Proofs by Program Transformations*

Abhik Roychoudhury[1], K. Narayan Kumar[1,2], C.R. Ramakrishnan[1], I.V. Ramakrishnan[1]

[1] Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794, USA
{abhik,kumar,cram,ram}@cs.sunysb.edu

[2] Chennai Mathematical Institute
92 G.N. Chetty Road
Chennai, India
kumar@smi.ernet.in

1 Introduction

Logic program transformation systems are often described as a collection of *unfolding*, *folding* and *goal replacement* transformation rules. Given a program P, a logic program transformation system derives a sequence of programs $P = P_0, P_1, \ldots, P_N$, such that for all $0 \leq i < N$, P_{i+1} is obtained from P_i by application of one of the above rules. Logic program transformation systems are usually proved correct by showing that all programs in the transformation sequence $P_0, P_1, \ldots, P_N$ are equivalent under a suitable semantics, such as the least Herbrand model semantics for definite programs.

An unfold/fold transformation system for definite logic programs was first described in a seminal paper by Tamaki and Sato [25]. Since then their system has been substantially extended and expanded [1, 8, 12, 24, 26] and applied to practical problems of importance (e.g., see [2, 4, 14, 18]). An excellent survey of research on this topic appears in [16].

Unfold/fold logic program transformation systems have been extensively used for program synthesis and program optimization. However, relatively little work has been done on using such systems for constructing *proofs*. Hsiang and Srivas [10] extended Prolog's evaluation with "limited forward chaining" to perform inductive theorem proving. This limited forward chaining step is in fact a restricted form of folding: only the theorem statement (which is restricted to be conjunctive) can be used as a folder clause. Kanamori and Fujita [11] used the original Tamaki-Sato unfold/fold transformation system [25] to facilitate the construction of induction proofs of theorems about the original program. However, this method performs *conjunctive* folding using only a single non-recursive clause, and does not perform goal replacement. Proietti and Pettorossi [19, 17] use Tamaki-Sato style unfolding and folding rules (the extension of [25] to disjunctive folding, as described in [8]) to prove the equivalence of two atoms. Their motivation is to use these equivalence proofs for program synthesis.

The need for permitting goal replacement and folding using multiple recursive clauses arises in the construction of induction proofs for automated verification of *parameterized* concurrent systems. A parameterized system, such as an m-bit shift register or a token ring of m processes, constitutes an *infinite* family of finite-state systems, one for each value of m. With finite-state model checking [5], a well-known technique used in automated verification, one is limited to verifying particular instances of such systems (such as a 16-bit shift register). Techniques for automated (or semi-automated) construction of explicit induction proofs is central to the verification of parameterized concurrent systems.

In this paper, we examine how unfolding, folding and goal replacement transformations can be used towards automating the construction of such induction proofs. In [22] we proposed an abstract unfold/fold transformation framework for definite logic programs. We also constructed SCOUT,[1] a concrete instance of the framework, that is strictly more powerful (in terms of transformation sequences permitted) than other unfold/fold transformation systems proposed in the literature. SCOUT combines the stratification-based Tamaki-Sato system in [26] with the counter-based Kanamori-Fujita system in [12] thereby obtaining a single system that strictly subsumes either of them. For each clause of any program in a transformation sequence, the SCOUT system maintains *approximate* counters thereby allowing disjunctive folding using recursive clauses (for details, refer [22]).

*The work of Abhik Roychoudhury, C.R. Ramakrishnan and I.V. Ramakrishnan was partially supported by NSF grants CCR-9711386 and EIA-9705998. The work of K. Narayan Kumar was partially supported by NSF grant CDA-9805735.

[1]SCOUT stands for Strata and COunter based Unfold/fold Transformations.

In order to use SCOUT as a proof system, a second look at the goal replacement transformation is necessary. Goal replacement, where semantically equivalent goals are interchanged, creates more opportunities for folding. This enables one to construct more complex proofs that arise in the verification of parameterized concurrent systems. There are two immediate problems with integrating goal replacement in an automated proof system. First, the identification of equivalent goals must be based on some syntactic (or analysis-based) criteria, since semantic equivalence is, in general, undecidable. Secondly, the conditions under which goal replacement is permitted by the transformation system are usually specified in terms of uncomputable measures such as the size of the shortest ground proofs of atoms in the original program. We need to distill these conditions into those that are *testable* at transformation time. To this end, we introduce *Syntactic Goal Replacement*, a testable goal replacement rule. This enables us to use program transformations for semi-automated deduction. We illustrate the power of SCOUT and Syntactic Goal Replacement by deriving an induction proof for an example distilled from verification of parameterized concurrent systems. The example shows the utility of the more general folding rule of SCOUT as well as the need for Syntactic goal replacement.

