *Nat* of the natural numbers and the domain of the type *set* is the set $P_{fin}(Nat)$ of all finite subsets of *Nat*;

(ii) the constant symbol 0 is interpreted as the natural number 0 and the function symbol s is interpreted as the successor function from *Nat* to *Nat*;

(iii) the predicate symbol $\leq$ is interpreted as the less-or-equal relation on natural numbers, and the predicate symbol $\in$ is interpreted as the membership of a natural number to a finite set of natural numbers.

The notion of a *variable assignment* σ *over a typed interpretation* is analogous to the untyped case, except that σ assigns to a variable an element of the domain of the type of the variable. The definition of the *satisfaction relation* $I \models_\sigma \varphi$, where I is a typed interpretation and σ is a variable assignment is also analogous to the untyped case, with the only difference that when we interpret an existentially quantified formula we assume that the quantified variable ranges over the domain of its type. We say that a formula φ is *true* in an interpretation I, written as $I \models \varphi$, iff $I \models_\sigma \varphi$ for all variable assignments σ. The problem of checking whether or not a WS1S formula is true in the interpretation $\mathcal{N}$ is decidable [4].

3 Translating WS1S Formulas into Normal Logic Programs

In this section we illustrate Step 1 of our method for synthesizing definite programs from WS1S formulas. In this step, starting from a WS1S formula, we derive a *stratified* normal logic program [1] (simply called *stratified programs*) by applying a variant of the Lloyd-Topor transformation, called *typed Lloyd-Topor transformation*. Given a stratified program P, we denote by $M(P)$ its *perfect model* (which is equal to its *least Herbrand model* if P is a definite program) [1].

Before presenting the typed Lloyd-Topor transformation, we need to introduce a definite program, called *NatSet*, which axiomatizes: (i) the natural numbers, (ii) the finite sets of natural numbers, (iii) the ordering on natural numbers ($\leq$), and (iv) the membership of a natural number to a finite set of natural numbers ($\in$). We represent: (i) a natural number $k\,(\geq 0)$ as a ground term of the form $s^k(0)$, and (ii) a set of natural numbers as a finite, ground list $[b_0, b_1, \ldots, b_m]$ where, for $i = 0, \ldots, m$, we have that b_i is either `y` or `n`. A number k belongs to the set represented by $[b_0, b_1, \ldots, b_m]$ iff $b_k = \mathtt{y}$. Thus, the finite, ground lists $[b_0, b_1, \ldots, b_m]$ and $[b_0, b_1, \ldots, b_m, \mathtt{n}, \ldots, \mathtt{n}]$ represent the same set. In particular, the empty set is represented by any list of the form $[\mathtt{n}, \ldots, \mathtt{n}]$. The program *NatSet* consists of the following clauses (we adopt infix notation for $\leq$ and $\in$):

$$\begin{array}{ll}
nat(0) \leftarrow & 0 \leq N \leftarrow \\
nat(s(N)) \leftarrow nat(N) & s(N_1) \leq s(N_2) \leftarrow N_1 \leq N_2 \\
set([\,]) \leftarrow & 0 \in [\mathtt{y}|S] \leftarrow \\
set([\mathtt{y}|S]) \leftarrow set(S) & s(N) \in [B|S] \leftarrow N \in S \\
set([\mathtt{n}|S]) \leftarrow set(S) &
\end{array}$$

Atoms of the form $nat(N)$ and $set(S)$ are called *type atoms*. Now we will establish a correspondence between the set of WS1S formulas which are true in $\mathcal{N}$ and the set of the so-called *explicitly typed* WS1S formulas which are true in the least Herbrand model $M(NatSet)$ (see Theorem 1 below).

Given a WS1S formula φ, the *explicitly typed* WS1S formula corresponding to φ is the formula φ_τ constructed as follows. We first replace the subformulas of the form $\exists N\, \psi$ by $\exists N\,(nat(N) \wedge \psi)$ and the subformulas of the form $\exists S\, \psi$ by $\exists S\,(set(S) \wedge \psi)$, thereby getting a new formula φ_η where every bound (individual or set) variable occurs in a type atom. Then, we get:

$$\varphi_\tau \;:\; nat(N_1) \wedge \ldots \wedge nat(N_h) \wedge set(S_1) \wedge \ldots \wedge set(S_k) \wedge \varphi_\eta$$

where $N_1, \ldots, N_h, S_1, \ldots, S_k$ are the variables which occur free in φ.

For instance, let us consider again the formula μ which expresses that N is the maximum number in a set S. The explicitly typed formula corresponding to μ is the following formula:

$$\mu_\tau \;:\; nat(N) \wedge set(S) \wedge N \in S \wedge \neg\exists N_1 (nat(N_1) \wedge N_1 \in S \wedge \neg N_1 \leq N)$$

For reasons of simplicity, in the following Theorem 1 we identify: (i) a natural number $k\,(\geq 0)$ in *Nat* with the ground term $s^k(0)$ representing that number, and (ii) a finite set of natural numbers in $P_{fin}(Nat)$ with any finite, ground list representing that set. By using these identifications, we can view any variable assignment over the typed interpretation $\mathcal{N}$ also as a variable assignment over the untyped interpretation $M(NatSet)$ (but not vice versa).

Theorem 1. Let φ be a WS1S formula and let φ_τ be the explicitly typed formula corresponding to φ. For every variable assignment σ over $\mathcal{N}$,

$$\mathcal{N} \models_\sigma \varphi \;\text{ iff }\; M(NatSet) \models_\sigma \varphi_\tau$$

Proof. The proof proceeds by induction on the structure of the formula φ.
(i) Suppose that φ is of the form $n_1 \leq n_2$. By the definition of the satisfaction relation, $\mathcal{N} \models_\sigma n_1 \leq n_2$ iff the natural number $\sigma(n_1)$ is less or equal than the natural number $\sigma(n_2)$. By the definition of least Herbrand model and by using the clauses in *NatSet* which define $\leq$, $\sigma(n_1)$ is less or equal than $\sigma(n_2)$ iff $M(\mathit{NatSet}) \models \sigma(n_1) \leq \sigma(n_2)$ (here we identify every natural number n with the ground term $s^n(0)$). It can be shown that $M(\mathit{NatSet}) \models \mathit{nat}(\sigma(n_1))$ and $M(\mathit{NatSet}) \models \mathit{nat}(\sigma(n_2))$. Thus, $M(\mathit{NatSet}) \models \sigma(n_1) \leq \sigma(n_2)$ iff $M(\mathit{NatSet}) \models_\sigma \mathit{nat}(n_1) \wedge \mathit{nat}(n_2) \wedge n_1 \leq n_2$. Now, the term n_1 is either of the form $s^{m1}(0)$ or of the form $s^{m1}(N_1)$, where $m1$ is a natural number. Similarly, the term n_2 is either of the form $s^{m2}(0)$ or of the form $s^{m2}(N)$, where $m2$ is a natural number. We consider the case where n_1 is $s^{m1}(N_1)$ and n_2 is $s^{m2}(N_2)$. The other cases are similar and we omit them. It can be shown that, for all natural numbers m, $M(\mathit{NatSet}) \models_\sigma \mathit{nat}(s^m(N))$ iff $M(\mathit{NatSet}) \models_\sigma \mathit{nat}(N)$. Thus, $M(\mathit{NatSet}) \models_\sigma \mathit{nat}(s^{m1}(N_1)) \wedge \mathit{nat}(s^{m2}(N_2)) \wedge s^{m1}(N_1) \leq s^{m2}(N_2)$ iff $M(\mathit{NatSet}) \models_\sigma \mathit{nat}(N_1) \wedge \mathit{nat}(N_2) \wedge s^{m1}(N_1) \leq s^{m2}(N_2)$, that is, $M(\mathit{NatSet}) \models_\sigma (n_1 \leq n_2)_\tau$.
(ii) The case where φ is of the form $n \in S$ is similar to Case (i).
(iii) Suppose that φ is of the form $\neg\psi$. By the definition of the satisfaction relation and the induction hypothesis, $\mathcal{N} \models_\sigma \neg\psi$ iff $M(\mathit{NatSet}) \models_\sigma \neg(\psi_\tau)$. Since ψ_τ is of the form $a_1(X_1) \wedge \ldots \wedge a_k(X_k) \wedge \psi_\eta$, where $X_1, \ldots, X_k$ are the free variables in ψ and $a_1(X_1), \ldots, a_k(X_k)$ are type atoms, by logical equivalence, we get: $M(\mathit{NatSet}) \models_\sigma \neg(\psi_\tau)$ iff $M(\mathit{NatSet}) \models_\sigma (a_1(X_1) \wedge \ldots \wedge a_k(X_k) \wedge \neg(\psi_\eta)) \vee \neg(a_1(X_1) \wedge \ldots \wedge a_k(X_k))$. Finally, since for all variable assignments σ, $M(\mathit{NatSet}) \models_\sigma a_1(X_1) \wedge \ldots \wedge a_k(X_k)$, we have that $M(\mathit{NatSet}) \models_\sigma \neg(\psi_\tau)$ iff $M(\mathit{NatSet}) \models_\sigma (a_1(X_1) \wedge \ldots \wedge a_k(X_k) \wedge \neg(\psi_\eta))$, that is, $M(\mathit{NatSet}) \models_\sigma (\neg\psi)_\tau$ (to see this, note that $\neg(\psi_\eta)$ is equal to $(\neg\psi)_\eta$).
(iv) The case where φ is of the form $\psi_1 \wedge \psi_2$ is similar to Case (iii).
(v) Suppose that φ is of the form $\exists N_1\, \psi$. By the definition of the satisfaction relation and by the induction hypothesis, $\mathcal{N} \models_\sigma \exists N_1\, \psi$ iff there exists n_1 in *Nat* such that $M(\mathit{NatSet}) \models_{\sigma[N_1 \mapsto n_1]} \psi_\tau$. Since ψ_τ is of the form $\mathit{nat}(N_1) \wedge \ldots \wedge \mathit{nat}(N_h) \wedge \mathit{set}(S_1) \wedge \ldots \wedge \mathit{set}(S_k) \wedge \psi_\eta$, where $N_1, \ldots, N_h, S_1, \ldots, S_k$ are the free variables in ψ, we have that:
there exists n_1 in *Nat* such that $M(\mathit{NatSet}) \models_{\sigma[N_1 \mapsto n_1]} \psi_\tau$
iff $M(\mathit{NatSet}) \models_\sigma \exists N_1\, (\mathit{nat}(N_1) \wedge \ldots \wedge \mathit{nat}(N_h) \wedge \mathit{set}(S_1) \wedge \ldots \wedge \mathit{set}(S_k) \wedge \psi_\eta)$
iff (by logical equivalence) $M(\mathit{NatSet}) \models_\sigma \mathit{nat}(N_2) \wedge \ldots \wedge \mathit{nat}(N_h) \wedge \mathit{set}(S_1) \wedge \ldots \wedge \mathit{set}(S_k) \wedge (\exists N_1\, \mathit{nat}(N_1) \wedge \psi_\eta)$
iff (by definition of explicitly typed formula) $M(\mathit{NatSet}) \models_\sigma (\exists N_1\, \psi)_\tau$.
(vi) The case where φ is of the form $\exists S\, \psi$ is similar to Case (v). □