2 A Program Transformation System for Constructing Proofs

In this section, we present the transformation rules which will be used to establish induction proofs of universally quantified formulas. First, let us look at the following example to illustrate the kind of properties we intend to prove. Consider the program P_0 given below.

```
thm(X) :- gen(X), test(X).
gen([]).                      test(X) :- canon(X).
gen([0|X]) :- gen(X).         test(X) :- trans(X,Y), test(Y).
canon([]).                    trans([0|X], [1|X]).
canon([1|X]) :- canon(X).     trans([1|T],[1|T1]) :- trans(T,T1).
```

Any list consisting of only 0's is generated by `gen` while the `test` predicate checks whether a given list can be transformed (through finite number of applications of `trans`) into a string consisting of only 1's. The `trans` predicate transforms a string by converting the leftmost occurrence of 0 in the string to 1. The property that we would like to establish is $\forall$ `X` `gen(X)` $\Rightarrow$ `test(X)`.

Now, from the definition of `thm` in P_0 we see that $\forall$ `X` `thm(X)` $\Leftrightarrow$ `gen(X)` $\wedge$ `test(X)`. Thus, if we can establish that $\forall$ `X` `thm(X)` $\Leftrightarrow$ `gen(X)` then we can conclude that the formula $\forall$ `X` `gen(X)` $\Rightarrow$ `test(X)` is true. One way to establish $\forall$ `X` `thm(X)` $\Leftrightarrow$ `gen(X)` is to show that the above program P_0 is equivalent to some program P_N in which the semantic equivalence of `thm(X)` and `gen(X)` can be inferred from the syntax of the program. For example, if the clauses for `thm` in P_N were :

```
thm([]).
thm([0|X]) :- thm(X).
```

then we would be able to infer that $\forall$ `X` `thm(X)` $\Leftrightarrow$ `gen(X)`. This is because the clauses of `gen(X)` and `thm(X)`, even though not syntactically identical, have identical "recursive structure" (we formalize this notion later in this paper).

We demonstrate the equivalence of two programs by showing that one of them can be transformed into the other by repeated application of the unfolding, folding and goal replacement transformation rules. In order to use such transformation rules for automated deduction, the test for applicability and the application of the rules must be automated. Moreover when more than one transformation rule is applicable (as is typically the case), control strategies are needed to decide which rule to apply. In this paper we address only the first issue.

2.1 The SCOUT system

We assume that the predicate symbols appearing in a transformation sequence $P_0, P_1, \ldots, P_N$ are a-priori partitioned into n strata, such that a predicate from stratum j is defined in the initial program P_0 using only predicates from strata $\leq j$. We also assume that for any two n-tuples of integers $\gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$ and $\gamma' = \langle \gamma'_1, \ldots, \gamma'_n \rangle$, we say $\gamma > \gamma'$ if γ is lexicographically greater than γ'. Addition and subtraction of n-tuples are defined as follows: $\gamma + \gamma' = \langle \gamma_1 + \gamma'_1, \ldots, \gamma_n + \gamma'_n \rangle$ and $\gamma \Leftrightarrow \gamma' = \langle \gamma_1 \Leftrightarrow \gamma'_1, \ldots, \gamma_n \Leftrightarrow \gamma'_n \rangle$. We now define

Definition 1 (Characteristic Vector) *Let P be a program such that the predicate symbols of P are partitioned into n strata. Let A be any atom in the Herbrand Base of P. Then, the characteristic vector of A is $ch(A) = \langle w_1, \ldots, w_n \rangle$ where $w_j = 1$ if the predicate symbol of A is in stratum j and 0 otherwise.*

Thus the characteristic vector of an atom A is simply an encoding of the stratum in which the predicate symbol of A belongs. We now describe the unfolding and folding rules of SCOUT [22]. For any transformation sequence $P_0, P_1, ..., P_N$ each clause C in program P_i is annotated with a pair of *stratified counters*. In other words, every clause $C \in P_i$ in a transformation sequence is annotated with a pair $(\gamma^i_{lo}(C), \gamma^i_{hi}(C))$, where $\gamma^i_{lo}(C), \gamma^i_{hi}(C) \in \mathbb{Z}^n$ and $\gamma^i_{hi}(C) \geq \gamma^i_{lo}(C)$. We assume that every clause $C \equiv A{:}\Leftrightarrow A_1, \ldots, A_k$ in the initial program P_0 is annotated with the pair $(ch(A), ch(A))$.