As a straightforward consequence of Theorem 1, we have the following result.

Corollary 1. For every closed WS1S formula φ, $\mathcal{N} \models \varphi$ iff $M(\mathit{NatSet}) \models \varphi_\tau$.

Notice that the introduction of type atoms is indeed necessary, because there are WS1S formulas φ such that $\mathcal{N} \models \varphi$ and $M(\mathit{NatSet}) \not\models \varphi$. For instance, $\mathcal{N} \models \forall N_1 \exists N_2\, N_1 \leq N_2$ and $M(\mathit{NatSet}) \not\models \forall N_1 \exists N_2\, N_1 \leq N_2$. Indeed, for a variable assignment σ over $M(\mathit{NatSet})$ which assigns $[\,]$ to N_1, we have $M(\mathit{NatSet}) \not\models_\sigma$

$\exists N_2\, N_1 \leq N_2$. (Notice that σ is not a variable assignment over $\mathcal{N}$ because $[\,]$ is not a natural number.)

Now we present a variant of the method proposed by Lloyd and Topor [15], called *typed Lloyd-Topor transformation*, which we use for deriving a stratified program from a given WS1S formula φ. We need to consider a class of formulas of the form: $A \leftarrow \beta$, called *statements*, where A is an atom, called the *head* of the statement, and β is a formula of the first order predicate calculus, called the *body* of the statement. In what follows we write $C[\gamma]$ to denote a formula where the subformula γ occurs as an *outermost conjunct*, that is, $C[\gamma] = \psi_1 \wedge \gamma \wedge \psi_2$ for some subformulas ψ_1 and ψ_2.

The Typed Lloyd-Topor Transformation.
We are given in input a set of statements, where: (i) we assume without loss of generality, that the only connectives and quantifiers occurring in the body of the statements are $\neg, \wedge$, and $\exists$, and (ii) $X, X_1, X_2, \ldots$ denote either individual or set variables.
We perform the following transformation (A) and then the transformation (B):

(A) We repeatedly apply the following rules A.1–A.4 until a set of clauses is generated:

(A.1) $A \leftarrow C[\neg\neg\gamma]$ is replaced by $A \leftarrow C[\gamma]$.

(A.2) $A \leftarrow C[\neg(\gamma \wedge \delta)]$ is replaced by $A \leftarrow C[\neg \mathit{newp}(X_1, \ldots, X_k)]$
$\mathit{newp}(X_1, \ldots, X_k) \leftarrow \gamma \wedge \delta$

where *newp* is a new predicate and $X_1, \ldots, X_k$ are the variables which occur free in $\gamma \wedge \delta$.

(A.3) $A \leftarrow C[\neg\exists X\, \gamma]$ is replaced by $A \leftarrow C[\neg \mathit{newp}(X_1, \ldots, X_k)]$
$\mathit{newp}(X_1, \ldots, X_k) \leftarrow \gamma$

where *newp* is a new predicate and $X_1, \ldots, X_k$ are the variables which occur free in $\exists X\, \gamma$.

(A.4) $A \leftarrow C[\exists X\, \gamma]$ is replaced by $A \leftarrow C[\gamma\{X/X_1\}]$

where X_1 is a new variable.

(B) Every clause $A \leftarrow G$ is replaced by $A \leftarrow G_\tau$.

Given a WS1S formula φ with free variables $X_1, \ldots, X_n$, we denote by $\mathit{Cls}(f, \varphi_\tau)$ the set of clauses derived by applying the typed Lloyd-Topor transformation starting from the singleton $\{f(X_1, \ldots, X_n) \leftarrow \varphi\}$, where f is a new n-ary predicate symbol. By construction, $\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)$ is a stratified program. We have the following theorem.

Theorem 2. Let φ be a WS1S formula with free variables $X_1, \ldots, X_n$ and let φ_τ be the explicitly typed formula corresponding to φ. For all ground terms $t_1, \ldots, t_n$, we have that:

$$M(\mathit{NatSet}) \models \varphi_\tau\{X_1/t_1,\ldots,X_n/t_n\} \text{ iff}$$
$$M(\mathit{NatSet} \cup \mathit{Cls}(f,\varphi_\tau)) \models f(t_1,\ldots,t_n)$$

Proof. It is similar to the proofs presented in [15,17] and we omit it.

From Theorems 1 and 2 we have the following corollaries.

Corollary 2. For every WS1S formula φ with free variables $X_1,\ldots,X_n$, and for every variable assignment σ over the typed interpretation $\mathcal{N}$,

$$\mathcal{N} \models_\sigma \varphi \text{ iff } M(\mathit{NatSet} \cup \mathit{Cls}(f,\varphi_\tau)) \models f(\sigma(X_1),\ldots,\sigma(X_n))$$

Corollary 3. For every closed WS1S formula φ,

$$\mathcal{N} \models \varphi \text{ iff } M(\mathit{NatSet} \cup \mathit{Cls}(f,\varphi_\tau)) \models f$$

Let us consider again the formula μ we have considered above. By applying the typed Lloyd-Topor transformation starting from the singleton $\{\mathit{max}(S,N) \leftarrow \mu\}$ we get the following set of clauses $\mathit{Cls}(\mathit{max},\mu_\tau)$:

$\mathit{max}(S,N) \leftarrow \mathit{nat}(N) \wedge \mathit{set}(S) \wedge N \in S \wedge \neg \mathit{newp}(S,N)$
$\mathit{newp}(S,N) \leftarrow \mathit{nat}(N) \wedge \mathit{nat}(N_1) \wedge \mathit{set}(S) \wedge N_1 \in S \wedge \neg N_1 \leq N$

Unfortunately, the stratified program $\mathit{NatSet} \cup \mathit{Cls}(f,\varphi_\tau)$ derived from the singleton $\{f(X_1,\ldots,X_n) \leftarrow \varphi\}$ is not always satisfactory from a computational point of view because it may not terminate when evaluating the query $f(X_1,\ldots,X_n)$ by using SLDNF resolution. (Actually, the above program $\mathit{Cls}(\mathit{max},\mu_\tau)$ which computes the maximum number of a set, terminates for all ground queries, but in Section 5 we will give an example where the program derived at the end of the typed Lloyd-Topor transformation does not terminate.) Similar termination problems may occur by using *tabled resolution* [5], instead of SLDNF resolution.

To overcome this problem, we apply to the program $\mathit{NatSet} \cup \mathit{Cls}(f,\varphi_\tau)$ the unfold/fold transformation strategy which we will describe in Section 5. In particular, by applying this strategy we derive definite programs which terminate for all ground queries by using LD resolution (that is, SLD resolution with the leftmost selection rule).

4 The Transformation Rules

In this section we describe the transformation rules which we use for transforming stratified programs. These rules are a subset of those presented in [17,18], and they are those required for the unfold/fold transformation strategy presented in Section 5.

For presenting our rules we need the following notions. A variable in the body of a clause C is said to be *existential* iff it does not occur in the head of C. The *definition* of a predicate p in a program P, denoted by $\mathit{Def}(p,P)$, is the set of the clauses of P whose head predicate is p. The *extended definition* of a predicate p in a program P, denoted by $\mathit{Def}^*(p,P)$, is the union of the definition of p and the definitions of all predicates in P on which p depends. (See [1]for the definition of the *depends on* relation.) A program is *propositional* iff every predicate occurring in the program is nullary. Obviously, if P is a propositional program then, for every predicate p, $M(P) \models p$ is decidable.

A *transformation sequence* is a sequence $P_0, \ldots, P_n$ of programs, where for $0 \leq k \leq n-1$, program P_{k+1} is derived from program P_k by the application of one of the transformation rules R1–R4 listed below. For $0 \leq k \leq n$, we consider the set Defs_k of the clauses introduced by the following rule R1 during the construction of the transformation sequence $P_0, \ldots, P_k$.