Rule 1 (Unfolding) Let C be a clause in P_i and A an atom in the body of C. Let $C_1, ..., C_m$ be the clauses in P_i whose heads are unifiable with A with most general unifiers $\sigma_1, ..., \sigma_m$. Let C'_j be the clause that is obtained by replacing $A\sigma_j$ by the body of $C_j\sigma_j$ in $C\sigma_j$ $(1 \leq j \leq m)$.
Then, assign $P_{i+1} := (P_i \Leftrightarrow \{C\}) \cup \{C'_1, ..., C'_m\}$. Also, set $\gamma^{i+1}_{lo}(C'_j) = \gamma^i_{lo}(C) + \gamma^i_{lo}(C_j)$ and $\gamma^{i+1}_{hi}(C'_j) = \gamma^i_{hi}(C) + \gamma^i_{hi}(C_j)$. The annotations of all other clauses in P_{i+1} is inherited from P_i. □

Rule 2 (Folding) Let $C_1, ..., C_m$ be clauses in P_i of the form $C_l \equiv A{:}\Leftrightarrow A_{l,1}, ..., A_{l,n_l}, A'_1, ..., A'_p$ and $D_1, ..., D_m$ be clauses in P_j $(j \leq i)$ of the form $D_l \equiv B_l{:}\Leftrightarrow B_{l,1}, ..., B_{l,n_l}$ satisfying :
1. $\forall 1 \leq l \leq m.\ \exists \sigma_l.\ \forall 1 \leq k \leq n_l.\ A_{l,k} = B_{l,k}\sigma_l$ where σ_l is a substitution.
2. $B_1\sigma_1 = B_2\sigma_2 = ... = B_m\sigma_m = B$
3. $D_1, ..., D_m$ are the only clauses in P_j whose heads are unifiable with B.
4. $\forall 1 \leq l \leq m$ σ_l substitutes the internal variables of D_l to distinct variables which do not appear in $\{A, B, A'_1, ..., A'_p\}$
5. $\forall 1 \leq l \leq m$ $\gamma^j_{hi}(D_l) < \gamma^i_{lo}(C_l) + \sum_{1 \leq k \leq p} ch(A'_k)$
Then, assign $P_{i+1} := (P_i \Leftrightarrow \{C_1, ..., C_m\}) \cup \{C'\}$ where $C' \equiv A{:}\Leftrightarrow B, A'_1, ..., A'_p$. Also set
$\gamma^{i+1}_{lo}(C') = min_{1 \leq l \leq m}(\gamma^i_{lo}(C_l) \Leftrightarrow \gamma^j_{hi}(D_l))$, $\gamma^{i+1}_{hi}(C') = max_{1 \leq l \leq m}(\gamma^i_{hi}(C_l) \Leftrightarrow \gamma^j_{lo}(D_l))$. The annotations of all other clauses in P_{i+1} is inherited from P_i. □

The unfolding and folding rules transform a program P_i with annotated clauses to another program P_{i+1} with annotated clauses. One key feature of the folding rule is that it permits disjunctive folding of recursive clauses. As discussed later, such folding is common in the proofs of verification of parameterized concurrent systems.

The goal replacement transformation allows semantically equivalent atoms to be interchanged. To present this rule, we will need the following definitions. For any conjunction of atoms $A_1, \ldots, A_n$ we use the notation $vars(A_1, \ldots, A_n)$ to denote the set of variables occurring in $A_1, \ldots, A_n$.

Definition 2 (Ground Proof of an Atom) *Let T be a tree, each of whose nodes is labeled with a ground atom. Then T is a ground proof in program P, if every node A in T satisfies the condition : $A{:}\Leftrightarrow A_1, ..., A_n$ is a ground instance of a clause in P, where $A_1, ..., A_n$ $(n \geq 0)$ are the only children of A in the tree T.*

Definition 3 (Weight of a Proof) *Let T be a ground proof in program P of some ground atom $A \in M(P)$. Then, the weight of T is $w(T) = \langle w_1, \ldots, w_n \rangle$ where w_j is the number of nodes in T whose predicate symbol is in stratum j.*

Thus the weight of a given ground proof simply accounts for the number of nodes in the proof tree, for each stratum.

Definition 4 (Weight of an Atom) *Let P_0 be the initial program of a transformation sequence and $A \in M(P_0)$ be a ground atom. Then the weight of A, denoted $w(A)$, is the weight of the lexicographically smallest proof of A in P_0. Thus, $\forall A\ w(A) \geq ch(A)$.*

Note that when the number of strata is 1, the weight of a ground atom A denotes simply the size of the shortest ground proof of A in P_0.