When considering clauses of programs, we will feel free to apply the following transformations which preserve the perfect model semantics:

(1) renaming of variables,

(2) rearrangement of the order of the literals in the body of a clause, and

(3) replacement of a conjunction of literals the form $L \wedge L$ in the body of a clause by the literal L.

Rule R1. Definition. We get the new program P_{k+1} by adding to program P_k a clause of the form $\mathit{newp}(X_1, \ldots, X_r) \leftarrow L_1 \wedge \ldots \wedge L_m$, where: (i) the predicate *newp* is a predicate which does not occur in $P_0 \cup \mathit{Defs}_k$, and (ii) $X_1, \ldots, X_r$ are distinct (individual or set) variables occurring in $L_1 \wedge \ldots \wedge L_m$.

Rule R2. Unfolding. Let C be a renamed apart clause in P_k of the form: $H \leftarrow G_1 \wedge L \wedge G_2$, where L is either the atom A or the negated atom $\neg A$. Let $H_1 \leftarrow B_1, \ldots, H_m \leftarrow B_m$, with $m \geq 0$, be all clauses of program P_k whose head is unifiable with A and, for $j = 1, \ldots, m$, let ϑ_j the most general unifier of A and H_j. We consider the following two cases.

Case 1: L is A. By *unfolding* clause C *w.r.t.* A we derive the new program $P_{k+1} = (P_k - \{C\}) \cup \{(H \leftarrow G_1 \wedge B_1 \wedge G_2)\vartheta_1, \ldots, (H \leftarrow G_1 \wedge B_m \wedge G_2)\vartheta_m\}$. In particular, if $m=0$, that is, if we unfold C w.r.t. an atom which is not unifiable with the head of any clause in P_k, then we derive the program P_{k+1} by deleting clause C.

Case 2: L is $\neg A$. Assume that: (i) $A = H_1\vartheta_1 = \cdots = H_m\vartheta_m$, that is, for $j = 1, \ldots, m$, A is an instance of H_j, (ii) for $j = 1, \ldots, m$, $H_j \leftarrow B_j$ has no existential variables, and (iii) $Q_1 \vee \ldots \vee Q_r$, with $r \geq 0$, is the disjunctive normal form of $G_1 \wedge \neg(B_1\vartheta_1 \vee \ldots \vee B_m\vartheta_m) \wedge G_2$. By *unfolding* clause C *w.r.t.* $\neg A$ we derive the new program $P_{k+1} = (P_k - \{C\}) \cup \{C_1, \ldots, C_m\}$, where for $j = 1, \ldots, r$, C_j is the clause $H \leftarrow Q_j$.

In particular: (i) if $m = 0$, that is, A is not unifiable with the head of any clause in P_k, then we get the new program P_{k+1} by deleting $\neg A$ from the body of clause C, and (ii) if for some $j \in \{1, \ldots, m\}$, B_j is the empty conjunction, that is, A is an instance of the head of a unit clause in P_k, then we derive P_{k+1} by deleting clause C from P_k.

Rule R3. Folding. Let $C : H \leftarrow G_1 \wedge B\vartheta \wedge G_2$ be a renamed apart clause in P_k and $D : \mathit{Newp} \leftarrow B$ be a clause in Defs_k. Suppose that for every existential variable X of D, we have that $X\vartheta$ is a variable which occurs neither in $\{H, G_1, G_2\}$ nor in the term $Y\vartheta$, for any variable Y occurring in B and different from X. By *folding* clause C using clause D we derive the new program $P_{k+1} = (P_k - \{C\}) \cup \{H \leftarrow G_1 \wedge \mathit{Newp}\,\vartheta \wedge G_2\}$.

Rule R4. Propositional Simplification. Let p be a predicate such that $\mathit{Def}^*(p, P_k)$ is propositional. If $M(\mathit{Def}^*(p, P_k)) \models p$ then we derive $P_{k+1} = (P_k - \mathit{Def}(p, P_k)) \cup \{p \leftarrow\}$. If $M(\mathit{Def}^*(p, P_k)) \models \neg p$ then we derive $P_{k+1} = (P_k - \mathit{Def}(p, P_k))$.

Notice that we can check whether or not $M(P) \models p$ holds by applying program transformation techniques [17] and thus, Rule R4 may be viewed as a derived rule.

The transformation rules R1–R4 we have introduced above, are collectively called *unfold/fold transformation rules.* We have the following correctness result, similar to [17].

Theorem 3. [Correctness of the Unfold/Fold Transformation Rules] Let us assume that during the construction of a transformation sequence $P_0, \ldots, P_n$, each clause of Defs_n which is used for folding, is unfolded (before or after its use for folding) w.r.t. an atom whose predicate symbol occurs in P_0. Then,

$$M(P_0 \cup \mathit{Defs}_n) = M(P_n).$$

Notice that the statement obtained from Theorem 3 by replacing 'atom' by 'literal', does not hold [17].

5 The Unfold/Fold Synthesis Method

In this section we present our program synthesis method, called *unfold/fold synthesis method*, which derives a definite program from any given WS1S formula. We show that the synthesis method terminates for all given formulas and also the derived programs terminate according to the following notion of program termination: a program P *terminates for a query* Q iff every SLD-derivation of $P \cup \{\leftarrow Q\}$ via any computation rule is finite.

The following is an outline of our unfold/fold synthesis method.

The Unfold/Fold Synthesis Method.
Let φ be a WS1S formula with free variables $X_1, \ldots, X_n$ and let φ_τ be the explicitly typed formula corresponding to φ.

Step 1. We apply the *typed Lloyd-Topor transformation* and we derive a set $\mathit{Cls}(f, \varphi_\tau)$ of clauses such that: (i) f is a new n-ary predicate symbol, (ii) NatSet $\cup \mathit{Cls}(f, \varphi_\tau)$ is a stratified program, and (iii) for all ground terms $t_1, \ldots, t_n$,

(1) $M(\mathit{NatSet}) \models \varphi_\tau\{X_1/t_1, \ldots, X_n/t_n\}$ iff
$M(\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)) \models f(t_1, \ldots, t_n)$

Step 2. We apply the *unfold/fold transformation strategy* (see below) and from the program $\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)$ we derive a definite program $\mathit{TransfP}$ such that, for all ground terms $t_1, \ldots, t_n$,

(2.1) $M(\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)) \models f(t_1, \ldots, t_n)$ iff $M(\mathit{TransfP}) \models f(t_1, \ldots, t_n)$;
(2.2) *TransfP* terminates for the query $f(t_1, \ldots, t_n)$.

In order to present the unfold/fold transformation strategy which we use for realizing Step 2 of our synthesis method, we introduce the following notions of *regular natset-typed clauses* and *regular natset-typed definitions*.

We say that a literal is *linear* iff each variable occurs at most once in it. The syntax of regular natset-typed clauses is defined by the following grammar (recall that by N we denote individual variables, by S we denote set variables, and by $X, X_1, X_2, \ldots$ we denote either individual or set variables):

Head terms: $h ::= 0 \mid s(N) \mid [\,] \mid [\mathtt{y}|S] \mid [\mathtt{n}|S]$
Clauses: $C ::= p(h_1, \ldots, h_k) \leftarrow \mid p_1(h_1, \ldots, h_k) \leftarrow p_2(X_1, \ldots, X_m)$

where for every clause C, (i) both $hd(C)$ and $bd(C)$ are linear atoms, and (ii) $\{X_1, \ldots, X_m\} \subseteq \mathit{vars}(h_1, \ldots, h_k)$ (that is, C has no existential variables). A *regular natset-typed program* is a set of regular natset-typed clauses.

The reader may check that the program *NatSet* presented in Section 3 is a regular natset-typed program. The following properties are straightforward consequences of the definition of regular natset-typed program.

Lemma 1. Let P be a regular natset-typed program. Then:
(i) P terminates for every ground query $p(t_1, \ldots, t_n)$ with $n > 0$;
(ii) If p is a nullary predicate then $\mathit{Def}^*(p, P)$ is propositional.

The syntax of natset-typed definitions is given by the following grammar:

Individual terms: $n ::= 0 \mid N \mid s(n)$
Terms: $t ::= n \mid S$
Type atoms: $T ::= \mathit{nat}(N) \mid \mathit{set}(S)$
Literals: $L ::= p(t_1, \ldots, t_k) \mid \neg p(t_1, \ldots, t_k)$
Definitions: $D ::= p(X_1, \ldots, X_k) \leftarrow T_1 \wedge \ldots \wedge T_r \wedge L_1 \wedge \ldots \wedge L_m$

where for all definitions D, $\mathit{vars}(D) \subseteq \mathit{vars}(T_1 \wedge \ldots \wedge T_r)$.

A sequence $D_1, \ldots, D_s$ of natset-typed definitions is said to be a *hierarchy* iff for $i = 1, \ldots, s$ the predicate appearing in $hd(D_i)$ does not occur in $D_1, \ldots, D_{i-1}, bd(D_i)$. Notice that in a hierarchy of natset-typed definitions, any predicate occurs in the head of at most one clause.

One can show that given a WS1S formula φ the set $\mathit{Cls}(f, \varphi_\tau)$ of clauses derived by applying the typed Lloyd-Topor transformation is a hierarchy $D_1, \ldots, D_s$ of natset-typed definitions and the last clause D_s is the one defining f.

The Unfold/Fold Transformation Strategy.

Input: (i) A regular natset-typed program P where for each nullary predicate p, $Def^*(p, TransfP)$ is either the empty set or the singleton $\{p \leftarrow\}$, and (ii) a hierarchy $D_1, \ldots, D_s$ of natset-typed definitions such that no predicate occurring in P occurs also in the head of a clause in $D_1, \ldots, D_s$.
Output: A regular natset-typed program *TransfP* such that, for all ground terms $t_1, \ldots, t_n$,
(2.1) $M(P \cup \{D_1, \ldots, D_s\}) \models f(t_1, \ldots, t_n)$ iff $M(\mathit{TransfP}) \models f(t_1, \ldots, t_n)$;
(2.2) *TransfP* terminates for the query $f(t_1, \ldots, t_n)$.

```
$\mathit{TransfP} := P$; $\mathit{Defs} := \emptyset$;
FOR $i = 1, \ldots, s$ DO
$\mathit{Defs} := \mathit{Defs} \cup \{D_i\}$;   $\mathit{InDefs} := \{D_i\}$;
By the definition rule we derive the program $\mathit{TransfP} \cup \mathit{InDefs}$.
WHILE $\mathit{InDefs} \neq \emptyset$ DO
(1) Unfolding. From program $\mathit{TransfP} \cup \mathit{InDefs}$ we derive $\mathit{TransfP} \cup U$ by: (i) applying the unfolding rule w.r.t. each atom occurring positively in the body of a clause in $\mathit{InDefs}$, thereby deriving $\mathit{TransfP} \cup U_1$, then (ii) applying the unfolding rule w.r.t. each negative literal occurring in the body of a clause in $U_1$, thereby deriving $\mathit{TransfP} \cup U_2$, and, finally, (iii) applying the unfolding rule w.r.t. ground literals until we derive a program $\mathit{TransfP} \cup U$ such that no ground literal occurs in the body of a clause of $U$.
(2) Definition-Folding. From program $\mathit{TransfP} \cup U$ we derive $\mathit{TransfP} \cup F \cup \mathit{NewDefs}$ as follows. Initially, $\mathit{NewDefs}$ is the empty set. For each non-unit clause $C$: $H \leftarrow B$ in $U$,
(i) we apply the definition rule and we add to $\mathit{NewDefs}$ a clause of the form $\mathit{newp}(X_1, \ldots, X_k) \leftarrow B$, where $X_1, \ldots, X_k$ are the non-existential variables occurring in $B$, unless a variant clause already occurs in $\mathit{Defs}$, modulo the head predicate symbol and the order and multiplicity of the literals in the body, and
(ii) we replace $C$ by the clause derived by folding $C$ w.r.t. $B$. The folded clause is an element of $F$.
No transformation rule is applied to the unit clauses occurring in $U$ and, therefore, also these clauses are elements of $F$.
(3) $\mathit{TransfP} := \mathit{TransfP} \cup F$;   $\mathit{Defs} := \mathit{Defs} \cup \mathit{NewDefs}$;   $\mathit{InDefs} := \mathit{NewDefs}$
END WHILE;
Propositional Simplification. For each predicate $p$ such that $Def^*(p, \mathit{TransfP})$ is propositional, we apply the propositional simplification rule and
if $M(\mathit{TransfP}) \models p$
then $\mathit{TransfP} := (\mathit{TransfP} - Def(p, \mathit{TransfP})) \cup \{p \leftarrow\}$
else $\mathit{TransfP} := (\mathit{TransfP} - Def(p, \mathit{TransfP}))$
END FOR
```

The reader may verify that if we apply the unfold/fold transformation strategy starting from the program $NatSet$ together with the clauses $Cls(max, \mu_\tau)$ which we have derived above by applying the typed Lloyd-Topor transformation, we get the following final program:

$max([\mathtt{y}|S], 0) \leftarrow new1(S)$
$max([\mathtt{y}|S], s(N)) \leftarrow max(S, N)$
$max([\mathtt{n}|S], s(N)) \leftarrow max(S, N)$
$new1([\,]) \leftarrow$
$new1([\mathtt{n}|S]) \leftarrow new1(S)$

To understand the first clause, recall that the empty set is represented by any list of the form $[\mathtt{n}, \ldots, \mathtt{n}]$. A more detailed example of application of the unfold/fold transformation strategy will be given later.

In order to prove the correctness and the termination of our unfold/fold transformation strategy we need the following lemmas whose proofs are mutually dependent.

Lemma 2. During the application of the unfold/fold transformation strategy, *TransfP* is a regular natset-typed program.

Proof. Initially, *TransfP* is the regular natset-typed program P. Now we assume that *TransfP* is a regular natset-typed program and we show that after an execution of the body of the FOR statement, *TransfP* is a regular natset-typed program.

First we prove that after the execution of the WHILE statement, *TransfP* is a regular natset-typed program. In order to prove this, we show that every new clause E which is added to *TransfP* at Point (3) of the strategy is a regular natset-typed clause.

Clause E is derived from a clause D of *InDefs* by unfolding (according to the Unfolding phase) and by folding (according to the Definition-Folding phase). By Lemma 3, D is a natset-typed definition of the form $p(X_1, \ldots, X_k) \leftarrow T_1 \wedge \ldots \wedge T_r \wedge L_1 \wedge \ldots \wedge L_m$. By unfolding w.r.t. the type atoms $T_1, \ldots, T_r$ (according to Point (i) of the Unfolding phase) we get clauses of the form $p(h_1, \ldots, h_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L'_1 \wedge \ldots \wedge L'_m$, where: (a) $h_1, \ldots, h_k$ are head terms, (b) $p(h_1, \ldots, h_k)$ is a linear atom (because $X_1, \ldots, X_k$ are distinct variables), and (c) for $i = 1, \ldots, m$, no argument of L'_i is a variable. By the inductive hypothesis *TransfP* is a regular natset-typed program and, therefore, by unfolding w.r.t. the literals $L'_1, \ldots, L'_m$ (according to Points (ii) and (iii) of the Unfolding phase) we get clauses of the form $D' : p(h_1, \ldots, h_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L''_1 \wedge \ldots \wedge L''_{m1}$. Either D' is a unit clause or, by folding according to the Definition-Folding phase, it is replaced by $p(h_1, \ldots, h_k) \leftarrow newp(X_1, \ldots, X_m)$ where $X_1, \ldots, X_m$ are the distinct, non-existential variables occurring in $bd(D')$. Hence, E is either a unit clause of the form $p(h_1, \ldots, h_k) \leftarrow$ or a clause of the form $p(h_1, \ldots, h_k) \leftarrow newp(X_1, \ldots, X_m)$, where $\{X_1, \ldots, X_m\} \subseteq vars(h_1, \ldots, h_k)$. Thus, E is a regular natset-typed clause.

We conclude the proof by observing that if we apply the propositional simplification rule to a natset-typed program, then we derive a natset-typed program, because by this rule we can only delete clauses or add natset-typed clauses of the form $p \leftarrow$. Thus, after an execution of the body of the FOR statement, *TransfP* is a regular natset-typed program. $\Box$

Lemma 3. During the application of the unfold/fold transformation strategy, *InDefs* is a set of natset-typed definitions.

Proof. Let us consider the i-th execution of the body of the FOR statement. Initially, *InDefs* is the singleton set $\{D_i\}$ of natset-typed definitions. Now we assume that *InDefs* is a set of natset-typed definitions and we prove that, after an execution of the WHILE statement, *InDefs* is a set of natset-typed definitions. It is enough to show that every new clause E which is added to *InDefs* at Point (3) of the strategy, is a natset-typed definition. By the Folding phase of the strategy, E is a clause of the form $newp(X_1,\ldots,X_k) \leftarrow B$ where B is the body of a clause derived from a clause D of *InDefs* by unfolding. By the inductive hypothesis, D is a natset-typed definition of the form $p(X_1,\ldots,X_k) \leftarrow T_1 \wedge \ldots \wedge T_r \wedge L_1 \wedge \ldots \wedge L_m$. By unfolding w.r.t. the type atoms $T_1,\ldots,T_r$ (according to Point (i) of the Unfolding phase) we get clauses of the form D' : $p(h_1,\ldots,h_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L'_1 \wedge \ldots \wedge L'_m$, where $vars(D') \subseteq vars(T'_1 \wedge \ldots \wedge T'_{r1})$. Since, by Lemma 2, *TransfP* is a regular natset-typed program, by unfolding w.r.t. the literals $L'_1,\ldots,L'_m$ (according to Points (ii) and (iii) of the Unfolding phase) we get clauses of the form D'' : $p(h_1,\ldots,h_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L''_1 \wedge \ldots \wedge L''_{m1}$ where $vars(D'') \subseteq vars(T'_1 \wedge \ldots \wedge T'_{r1})$. Thus, E is a natset-typed definition of the form $newp(X_1,\ldots,X_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L''_1 \wedge \ldots \wedge L''_{m1}$ with $vars(E) \subseteq vars(T'_1 \wedge \ldots \wedge T'_{r1})$.

We conclude the proof by observing that the Propositional Simplification phase does not change *InDefs*, and thus, after the execution of the body of the FOR statement, *InDefs* is a set of natset-typed definitions. $\Box$

Theorem 4. Let P and $D_1,\ldots,D_s$ be the input program and the input hierarchy, respectively, of the unfold/fold transformation strategy and let *TransfP* be the output of the strategy. Then,

(1) *TransfP* is a natset-typed program;

(2) for every nullary predicate p, $Def^*(p, TransfP)$ is either $\emptyset$ or $\{p \leftarrow\}$;

(3) for all ground terms $t_1,\ldots,t_n$,

(3.1) $M(P \cup \{D_1,\ldots,D_s\}) \models f(t_1,\ldots,t_n)$ iff $M(TransfP) \models f(t_1,\ldots,t_n)$;

(3.2) *TransfP* terminates for the query $f(t_1,\ldots,t_n)$.