Rule 3 (Goal Replacement) *Let C be a clause $A{:}\Leftrightarrow A_1, \ldots, A_k, G$ in P_i, and G' be an atom such that $vars(G) = vars(G') \subseteq vars(A, A_1, \ldots, A_k)$. Suppose*
1. for all ground instantiation θ of G, G' we have
(i) $P_i \vdash G\theta \Leftrightarrow P_i \vdash G'\theta$ (ii) $\delta \leq w(G\theta) \Leftrightarrow w(G'\theta) \leq \delta'$
2. $\gamma^i_{lo}(C) + \delta + \sum_{1 \leq j \leq k} ch(A_j) > \langle 0, \ldots, 0 \rangle$

Then $P_{i+1} := (P_i \Leftrightarrow \{C\}) \cup \{C'\}$ where $C' \equiv A{:}\Leftrightarrow A_1, \ldots, A_k, G'$. Also set $\gamma^{i+1}_{lo}(C') = \gamma^i_{lo}(C) + \delta$ and $\gamma^{i+1}_{hi}(C') = \gamma^i_{hi}(C) + \delta'$. □

Note that although we replace a single atom G by another atom G' , we can replace conjunctions of atoms using a sequence of definition, folding, goal replacement and unfolding transformations. However, the above rule allows replacement in the body of *only one* clause in program P_i. It would be interesting to study how we can extend this rule to perform multiple replacements simultaneously in P_i without compromising correctness (as discussed in [3]).

Rules 1,2,3 are concrete instances of the transformation rules presented in [22]. Hence, the correctness proof of [22] directly yields the following correctness result w.r.t. least Herbrand model semantics. For any definite logic program P, let $M(P)$ denote its least Herbrand model.

Theorem 1 [22] *Let $P_0, P_1, \ldots, P_N$ be a sequence of definite programs where P_{i+1} is obtained from P_i by unfolding (rule 1), folding (rule 2) or goal replacement (rule 3). Then $\forall\ 0 \le i \le N$ we have $M(P_i) = M(P_0)$.*

2.2 Syntactic Goal Replacement

Unlike the unfolding and folding rules, applicability of goal replacement rule (rule 3) is not testable. Moreover to apply this rule, we need to compute suitable δ, δ' which are bounds on the difference between the weight of replaced and replacing atoms. We now formulate a testable version of the goal replacement rule. To do so, we need a nontrivial computational mechanism to check the semantic equivalence of two given atoms purely based on syntax. We must also identify testable conditions that imply the untestable restrictions on weights of atoms required by the general goal replacement rule. The notion of syntactic equivalence described below addresses the first issue, while the definition of the syntactic goal replacement rule resolves the second issue.

Syntactic Equivalence Consider the following example program P

```
p(X) :- r(X).              q(X) :- s(X).
p(X) :- e(X,Y), p(Y).      q(X) :- e(X,Y), q(Y).
r(X) :- b(X).              s(X) :- b(X).
```

`r(X)` and `s(X)` are equivalent since the clauses defining them have identical right hand sides. We can now use this to infer that `q(X)` and `p(X)` are equivalent. Note that even though the clauses of `p(X)` and `q(X)` are not syntactically identical, the "recursive structure" of these clauses is the same. We formalize this notion in the definition given below.

Definition 5 (Syntactic Equivalence of Atoms) *Let $\cong^P$ be an equivalence relation on the set of predicates of a program P and let $A = p(t_1, \ldots, t_k)$ and $B = q(t'_1, \ldots, t'_k)$ be two atoms. Then atoms A and B are said to be syntactically equivalent w.r.t. to the relation $\cong^P$, denoted $A \cong^P_{atom} B$, if we have $p \cong^P q$ and $(t_1, \ldots, t_k)$ is a variant of $(t'_1, \ldots, t'_k)$*

Definition 6 (Syntactic Equivalence of Predicates) *An equivalence relation $\cong^P$ on the set of predicates of P is said to be a syntactic equivalence relation if whenever $p \cong^P q$ we have:*
1. The predicates p and q belong to the same stratum.
2. Let the clauses of predicates p and q in program P be $\{C_1, \ldots, C_m\}$ and $\{D_1, \ldots, D_m\}$ respectively. Then, for all $1 \le i \le m$ we have :
(i) *C_i is a variant of D_i when all predicate symbols in C_i and D_i are replaced with the same predicate.*
(ii) *Let C_i and D_i be of the form $H {:}\Leftrightarrow B_1, ..., B_k$ and $H' {:}\Leftrightarrow B'_1, ..., B'_k$ respectively. Then for all $1 \le l \le k$ $B_l \cong^P_{atom} B'_l$.*