Proof. Point (1) is a straightforward consequence of Lemma 2.

For Point (2), let us notice that, by Lemma 2, at each point of the unfold/fold transformation strategy *TransfP* is a natset-typed program and therefore, by Lemma 1, for every nullary predicate p, $Def^*(p, TransfP)$ is propositional. Since the last step of the unfold/fold transformation strategy consists in applying to *TransfP* the propositional simplification rule for each predicate having a propositional extended definition, $Def^*(p, TransfP)$ is either $\emptyset$ or $\{p \leftarrow\}$.

Point (3.1) will be proved by using the correctness of the transformation rules w.r.t. the Perfect Model semantics (see Theorem 3). Let us first notice that the unfold/fold transformation strategy generates a transformation sequence (see Section 4), where: the initial program is P, the final program is the final value of *TransfP*, and the set of clauses introduced by the definition rule R1 is the final value of *Defs*.

To see that our strategy indeed generates a transformation sequence, let us observe the following facts (A) and (B):

(A) The addition of *InDefs* to *TransfP* at the beginning of each execution of the body of the FOR statement is an application of the definition rule. Indeed, for $i = 1, \ldots s$, *InDefs* $= \{D_i\}$ and, by the hypotheses on the input sequence $D_1, \ldots, D_s$, we have that the head predicate of D_i does not occur in the current value of $P \cup \mathit{Defs}$.

(B) When we unfold the clauses of U_1 w.r.t. negative literals, we have that:
(B.1) Condition (i) of Case (2) of the unfolding rule (see Section 4) is satisfied because:
(a) Every clause D of *InDefs* is a natset-typed definition (see Lemma 3) and, thus, for each variable X occurring in D there is a type atom of the form $a(X)$ in $bd(D)$. Since we unfold the clauses of *InDefs* w.r.t. all the atoms which occur positively in the bodies of the clauses in *InDefs*, and in particular, w.r.t. type atoms, every argument of a negative literal in the body of a clause of U_1 is of one of the following forms: 0, $s(n)$, $[\,]$, $[\mathtt{y}|S]$, $[\mathtt{n}|S]$.
(b) For each negative literal $\neg p(t_1, \ldots, t_k)$ in the body of a clause of U_1, the definition of p is a subset of the regular natset-typed program *TransfP* (see Lemma 2) and, hence, the head of a clause in *TransfP* is a linear atom of the form $p(h_1, \ldots, h_k)$, where $h_1, \ldots, h_k$ are head terms (see the definition of regular natset-typed clauses above).
From (a) and (b) it follows that if $p(t_1, \ldots, t_k)$ is unifiable with $p(h_1, \ldots, h_k)$ then $p(t_1, \ldots, t_k)$ is an instance of $p(h_1, \ldots, h_k)$.
(B.2) Condition (ii) of Case (2) of the unfolding rule is satisfied because *TransfP* is a regular natset-typed program (see Lemma 2) and, thus, no clause in *TransfP* has existential variables.

Now, the transformation sequence constructed by the unfold/fold transformation strategy satisfies the hypothesis of Theorem 3. Indeed, let us consider a clause D which is used for folding a clause C. Since C has been derived at the end of the Unfolding phase, no ground literal occurs in $bd(C)$ and, thus, there is at least one variable occurring in D. Hence, there is at least one type atom in $bd(D)$, because D is a natset-typed definition (see Lemma 3). Therefore, during an application of the unfold/fold transformation strategy (before or after the use of D for folding), D is unfolded w.r.t. a type atom (see Point (i) of the Unfolding phase). Thus, by Theorem 3, we have that $M(P \cup \mathit{Defs}) = M(\mathit{TransfP})$, where by *Defs* and *TransfP* we indicate the values of these variables at the end of the unfold/fold transformation strategy. Observe that $\mathit{Def}^*(f, P \cup \mathit{Defs}) = \mathit{Def}^*(f, P \cup \{D_1, \ldots, D_s\})$ and, therefore, $M(P \cup \{D_1, \ldots, D_s\}) \models f(t_1, \ldots, t_n)$ iff $M(P \cup \mathit{Defs}) \models f(t_1, \ldots, t_n)$ iff $M(\mathit{TransfP}) \models f(t_1, \ldots, t_n)$.

Finally, let us prove Point (3.2). We consider the following two cases:

($n = 0$) f is nullary and hence, by Point (2) of this theorem, $\mathit{Def}^*(f, \mathit{TransfP})$ is either $\emptyset$ or $\{f \leftarrow\}$. Thus, *TransfP* terminates for the query f.

($n > 0$) By Point (1) of this theorem, *TransfP* is a natset-typed program and thus, by Lemma 1, *TransfP* terminates for the ground query $f(t_1, \ldots, t_n)$. □

Theorem 5. The unfold/fold transformation strategy terminates.

Proof. We have to show that the WHILE statement in the body of the FOR statement terminates.

Each execution of the Unfolding phase terminates. Indeed, (a) the number of applications of the unfolding rule at Points (i) and (ii) is finite, because *InDefs* is a finite set of clauses and the body of each clause has a finite number of literals, and (b) at Point (iii) only a finite number of unfolding steps can be applied w.r.t. ground literals, because the program held by *TransfP* during the Unfolding phase terminates for every ground query. To see this latter fact, let us notice that, by Lemma 2, *TransfP* is a natset-typed program. Thus, by Lemma 1, *TransfP* terminates for any ground query $p(t_1, \ldots, t_n)$ with $n \geq 1$. For a ground query p, where p is a nullary predicate, *TransfP* terminates because $\mathit{Def}^*(p, \mathit{TransfP})$ is either the empty set or it is the singleton $\{p \leftarrow\}$. Indeed, this follows from our assumptions on the input program and from the execution of the Propositional Simplification phase after completion of the WHILE statement.

Each execution of the Definition-Folding phase terminates because a finite number of clauses are introduced by definition and a finite number of clauses are folded.

Thus, in order to show that the strategy terminates, it is enough to show that after a finite number of executions of the body of the WHILE statement, we get $\mathit{InDefs} = \emptyset$. Let Defs_j and InDefs_j be the values of *Defs* and *InDefs*, respectively, at the end of the j-th execution of the body of the WHILE statement. If the WHILE statement terminates after z executions of its body, then, for all $j > z$, we define Defs_j to be Defs_z and InDefs_j to be $\emptyset$. We have that, for any $j \geq 1$, $\mathit{InDefs}_j = \emptyset$ iff $\mathit{Defs}_{j-1} = \mathit{Defs}_j$. Since for all $j \geq 1$, $\mathit{Defs}_{j-1} \subseteq \mathit{Defs}_j$, the termination of the strategy will follow from the following property:

$$\text{there exists } K > 0 \text{ such that, for all } j \geq 1,\ |\mathit{Defs}_j| \leq K \qquad (*)$$

Let $\mathit{TransfP}_0$, Defs_0, and InDefs_0 ($\subseteq \mathit{Defs}_0$) be the values of *TransfP*, *Defs*, and *InDefs*, respectively, at the beginning of the execution of the WHILE statement. By Lemma 3, for all $j \geq 1$, Defs_j is a set of natset-typed definitions. Property (*) follows from the fact that, for all $D \in \mathit{Defs}_j$, the following holds:

(a) every predicate occurring in $\mathit{bd}(D)$ also occurs in $\mathit{TransfP}_0 \cup \mathit{InDefs}_0$;

(b) for every literal L occurring in $\mathit{bd}(D)$,

$\mathit{height}(L) \leq \max\{\mathit{height}(M) \mid M \text{ is a literal in the body of a clause in } \mathit{Defs}_0\}$

where the *height* of a literal is defined as the length of the maximal path from the root to a leaf of the literal considered as a tree;

(c) $|\mathit{vars}(D)| \leq \max\{\mathit{vars}(D') \mid D' \text{ is a clause in } \mathit{Defs}_0\}$;

(d) no two clauses in Defs_j can be made equal by one or more applications of the following transformations: renaming of variables, renaming of head predicates,

rearrangement of the order of the literals in the body, and deletion of duplicate literals.
Recall that $bd(D)$ is equal to $bd(E')$ where E' is derived by unfolding (according to the Unfolding phase of the strategy) a clause E in $TransfP_0 \cup InDefs_j$ and E belongs to $InDefs_j$.
Now Property (a) is a straightforward consequence of the definition of the unfolding rule.
Property (b) can be shown as follows. E is of the form $newp(X_1, \ldots, X_k) \leftarrow T_1 \wedge \ldots \wedge T_r \wedge L_1 \wedge \ldots \wedge L_m$. By unfolding w.r.t. the type atoms $T_1, \ldots, T_r$ (according to Point (i) of the Unfolding phase) we get clauses of the form $newp(h_1, \ldots, h_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L'_1 \wedge \ldots \wedge L'_m$, where $h_1, \ldots, h_k$ are head terms and, for all $i \in \{1, \ldots, m\}$, $height(L'_i) \leq height(L_i) + 1$. By Lemma 2, $TransfP_0$ is a regular natset-typed program and, therefore, by unfolding w.r.t. the literals $L'_1, \ldots, L'_m$ (according to Point (ii) of the Unfolding phase) we get clauses of the form $newp(h_1, \ldots, h_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L''_1 \wedge \ldots \wedge L''_{m1}$, where for all $i \in \{1, \ldots, m1\}$, there exists $i1 \in \{1, \ldots, m\}$, such that $height(L''_i) = height(L'_{i1}) - 1$. Thus, Property (b) follows from the fact that E' is derived by unfolding w.r.t. ground literals from a clause of the form $newp(h_1, \ldots, h_k) \leftarrow T'_1 \wedge \ldots \wedge T'_{r1} \wedge L''_1 \wedge \ldots \wedge L''_{m1}$ and every unfolding w.r.t. a ground literal does not increase the height of the other literals in a clause.
Property (c) follows from Lemma 2 and the fact that by unfolding a clause E using regular natset-typed clauses we get clauses E' where $vars(E') \subseteq vars(E)$. To see this, recall that in a regular natset-typed clause C every term has at most one variable and $vars(bd(C)) \subseteq vars(hd(C))$ and, thus, by unfolding, a variable is replaced by a term with at most one variable and no new variables are introduced.
Finally, Point (d) is a consequence of Point (i) of the Definition-Folding phase of the unfold/fold strategy. $\square$

6 Deciding WS1S via the Unfold/Fold Proof Method

In this section we show that if we start from a *closed* WS1S formula φ, our synthesis method can be used for checking whether or not $\mathcal{N} \models \varphi$ holds and, thus, our synthesis method works also as a proof method which is a decision procedure for closed WS1S formulas.