It is easy to see that the family of syntactic equivalence relations is closed under union. Thus there is a largest syntactic equivalence relation $\equiv^P$. The relation $\equiv^P$ can be computed as the greatest fixed point of a functional derived from the conditions of definition 6. In practice, in order to show that $p \equiv^P q$ one often computes a smaller syntactic equivalence relation. This is because $p \equiv^P q$ if and only if $p \cong^P q$ for some syntactic equivalence relation $\cong^P$. Also, for two atoms A and B we say $A \equiv^P_{atom} B$ if and only if $A \cong^P_{atom} B$ for some syntactic equivalence relation $\cong^P$.

Thus, given two atoms A and B and a program P, we can check whether $A \equiv^P_{atom} B$ automatically. In the example program fragment P given above, the the relation $\cong^P = \{(\mathtt{p}, \mathtt{q}), (\mathtt{r}, \mathtt{s})\} \cup Id$, where Id is the identity relation, is a syntactic equivalence relation. Therefore $\mathtt{p(X)} \cong^P_{atom} \mathtt{q(X)}$ and hence $\mathtt{p(X)} \equiv^P_{atom} \mathtt{q(X)}$.

Note that we can straightforwardly *generalize* our definition of syntactic equivalence to define syntactic equivalence of subgoals. Thus, we can then make inferences like $\mathtt{p(f(X))} \equiv \mathtt{q(X)}$ based on the syntax[2]. The details appear in [21] and will be incorporated into the full version of this paper.

[2]With definitions 5, 6 we can only infer $\mathtt{p(f(X))} \equiv \mathtt{q(f(X))}$ if $\mathtt{p} \equiv \mathtt{q}$.

We establish the following lemma about ground proofs of syntactically equivalent atoms. Thus, if two atoms G and G' are syntactically equivalent in program P, then for every ground proof of a ground instance of G in P there is a ground proof (of equal weight) of the corresponding ground instance of G' in P.

Lemma 1 (Proofs of syntactically equivalent atoms) *For any ground proof T of a ground atom $G\theta$ in program P if $G \equiv_{atom}^{P} G'$, then there is a ground proof T' of $G'\rho\theta$ (where the substitution ρ renames the variables of G to the corresponding variables of G') in P s.t. $w(T) = w(T')$,* i.e. *the weights of T and T' are equal.*

Proof : By induction on the size of proof T. $\Box$

We now introduce the notion of *relevant clause set* of an atom. Intuitively, it is a conservative estimate (*i.e.* a superset) of the set of clauses which are used in the proof of some ground instance of the atom.

Definition 7 (Relevant Clause Set) *Let A be an atom and P a program. Let $reach(A, P)$ denote the set of predicates which are reachable from the predicate of A in the predicate dependency graph*[3] *of P. Then, the relevant clause set of A in P (denoted $rel(A, P)$) is the set of clauses of the predicates in $reach(A, P)$.*

We now define the Syntactic Goal Replacement rule. For any clause C, $hd(C)$ denotes the head atom of C.

Rule 4 (Syntactic Goal Replacement) Let C be a clause in P_i of the form $A {:\Leftrightarrow} A_1, ..., A_k, G$ and consider another clause C' (not in P_i) of the form : $A {:\Leftrightarrow} A_1, ..., A_k, G'$ such that
1. G and G' are syntactically equivalent *i.e.* $G \equiv_{atom}^{P_i} G'$, and $vars(G) = vars(G') \subseteq vars(A, A_1, \ldots, A_k)$
2. The clauses in $rel(G', P_0)$ are never modified in the transformation sequence $P_0, P_1, \ldots, P_i$ *i.e.* $rel(G', P_0) = rel(G', P_i)$.
3. For each clause $D \in P_i$ $\gamma_{lo}^{i}(D) \geq ch(hd(D))$.
4. Let $Cl_i(G)$ be the clauses in P_i whose heads unify with the atom G. We define
$\delta = min_{D \in Cl_i(G)} (\gamma_{lo}^{i}(D) \Leftrightarrow ch(hd(D)))$. We must have $\gamma_{lo}^{i}(C) + \delta + \sum_{1 \leq j \leq k} ch(A_j) > \langle 0, \ldots, 0 \rangle$.