If φ is a *closed* WS1S formula then the predicate f introduced when constructing the set $Cls(f, \varphi_\tau)$, is a nullary predicate. Let $TransfP$ be the program derived by the unfold/fold transformation strategy starting from the program $NatSet \cup Cls(f, \varphi_\tau)$. As already known from Point (2) of Theorem 4, we have that $Def^*(f, TransfP)$ is either the empty set or the singleton $\{f \leftarrow\}$. Thus, we can decide whether or not $\mathcal{N} \models \varphi$ holds by checking whether or not $f \leftarrow$ belongs to $TransfP$. Since the unfold/fold transformation strategy always terminates, we have that our unfold/fold synthesis method is indeed a decision procedure for closed WS1S formulas. We summarize our proof method as follows.

The Unfold/Fold Proof Method.
Let φ be a closed WS1S formula.
Step 1. We apply the typed Lloyd-Topor transformation and we derive the set $\mathit{Cls}(f, \varphi_\tau)$ of clauses.
Step 2. We apply the unfold/fold transformation strategy and from the program $\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)$ we derive a definite program *TransfP*.
If the unit clause $f \leftarrow$ belongs to *TransfP* then $\mathcal{N} \models \varphi$ else $\mathcal{N} \models \neg\varphi$.

Now we present a simple example of application of our unfold/fold proof method.

Example 1. (*An application of the unfold/fold proof method.*) Let us consider the closed WS1S formula $\varphi : \forall X\, \exists Y\, X \leq Y$. By applying the typed Lloyd-Topor transformation starting from the statement $f \leftarrow \varphi$, we get the following set of clauses $\mathit{Cls}(f, \varphi_\tau)$:

1. $h(X) \leftarrow nat(X) \wedge nat(Y) \wedge X \leq Y$
2. $g \leftarrow nat(X) \wedge \neg h(X)$
3. $f \leftarrow \neg g$

Now we apply the unfold/fold transformation strategy to the program *NatSet* and the following hierarchy of natset-typed definitions: clause 1, clause 2, clause 3. Initially, the program *TransfP* is *NatSet*. The transformation strategy proceeds left-to-right over that hierarchy.
(1) *Defs* and *InDefs* are both set to {clause 1}.
(1.1) *Unfolding.* By unfolding, from clause 1 we get:

4. $h(0) \leftarrow$
5. $h(0) \leftarrow nat(Y)$
6. $h(s(X)) \leftarrow nat(X) \wedge nat(Y) \wedge X \leq Y$

(1.2) *Definition-Folding.* In order to fold the body of clause 5 we introduce the following new clause:

7. $new1 \leftarrow nat(Y)$

Clause 6 can be folded by using clause 1. By folding clauses 5 and 6 we get:

8. $h(0) \leftarrow new1$
9. $h(s(X)) \leftarrow h(X)$

(1.3) At this point $\mathit{TransfP} = \mathit{NatSet} \cup \{$clause 4, clause 8, clause 9$\}$, $\mathit{Defs} = \{$clause 1, clause 7$\}$, and $\mathit{InDefs} = \{$clause 7$\}$.
(1.4) By first unfolding clause 7 and then folding using clause 7 itself, we get:

10. $new1 \leftarrow$
11. $new1 \leftarrow new1$

No new clause is introduced (i.e., $\mathit{NewDefs} = \emptyset$). At this point $\mathit{TransfP} = \mathit{NatSet} \cup \{$clause 4, clause 8, clause 9, clause 10, clause 11$\}$, $\mathit{Defs} = \{$clause 3, clause 7$\}$, and $\mathit{InDefs} = \emptyset$. Thus, the WHILE statement terminates.
Since $\mathit{Def}^*(new1, \mathit{TransfP})$ is propositional and $M(\mathit{TransfP}) \models new1$, by the propositional simplification rule we have:

$\mathit{TransfP} = \mathit{NatSet} \cup \{$clause 4, clause 8, clause 9, clause 10$\}$.

(2) *Defs* is set to $\{$clause 1, clause 2, clause 7$\}$ and *InDefs* is set to $\{$clause 2$\}$.
(2.1) *Unfolding.* By unfolding, from clause 2 we get:

12. $g \leftarrow \mathit{nat}(X) \wedge \neg h(X)$

(Notice that, by unfolding, clause $g \leftarrow \neg h(0)$ is deleted.)
(2.2) *Definition-Folding.* Clause 12 can be folded by using clause 2 which occurs in *Defs*. Thus, no new clause is introduced (i.e., $\mathit{NewDefs} = \emptyset$) and by folding we get:

13. $g \leftarrow g$

(2.3) At this point $\mathit{TransfP} = \mathit{NatSet} \cup \{$clause 4, clause 8, clause 9, clause 10, clause 13$\}$, $\mathit{Defs} = \{$clause 1, clause 2, clause 7$\}$, and $\mathit{InDefs} = \emptyset$. Thus, the WHILE statement terminates.
Since $\mathit{Def}^*(g, \mathit{TransfP})$ is propositional and $M(\mathit{TransfP}) \models \neg g$, by the propositional simplification rule we delete clause 13 from *TransfP* and we have:

$\mathit{TransfP} = \mathit{NatSet} \cup \{$clause 4, clause 8, clause 9, clause 10$\}$.

(3) *Defs* is set to $\{$clause 1, clause 2, clause 3, clause 7$\}$ and *InDefs* is set to $\{$clause 3$\}$.
(3.1) *Unfolding.* By unfolding clause 3 we get:

14. $f \leftarrow$

(Recall that, there is no clause in *TransfP* with head g.)
(3.2) *Definition-Folding.* No transformation steps are performed on clause 14 because it is a unit clause.
(3.3) At this point $\mathit{TransfP} = \mathit{NatSet} \cup \{$clause 4, clause 8, clause 9, clause 10, clause 14$\}$, $\mathit{Defs} = \{$clause 1, clause 2, clause 3, clause 7$\}$, and $\mathit{InDefs} = \emptyset$.
The transformation strategy terminates and, since the final program *TransfP* includes the unit clause $f \leftarrow$, we have proved that $\mathcal{N} \models \forall X\, \exists Y\, X \leq Y$.

We would like to notice that neither SLDNF nor Tabled Resolution (as implemented in the XSB system [22]) are able to construct a refutation of $\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau) \cup \{\leftarrow f\}$ (and thus construct a proof of φ), where φ is the WS1S formula $\forall X\, \exists Y\, X \leq Y$. Indeed, from the goal $\leftarrow f$ we generate the goal $\leftarrow \neg g$, and neither SLDNF nor Tabled Resolution are able to infer that $\leftarrow \neg g$ succeeds by detecting that $\leftarrow g$ generates an infinite set of failed derivations. □

We would like to mention that some other transformations could be applied for enhancing our unfold/fold transformation strategy. In particular, during the strategy we may apply the subsumption rule to shorten the transformation process by deleting some useless clauses. For instance, in Example 1 we can delete clause 5 which is subsumed by clause 4, thereby avoiding the introduction of the new predicate *new*1. In some other cases we can drop unnecessary type atoms. For instance, in Example 1 in clause 1 the type atom $\mathit{nat}(X)$ can be dropped because it is implied by the atom $X \leq Y$. The program derived at the end of the execution of the WHILE statement of the unfold/fold transformation strategy are nondeterministic, in the sense that an atom with non-variable arguments may be unifiable with the head of several clauses. We can apply the technique for deriving deterministic program presented in [19] for deriving deterministic programs and thus, obtaining smaller programs.

When the unfold/fold transformation strategy is used for program synthesis, it is often the case that the above mentioned transformations also improve the efficiency of the derived programs.

Finally, we would like to notice that the unfold/fold transformation strategy can be applied starting from a program $P \cup \mathit{Cls}(f, \varphi_\tau)$ (instead of $\mathit{NatSet} \cup \mathit{Cls}(f, \varphi_\tau)$) where: (i) P is the output of a previous application of the strategy, and (ii) φ is a formula built like a WS1S formula, except that it uses predicates occurring in P (besides $\leq$ and $\in$). Thus, we can synthesize programs (or construct proofs) in a *compositional* way, by first synthesizing programs for subformulas. We will follow this compositional methodology in the example of the following Section 7.

7 An Application to the Verification of Infinite State Systems: the Dynamic Bakery Protocol

In this section we present an example of verification of a safety property of an infinite state system by considering CLP(WS1S) programs [11]. As already mentioned, by applying our unfold/fold synthesis method we will then translate CLP(WS1S) programs into logic programs.