Then, assign $P_{i+1} := (P_i \Leftrightarrow \{C\}) \cup \{C'\}$ where C' is $A {:\Leftrightarrow} A_1, ..., A_k, G'$. Also, set $\gamma_{lo}^{i+1}(C') = \gamma_{lo}^{i}(C) + \delta$ and $\gamma_{hi}^{i+1}(C') = \gamma_{hi}^{i}(C) + \delta'$ where $\delta' = \langle \infty, 0, \ldots, 0 \rangle$. [4] $\Box$

Special case of Goal Replacement Syntactic Goal Replacement (rule 4) can be proved to be a special case of the Goal Replacement transformation of rule 3. Note that since $G \equiv_{atom}^{P_i} G'$ and $vars(G) = vars(G')$, therefore by lemma 1 we have $\forall \theta \; P_i \vdash G\theta \Leftrightarrow P_i \vdash G'\theta$. We also have $\gamma_{lo}^{i}(C) + \delta + \sum_{1 \leq j \leq k} ch(A_j) > \langle 0, \ldots, 0 \rangle$. Thus, to show that Syntactic Goal Replacement is a special case of Goal Replacement, it is sufficient to prove that whenever Syntactic Goal Replacement is applied to clause C to replace G by G' we have $\forall \theta \; \delta \leq w(G\theta) \Leftrightarrow w(G'\theta) \leq \delta'$. Since $\delta' = \langle \infty, 0, \ldots, 0 \rangle$, therefore δ' is lexicographically greater than the weight of any ground atom; hence $w(G\theta) \Leftrightarrow w(G'\theta) \leq \delta'$. Finally, conditions (2) and (3) of the Syntactic Goal Replacement rule ensure that $\forall \theta \; w(G\theta) \Leftrightarrow w(G'\theta) \geq \delta$. A formal proof showing that rule 4 is a special case of rule 3 can be found in [21].

Applicability of rule 4 is testable and the clause annotations of the new clause C' can be effectively computed since we have conservatively estimated the value of δ, δ'. Note that in rule 4 we have set δ' to $\langle \infty, 0, \ldots, 0 \rangle$. This will prevent the new clause C' from being used as a folder later in the transformation sequence. However, our choice of δ satisfies $\delta \geq \langle 0, \ldots, 0 \rangle$ and therefore we will always have $\gamma_{lo}^{i+1}(C') \geq \gamma_{lo}^{i}(C)$. Thus, C' can participate in future folding transformations as one of the folded clauses. Also, note that a tighter value of δ' is hard to obtain. This is because we need to satisfy $w(G\theta) \Leftrightarrow w(G'\theta) \leq \delta'$ for any ground substitution θ. The proof sizes of $G\theta$ and $G'\theta$ could be monotonic on the instantiation of some variable of G, G' and θ could be constructed to instantiate that variable to larger and larger ground terms, thereby ruling out a tighter value of δ'.

From the correctness of the unfolding, folding and goal replacement rules (theorem 1), we obtain the following correctness result.

Theorem 2 (Correctness of Unfolding, Folding, Syntactic Goal Replacement) *Let $P_0, P_1, \ldots, P_N$ be a sequence of definite logic programs where P_{i+1} is obtained from P_i by unfolding (rule 1), folding (rule 2) or syntactic goal replacement (rule 4). Then $\forall \; 0 \leq i \leq N$ we have $M(P_i) = M(P_0)$.*

[3]The predicate dependency graph of a program P has the predicate symbols of P as its vertices, and there is an edge from predicate p to predicate q if q occurs in the body of a clause of p in program P.

[4]Note that ∞ is only a notational convenience. It represents a value that exceeds the weights of all atoms. Formally, this is achieved by extending the clause annotations by one extra stratum.

3 An Induction Proof

We now illustrate the use of our program transformation rules by transforming the generate-test program described at the beginning of Section 2 to the desired form. The program P_0 is given below. In P_0, all predicates are in the same stratum and the lower and upper clause measures are set to 1 for all clauses. In the following, the clause annotations $(\gamma^i_{lo}(C), \gamma^i_{hi}(C))$ for any clause $C \in P_i$ are shown in parentheses beside clause C. Recall that we want to prove the universally quantified formula $\forall$ X `gen(X)` $\Rightarrow$ `test(X)`.

```
thm(X) :- gen(X), test(X)  (1,1)
gen([]).                   (1,1)  test(X) :- canon(X).                (1,1)
gen([0|X]) :- gen(X).      (1,1)  test(X) :- trans(X,Y), test(Y).     (1,1)
canon([]).                 (1,1)  trans([0|X], [1|X]).                (1,1)
canon([1|X]) :- canon(X).  (1,1)  trans([1|T],[1|T1]) :- trans(T,T1). (1,1)
```

Unfolding the only clause of `thm/1` several times we obtain :

```
thm([]).                                        (4,4)
thm([0|X]) :- gen(X), canon(X).                 (6,6)
thm([0|X]) :- gen(X), trans(X,Y), test([1|Y]).  (6,6)
```

We now introduce the definition :

```
test1(Y) :- test([1|Y]).  (1,1)
```

and fold the occurrence of `test([1|Y])` in the last clause of `thm/1` to obtain :

```
thm([]).                                   (4,4)
thm([0|X]) :- gen(X), canon(X).            (6,6)
thm([0|X]) :- gen(X), trans(X,Y), test1(Y).  (5,5)
```

Unfolding the definition clause of `test1/1` several times and then folding (using the definition clause of `test1/1` as the folder) we get :

```
test1(Y) :- canon(Y).             (3,3)
test1(Y) :- trans(Y,Z), test1(Z).  (2,2)
```

Note that `test1(Y)` and `test(Y)` are syntactically equivalent, since the clauses of `test/1` are :

```
test(X) :- canon(X).             (1,1)
test(X) :- trans(X,Y), test1(Y).  (1,1)
```

We therefore apply *Syntactic Goal Replacement* in the last clause of `thm/1` to obtain the following :

```
thm([]).                                    (4,4)
thm([0|X]) :- gen(X), canon(X).             (6,6)
thm([0|X]) :- gen(X), trans(X,Y), test(Y).   (6,∞)
```

We can now fold the above clauses of `thm/1` using the clauses of `test/1` as the folder. Note that we are folding using *multiple recursive clauses as the folder*. Thus, we obtain :

```
thm([]).                        (4,4)
thm([0|X]) :- gen(X), test(X).  (5,∞)
```

Finally, we fold using the clause of `thm/1` in P_0 as folder to obtain the program P_{final}

```
thm([]).                   (4,4)
thm([0|X]) :- thm(X).      (4,∞)
gen([]).                   (1,1)  test(X) :- canon(X).                 (1,1)
gen([0|X]) :- gen(X).      (1,1)  test(X) :- trans(X,Y), test(Y).      (1,1)
canon([]).                 (1,1)  trans([0|T],[1|T]).                  (1,1)
canon([1|X]) :- canon(X).  (1,1)  trans([1|T], [1|T1]) :- trans(T,T1). (1,1)
```

The atoms `thm(X)` and `gen(X)` are now syntactically equivalent (refer definition 6). Thus, $M(P_{final}) \models \forall$ X `thm(X)` $\Leftrightarrow$ `gen(X)`. Since $M(P_{final}) = M(P_0)$ therefore $M(P_0) \models \forall$ X `thm(X)` $\Leftrightarrow$ `gen(X)`. By definition of `thm(X)` in P_0, this means that $M(P_0) \models \forall$ X `gen(X)` $\Rightarrow$ `test(X)`. Thus, by using SCOUT and Syntactic Goal Replacement, we have constructed a nontrivial induction proof.

Verification of Parameterized Concurrent Systems The above proof is illustrative since it is structurally similar to the proofs that arise in the verification of concurrent systems. In fact, using the transformation rules of SCOUT and the Syntactic Goal Replacement rule in a similar fashion we verified properties like liveness of a m-bit shift register, correctness of a m-bit carry-lookahead adder etc. Thus, in the problem of verification of liveness of a m-bit shift register : the predicate `gen` represents the encoding of the m-bit shift register while the predicate `test` represents the encoding of the liveness property that we verify. To accomplish the proof of liveness for any m, we perform a folding step using `test` as the folder similar to the above example. Moreover, as in the above example, the proof of liveness also involves application of the Syntactic Goal Replacement rule to replace a specialized version of `test`. This corresponds to proving that the liveness property holds in a process Q iff the property holds in a sub-process of Q. The verification examples are not shown here due to space considerations. Some of them are available in [23], and they will be incorporated in the full version of this paper.

Interestingly, recent advances in logic programming based model-checking [7] open up the possibility of using logic program transformations for the verification of parameterized concurrent systems. The XMC model-checker [20], built on top of the XSB tabled logic programming system [27], can verify finite-state systems specified using value-passing CCS [15] and formulas expressed in modal mu-calculus [13]. XMC's space and time performance is competitive with hand-coded (in C/C++) model checkers such as the Concurrency Factory [6] and SPIN [9] from Bell Labs. Essentially, the XMC model-checker performs verification of finite-state concurrent systems through controlled and efficient unfolding. By using meta-programming facilities of logic programming, one can implement deductive techniques (achieved through folding and goal replacement transformations) and thereby integrate them tightly with algorithmic model checking (which is done by applying unfolding). This indicates that a unfold/fold logic program transformation framework holds promise as a framework for integration of algorithmic and deductive verification in general, and verification of parameterized concurrent systems in particular.