The syntax of CLP(WS1S) programs is defined as follows. We consider a set of *user-defined* predicate symbols. A CLP(WS1S) clause is of the form $A \leftarrow \varphi \wedge G$, where A is an atom, φ is a formula of WS1S, G is a goal, and the predicates occurring in A or in G are all user-defined. A CLP(WS1S) *program* is a set of CLP(WS1S) clauses. We assume that CLP(WS1S) programs are stratified.

Given a CLP(WS1S) program P, we define the semantics of P to be its perfect model, denoted $M(P)$ (here we extend to CLP(WS1S) programs the definitions which are given for normal logic programs in [1]).

Our example concerns the Dynamic Bakery protocol, called *DBakery* for short, and we prove that it ensures mutual exclusion in a system of processes which share a common resource, even if the number of processes in the system changes during a protocol run in a dynamic way. The *DBakery* protocol is a variant of the N-process Bakery protocol [13].

In order to give the formal specifications of the *DBakery* protocol and its mutual exclusion property, we will use CLP(WS1S) as we now indicate. The transition relation between pairs of system states, the initial system state, and the system states which are *unsafe* (that is, the system states where more than one process uses the shared resource) are specified by WS1S formulas. However, in order to specify the mutual exclusion property we cannot use WS1S formulas only. Indeed, mutual exclusion is a reachability property which is undecidable in the case of infinite state systems. The approach we follow in this example is to specify reachability (and, thus, mutual exclusion) as a CLP(WS1S) program (see the program $P_{DBakery}$ below).

Let us first describe the *DBakery* protocol. We assume that every process is associated with a natural number, called a *counter*, and two distinct processes have distinct counters. At each instant in time, the system of processes is represented by a pair $\langle W, U\rangle$, called a *system state*, where W is the set of the counters of the processes *waiting* for the resource, and U is the set of the counters of the processes *using* the resource.

A system state $\langle W, U\rangle$ is *initial* iff $W \cup U$ is the empty set.

The transition relation from a system state $\langle W, U\rangle$ to a new system state $\langle W', U'\rangle$ is the union of the following three relations:

(T1: *creation of a process*)
if $W \cup U$ is empty *then* $\langle W', U'\rangle = \langle \{0\}, \emptyset\rangle$ *else* $\langle W', U'\rangle = \langle W \cup \{m+1\}, U\rangle$, where m is the maximum counter in $W \cup U$,

(T2: *use of the resource*)
if there exists a counter n in W which is the minimum counter in $W \cup U$ *then* $\langle W', U'\rangle = \langle W - \{n\}, U \cup \{n\}\rangle$,

(T3: *release of the resource*)
if there exists a counter n in U *then* $\langle W', U'\rangle = \langle W, U - \{n\}\rangle$.

The mutual exclusion property holds iff from the initial system state it is not possible to reach a system state $\langle W, U\rangle$ which is *unsafe*, that is, such that U is a set of at least two counters.

Let us now give the formal specification of the *DBakery* protocol and its mutual exclusion property. We first introduce the following WS1S formulas (between parentheses we indicate their meaning):

$empty(X) \equiv \neg\exists x\ x \in X$
(the set X is empty)

$max(X,m) \equiv m \in X \wedge \forall x\ (x \in X \rightarrow x \leq m)$
(m is the maximum in the set X)

$min(X,m) \equiv m \in X \wedge \forall x\ (x \in X \rightarrow m \leq x)$
(m is the minimum in the set X)

(Here and in what follows, for reasons of readability, we allow ourselves to use lower case letters for individual variables of WS1S formulas.)
A system state $\langle W, U\rangle$ is *initial* iff $\mathcal{N} \models init(\langle W, U\rangle)$, where:

$$init(\langle W, U\rangle) \equiv empty(W) \wedge empty(U)$$

The transition relation R between system states is defined as follows:

$\langle\langle W, U\rangle, \langle W', U'\rangle\rangle \in R$ iff
$\mathcal{N} \models cre(\langle W, U\rangle, \langle W', U'\rangle) \vee use(\langle W, U\rangle, \langle W', U'\rangle) \vee rel(\langle W, U\rangle, \langle W', U'\rangle)$

where the predicates *cre*, *use*, and *rel* define the transition relations T1, T2, and T3, respectively. We have that:

$$\begin{aligned}
\mathit{cre}(\langle W,U\rangle\,,\,\langle W',U'\rangle) \;\equiv\;& U'\!=\!U \,\wedge\, \exists Z\,(Z\!=\!W\cup U\wedge \\
& ((\mathit{empty}(Z)\wedge W'\!=\!\{0\})\vee \\
& (\neg \mathit{empty}(Z)\wedge \exists m\,(\mathit{max}(Z,m)\wedge W'\!=\!W\cup\{s(m)\}))))\\
\mathit{use}(\langle W,U\rangle\,,\,\langle W',U'\rangle) \;\equiv\;& \exists n\,(n\in W\wedge \exists Z\,(Z\!=\!W\cup U\wedge \mathit{min}(Z,n))\wedge \\
& W'\!=\!W-\{n\}\,\wedge\, U'\!=\!U\cup\{n\})\\
\mathit{rel}(\langle W,U\rangle\,,\,\langle W',U'\rangle) \;\equiv\;& W'\!=\!W \,\wedge\, \exists n\,(n\in U\wedge U'\!=\!U-\{n\})
\end{aligned}$$

where the subformulas involving the set union ($\cup$), set difference ($-$), and set equality ($=$) operators can be expressed as WS1S formulas.

Mutual exclusion holds in a system state $\langle W,U\rangle$ iff $\mathcal{N}\models\neg\mathit{unsafe}(\langle W,U\rangle)$, where $\mathit{unsafe}(\langle W,U\rangle)\;\equiv\;\exists n_1\,\exists n_2\,(n_1\in U\,\wedge\, n_2\in U\,\wedge\,\neg(n_1=n_2))$, i.e., a system state $\langle W,U\rangle$ is unsafe iff there exist at least two distinct counters in U.

Now we will specify the system states reached from a given initial system state by introducing the CLP(WS1S) program $P_{DBakery}$ consisting of the following clauses:

$\mathit{reach}(S)\leftarrow \mathit{init}(S)$
$\mathit{reach}(S1)\leftarrow \mathit{cre}(S,S1)\wedge \mathit{reach}(S)$
$\mathit{reach}(S1)\leftarrow \mathit{use}(S,S1)\wedge \mathit{reach}(S)$
$\mathit{reach}(S1)\leftarrow \mathit{rel}(S,S1)\wedge \mathit{reach}(S)$

where $\mathit{init}(S)$, $\mathit{cre}(S,S1)$, $\mathit{use}(S,S1)$, and $\mathit{rel}(S,S1)$ are the WS1S formulas listed above.

From $P_{DBakery}$ we derive a definite program $P'_{DBakery}$ by replacing the WS1S formulas occurring in $P_{DBakery}$ by the corresponding atoms $\mathit{init}(S)$, $\mathit{cre}(S,S1)$, $\mathit{use}(S,S1)$, and $\mathit{rel}(S,S1)$, and by adding to the program the clauses (not listed here) defining these atoms, which are derived from the corresponding WS1S formulas listed above, by applying the unfold/fold synthesis method (see Section 5). Let us call these clauses *Init*, *Cre*, *Use*, and *Rel*, respectively.

In order to verify that the *DBakery* protocol ensures mutual exclusion for every system of processes whose number dynamically changes over time, we have to prove that for every ground term s denoting a finite set of counters, $\mathit{ur}(s)\notin M(P'_{DBakery}\cup\{\text{clause }1\})$, where clause 1 is the following clause which we introduce by the definition rule:

1. $\mathit{ur}(S)\leftarrow \mathit{unsafe}(S)\wedge \mathit{reach}(S)$

and $\mathit{unsafe}(S)$ is defined by a set, called *Unsafe*, of clauses which are derived from the corresponding WS1S formula by using the unfold/fold synthesis method.

In order to verify the mutual exclusion property for the *DBakery* protocol it is enough to show that $P'_{DBakery}\cup\{\text{clause }1\}$ can be transformed into a new definite program without clauses for $\mathit{ur}(S)$. This transformation can be done, as we now illustrate, by a straightforward adaptation of the proof technique presented for Constraint Logic Programs in [7]. In particular, before performing folding steps, we will add suitable atoms in the bodies of the clauses to be folded.

We start off this verification by unfolding clause 1 w.r.t. the atom *reach*. We obtain the following clauses:

2. $\mathit{ur}(S)\leftarrow \mathit{unsafe}(S)\wedge \mathit{init}(S)$

3. $ur(S1) \leftarrow \mathit{unsafe}(S1) \wedge \mathit{cre}(S,S1) \wedge \mathit{reach}(S)$
4. $ur(S1) \leftarrow \mathit{unsafe}(S1) \wedge \mathit{use}(S,S1) \wedge \mathit{reach}(S)$
5. $ur(S1) \leftarrow \mathit{unsafe}(S1) \wedge \mathit{rel}(S,S1) \wedge \mathit{reach}(S)$

Now we can remove clause 2 because

$M(\mathit{Unsafe} \cup \mathit{Init}) \models \neg\exists S\, (\mathit{unsafe}(S) \wedge \mathit{init}(S))$.