References

[1] C. Aravindan and P.M. Dung. On the correctness of unfold/fold transformations of normal and extended logic programs. *Journal of Logic Programming*, pages 295–322, 1995.

[2] A. Bossi, N. Cocco, and S. Dulli. A method of specializing logic programs. *ACM TOPLAS*, pages 253–302, 1990.

[3] A. Bossi, N. Cocco, and S. Etalle. Simultaneous replacement in normal programs. *Journal of Logic and Computation*, 6(1):79–120, February 1996.

[4] D. Boulanger and M. Bruynooghe. Deriving unfold/fold transformations of logic programs using extended OLDT-based abstract interpretation. *Journal of Symbolic Computation*, pages 495–521, 1993.

[5] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS,* 8(2), 1986.

[6] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In *Proceedings of the Seventh International Conference on Computer Aided Verification (CAV '96),* Vol. 1102 of *Lecture Notes in Computer Science*, pages 398–401. Springer-Verlag, 1996.

[7] B. Cui, Y. Dong, X. Du, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, A. Roychoudhury, S.A. Smolka, and D.S. Warren. Logic programming and model checking. In *Proceedings of PLILP/ALP, LNCS 1490*, pages 1–20, 1998.

[8] M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In *Proceedings of PLILP, LNCS 844*, pages 340–354, 1994.

[9] G. J. Holzmann and D. Peled. The state of SPIN. In *Proceedings of the Seventh International Conference on Computer Aided Verification (CAV '96)*, Vol. 1102 of *Lecture Notes in Computer Science*, pages 385–389. Springer-Verlag, 1996.

[10] J. Hsiang and M. Srivas. Automatic inductive theorem proving using Prolog. *Theoretical Computer Science'*, 54:3–28, 1987.

[11] T. Kanamori and H. Fujita. Formulation of Induction Formulas in Verification of Prolog Programs. *Proceedings of International Conference on Automated Deduction (CADE)*, pages 281–299, 1986.

[12] T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. In *USA-Japan Seminar on Logics of Programs*, 1987.

[13] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[14] M. Leuschel, D. De Schreye, and A. De Waal. A conceptual embedding of folding into partial deduction : Towards a maximal integration. In *Joint International Conference and Symposium on Logic Programming*, pages 319–332, 1996.

[15] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[16] A. Pettorossi and M. Proietti. *Transformation of logic programs*, volume 5 of *Handbook of Logic in Artificial Intelligence*, pages 697–787. Oxford University Press, 1998.

[17] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming, to appear*, 1999.

[18] A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *Proceedings of POPL*, pages 414–427, 1997.

[19] M. Proietti and A. Pettorossi. Synthesis of programs from unfold/fold proofs. In *Proceedings of Logic Program Synthesis and Transformation*, pages 141–158. Springer-Verlag, 1994.

[20] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, Terrance Swift, S.A. Smolka, and D. S. Warren. Efficient model-checking using tabled resolution. *Proceedings of Computer Aided Verification (CAV)*, 1997.

[21] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. A generalized unfold/fold transformation system for definite logic programs. Technical Report 98/37, Dept. of Computer Science, SUNY Stony Brook, 1998.

[22] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. A parameterized unfold/fold transformation framework for definite logic programs. *To appear in proocedings of Principles and Practice of Declarative Programming (PPDP)*, 1999.

[23] A. Roychoudhury, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Automated verification of parameterized systems by extending tabled resolution. Technical report, Department of Computer Science, SUNY Stony Brook, Available at `http://www.cs.sunysb.edu/~abhik/transform/papers.html`, 1998.

[24] H. Seki. Unfold/fold transformation of general logic programs for well-founded semantics. *In Journal of Logic Programming*, pages 5–23, 1993.

[25] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of International Conference on Logic Programming*, pages 127–138, 1984.

[26] H. Tamaki and T. Sato. A generalized correctness proof of the unfold/ fold logic program transformation. Technical report, Ibaraki University, Japan, 1986.

[27] XSB. The XSB logic programming system v1.8, 1998. Available from `http://www.cs.sunysb.edu/~sbprolog`.