The proof of this facts and the proofs of the other facts we state below, are performed by applying the unfold/fold proof method of Section 5. Then, we fold clauses 3 and 5 by using the definition clause 1 and we obtain:

6. $ur(S1) \leftarrow \mathit{unsafe}(S1) \wedge \mathit{cre}(S,S1) \wedge ur(S)$
7. $ur(S1) \leftarrow \mathit{unsafe}(S1) \wedge \mathit{rel}(S,S1) \wedge ur(S)$

Notice that this application of the folding rule is justified by the following two facts:

$M(\mathit{Unsafe} \cup \mathit{Cre}) \models \forall S \forall S1\ (\mathit{unsafe}(S1) \wedge \mathit{cre}(S,S1) \rightarrow \mathit{unsafe}(S))$
$M(\mathit{Unsafe} \cup \mathit{Rel}) \models \forall S \forall S1\ (\mathit{unsafe}(S1) \wedge \mathit{rel}(S,S1) \rightarrow \mathit{unsafe}(S))$

so that, before folding, we can add the atom $\mathit{unsafe}(S)$ to the bodies of clauses 3 and 5. Now, since $M(\mathit{Unsafe} \cup \mathit{Use}) \models \neg\forall S \forall S1\, (\mathit{unsafe}(S1) \wedge \mathit{use}(S,S1) \rightarrow \mathit{unsafe}(S))$, clause 4 *cannot* be folded using the definition clause 1. Thus, we introduce the new definition clause:

8. $p1(S) \leftarrow c(S) \wedge \mathit{reach}(S)$

where $c(\langle W,U\rangle) \equiv \exists n\, (n \in W \wedge \exists Z\, (Z = W \cup U \wedge \mathit{min}(Z,n))) \wedge \neg \mathit{empty}(U)$ which means that: in the system state $\langle W,U\rangle$ there is at least one process which uses the resource and there exists a process waiting for the resource with counter n which is the minimum counter in $W \cup U$.

Notice that, by applying the unfold/fold synthesis method, we may derive a set, called *Busy* (not listed here), of definite clauses which define $c(S)$.
By using clause 8 we fold clause 4, and we obtain:

9. $ur(S1) \leftarrow \mathit{unsafe}(S1) \wedge \mathit{use}(S,S1) \wedge p1(S)$

We proceed by applying the unfolding rule to the newly introduced clause 8, thereby obtaining:

10. $p1(S) \leftarrow c(S) \wedge \mathit{init}(S)$
11. $p1(S1) \leftarrow c(S1) \wedge \mathit{cre}(S,S1) \wedge \mathit{reach}(S)$
12. $p1(S1) \leftarrow c(S1) \wedge \mathit{use}(S,S1) \wedge \mathit{reach}(S)$
13. $p1(S1) \leftarrow c(S1) \wedge \mathit{rel}(S,S1) \wedge \mathit{reach}(S)$

Clauses 10 and 12 are removed, because

$M(\mathit{Busy} \cup \mathit{Init}) \models \neg\exists S\, (c(S) \wedge \mathit{init}(S))$
$M(\mathit{Busy} \cup \mathit{Use}) \models \neg\exists S\, \exists S1\, (c(S1) \wedge \mathit{use}(S,S1))$

We fold clauses 11 and 13 by using the definition clauses 8 and 1, respectively, thereby obtaining:

14. $p1(S1) \leftarrow c(S1) \wedge\ \mathit{cre}(S,S1) \wedge p1(S)$
15. $p1(S1) \leftarrow c(S1) \wedge\ \mathit{rel}(S,S1) \wedge ur(S)$

Notice that this application of the folding rule is justified by the following two facts:

$M(\mathit{Busy} \cup \mathit{Cre}) \models \forall S \forall S1\ ((c(S1) \wedge \mathit{cre}(S, S1)) \rightarrow c(S))$
$M(\mathit{Busy} \cup \mathit{Rel}) \models \forall S \forall S1\ ((c(S1) \wedge \mathit{rel}(S, S1)) \rightarrow \mathit{unsafe}(S))$

Thus, starting from program $P'_{DBakery} \cup \{\text{clause 1}\}$ we have derived a new program Q consisting of clauses 6, 7, 14, and 15. Since all clauses in $\mathit{Def}^*(\mathit{ur}, Q)$ are recursive, we have that for every ground term s denoting a finite set of counters, $\mathit{ur}(s) \notin M(Q)$ and by the correctness of the transformation rules [18], we conclude that mutual exclusion holds for the *DBakery* protocol.

8 Related Work and Conclusions

We have proposed an automatic synthesis method based on unfold/fold program transformations for translating CLP(WS1S) programs into normal logic programs. This method can be used for avoiding the use of ad-hoc solvers for WS1S constraints when constructing proofs of properties of infinite state multiprocess systems.

Our synthesis method follows the general approach presented in [18] and it terminates for any given WS1S formula. No such termination result was given in [18]. In this paper we have also shown that, when we start from a closed WS1S formula φ, our synthesis strategy produces a program which is either (i) a unit clause of the form $f \leftarrow$, where f is a nullary predicate equivalent to the formula φ, or (ii) the empty program. Since in case (i) φ is true and in case (ii) φ is false, our strategy is also a decision procedure for closed WS1S formulas. This result extends [17] which presents a decision procedure based on the unfold/fold proof method for the *clausal fragment* of the WSkS theory, i.e., the fragment dealing with universally quantified disjunctions of conjunctions of literals.

Some related methods based on program transformation have been recently proposed for the verification of infinite state systems [14,21]. However, as it is shown by the example of Section 7, an important feature of our verification method is that the number of processes involved in the protocol may change over time and other methods find it problematic to deal with such dynamic changes. In particular, the techniques presented in [21] for verifying safety properties of *parametrized systems* deal with reactive systems where the number of processes is a parameter which does not change over time.

Our method is also related to a number of other methods which use logic programming and, more generally, constraint logic programming for the verification of reactive systems (see, for instance, [6,9,16,20] and [8] for a survey). The main novelty of our approach w.r.t. these methods is that it combines logic programming and monadic second order logic, thereby modelling in a very direct way systems with an unbounded (and possibly variable) number of processes.

Our unfold/fold synthesis method and our unfold/fold proof method have been implemented by using the MAP transformation system [24]. Our implementation is reasonably efficient for WS1S formulas of small size (see the example formulas of Section 7). However, our main concern in the implementation was not efficiency and our system should not be compared with ad-hoc, well-established theorem provers for WS1S formulas based on automata theory, like the MONA system [10]. Nevertheless, we believe that our technique has its novelty and deserves to be developed because, being based on unfold/fold rules, it can easily be combined with other techniques for program derivation, specialization, synthesis, and verification, which are also based on unfold/fold transformations.

References

1. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19, 20:9–71, 1994.
2. D. Basin and S. Friedrich. Combining WS1S and HOL. In D.M. Gabbay and M. de Rijke, editors, *Frontiers of Combining Systems 2*, volume 7 of *Studies in Logic and Computation*, pages 39–56. Research Studies Press/Wiley, 2000.
3. D. Basin and N. Klarlund. Automata based symbolic reasoning in hardware verification. *The Journal of Formal Methods in Systems Design*, 13(3):255–288, 1998.
4. J. R. Büchi. Weak second order arithmetic and and finite automata. *Z. Maath Logik Grundlagen Math*, 6:66–92, 1960.
5. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.
6. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, Lecture Notes in Computer Science 1579, pages 223–239. Springer-Verlag, 1999.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of sets of infinite state systems using program transformation. In A. Pettorossi, editor, *Proceedings of LOPSTR 2001, Eleventh International Workshop on Logic-based Program Synthesis and Transformation*, Lecture Notes in Computer Science 2372, pages 111–128. Springer-Verlag, 2002.
8. L. Fribourg. Constraint logic programming applied to model checking. In A. Bossi, editor, *Proc. 9th Int. Workshop on Logic-based Program Synthesis and Transformation (LOPSTR'99), Venezia, Italy, Sept. 1999*, Lecture Notes in Computer Science 1817, pages 31–42. Springer, 2000.
9. L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *CONCUR '97*, Lecture Notes in Computer Science 1243, pages 96–107. Springer-Verlag, 1997.
10. J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19-20, 1995*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 1996.
11. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
12. N. Klarlund, M. Nielsen, and K. Sunesen. Automated logical verification based on trace abstraction. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 101–110. ACM, 1996.
13. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

14. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proceedings of LOPSTR '99, Venice, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 1999.
15. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second Edition.
16. U. Nilsson and J. Lübcke. Constraint logic programming for local and symbolic model-checking. In J. W. Lloyd, editor, *First International Conference on Computational Logic, CL'2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 384–398, 2000.
17. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In J. W. Lloyd, editor, *First International Conference on Computational Logic, CL'2000, London, UK, 24-28 July, 2000*, Lecture Notes in Artificial Intelligence 1861, pages 613–628. Springer, 2000.
18. A. Pettorossi and M. Proietti. Program Derivation = Rules + Strategies. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond* (*Essays in honour of Bob Kowalski, Part I*), Lecture Notes in Computer Science 2407, pages 273–309. Springer, 2002.
19. A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *Proc. 24-th ACM Symposium on Principles of Programming Languages, Paris, France*, pages 414–427. ACM Press, 1997.
20. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *CAV '97*, Lecture Notes in Computer Science 1254, pages 143–154. Springer-Verlag, 1997.
21. A. Roychoudhury and I.V. Ramakrishnan. Automated inductive verification of parameterized protocols. In *CAV 2001*, pages 25–37, 2001.
22. K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, and E. Johnson. The XSB system, version 2.2., 2000.
23. J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.
24. The MAP group. The MAP transformation system. Available from `http://www.iasi.rm.cnr.it/~proietti/system.html`, 1995–2002.
25. W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–455. Springer, 1997.