Unfold/fold Transformations for Automated Verification of Parameterized Concurrent Systems

Abhik Roychoudhury[1] and C.R. Ramakrishnan[2]

[1] School of Computing, National University of Singapore, Singapore.
abhik@comp.nus.edu.sg
[2] Dept. of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794, USA.
cram@cs.sunysb.edu

Abstract. Formal verification of reactive concurrent systems is important since many hardware and software components of our computing environment can be modeled as reactive concurrent systems. Algorithmic techniques for verifying concurrent systems such as model checking can be applied to finite state systems only. This chapter investigates the verification of a common class of infinite state systems, namely parameterized systems. Such systems are parameterized by the number of component processes, for example an n-process token ring for any n. Verifying the entire infinite family represented by a parameterized system lies beyond the reach of traditional model checking. On the other hand, deductive techniques to verify infinite state systems often require substantial user guidance.

The goal of this work is to integrate algorithmic and deductive techniques for automating proofs of temporal properties of parameterized concurrent systems. Here, the parameterized system to be verified and the temporal property are encoded together as a logic program. The problem of verifying the temporal property is then reduced to the problem of determining equivalence of predicates in this logic program. These predicate equivalences are established by transforming the program such that the semantic equivalence of the predicates can be inferred from the structure of their clauses in the transformed program.

For transforming the predicates, we use the well-established unfold/fold transformations of logic programs. Unfolding represents a step of resolution and can be used to evaluate the base case and the finite part of the induction step in an induction proof. Folding and other transformations represent deductive reasoning and can be used to recognize the induction hypothesis. Together these transformations are used to construct induction proofs of temporal properties. Strategies are developed to help guide the application of the transformation rules. The transformation rules and strategies have been implemented to yield an automatic and programmable first order theorem prover for parameterized systems. Case studies include multi-processor cache coherence protocols and the Java Meta-Locking protocol from Sun Microsystems. The program transformation based prover has been used to automatically prove various safety properties of these protocols.

1 Introduction

Many hardware and software components of our everyday computing environment can be modeled as a *reactive concurrent system*. These include hardware controllers, operating systems, network protocols, and distributed applications *e.g.* air traffic control systems. Intuitively, a reactive concurrent system is a collection of nonterminating processes which run concurrently, and communicate with each other as well as an external environment to perform a common task. Proving correctness of such a system involves showing that it displays some desired behavior. Formally proving correctness of such systems has been a topic of intense research for the past two decades, leading to the birth of successful techniques like *model checking* [8].

Formal verification of reactive systems involves: *(i)* constructing the "specification" *i.e.* the description of the desired behavior(s) of the program, *(ii)* constructing the "implementation" *i.e.* the formal description of the reactive system being verified, and *(iii)* formally proving that the implementation satisfies the specification. Given appropriate formalisms for expressing the specification and implementation, we then need a proof system for establishing that a given implementation satisfies a given specification. Given a proof system and a proof obligation (*i.e.* a given implementation and specification), one needs to construct a proof tree by repeated application of the rules to the proof obligation. In general, this proof tree construction is undecidable [31].

However, for *finite state* concurrent systems, this can be achieved algorithmically by searching *the* finite model of the implementation, *i.e.* by searching the states of the finite state transition system representing the behaviors of the concurrent system. This is the basic idea behind model checking. Model checking [8] is an *automated* formal verification technique for proving properties of finite state concurrent programs. Here the specification is typically provided as a temporal logic formula. The implementation is often expressed using a process calculus, which is translated to a finite state transition system. Verifying the truth of the temporal formula is accomplished by traversing the states of this transition system based on the structure of the temporal formula. If the formula is true, then the search succeeds; otherwise the search fails and yields a counterexample.

The Problem Addressed The applicability of model checking is inherently restricted to finite state systems. Many of the verification tasks one would like to conduct however deal with infinite state systems. In particular, we often need to verify "parameterized" systems such as an n-bit adder or an n-process token ring for any n. Intuitively, a parameterized system is an infinite family of finite state systems parameterized by a recursively defined type *e.g.* the set of natural numbers $\mathbb{N}$. Thus an n-bit adder is a parameterized system, the parameter in question being $n \in \mathbb{N}$, the width of the adder circuit. Verification of distributed algorithms can be naturally cast as verifying parameterized systems, the parameter being the number of processes. For example, consider a distributed algorithm where n users share a resource and follow some protocol to ensure mutually exclusive access. Using model checking, we can verify mutual exclusion only for finite instances of the algorithm, *i.e.* for $n = 2$, $n = 3, \ldots$, but not for every n. The verification of parameterized systems lies beyond the reach of traditional model checkers: the representations and the model-checking algorithms that manipulate these representations are designed to work on finite state systems and it is not at all trivial to adapt them for parameterized system verification.

In general, automated verification of parameterized systems has been shown to be undecidable [3]. Thus, verification of parameterized systems is often accomplished via *theorem proving*, i.e. mechanically checking the steps of a human proof using a deductive system. Even with substantial help from the deductive system in dispensing routine parts of the proof, such theorem proving efforts require considerable user guidance. Alternatively, one can identify subclasses of parameterized systems for which verification is decidable [21, 33]. Using this approach meaningful subclasses have been identified, such as token rings of similar processes [14] and classes of parameterized synchronous systems [17].

The Approach Taken A parameterized system represents an infinite family parameterized by a recursively defined type. Therefore, it is natural to attempt proving properties of parameterized systems by inducting over this type. In this work, we aim to automate the construction of such induction proofs by restricting the deductive machinery for constructing proofs. We construct an automatic and programmable first order logic prover with limited deductive capability.

The research reported in this chapter is part of recent efforts to exploit logic programming technology for developing new tools and techniques to specify and verify concurrent systems. For example, constraint logic programming has been used for the analysis and verification of hybrid systems [49] and more recently for model checking of finite-state [34] / infinite-state systems [13]. In [40], a memoized logic programming engine is used to develop XMC, an efficient and flexible model checker whose performance is comparable to that of highly optimized model checkers such as Spin [22]. Recently, [12] used constraint logic programming to construct proofs of safety properties of parameterized cache coherence protocols. Essentially, these techniques aim to use (constraint) logic program evaluation to efficiently construct verification proofs involving state space search (accomplished via resolution) and (possibly) constraint solving. These techniques do not construct induction proofs and are not applicable to parameterized networks of different topologies; thus, the technique of [12] cannot be used to prove properties of parameterized tree networks. On the other hand, we construct an automatic and programmable first order logic prover with limited deductive capability. This prover is geared to construct nested induction proofs which typically proceed by inducting on the structure of the parameterized network. The core technology of our prover is provided by logic program transformations. We discuss related proof techniques based on program transformations in Section 9.

Our work provides a methodology for constructing induction proofs by suitably extending the resolution based evaluation mechanism of logic programs [42, 44]. In this approach, the parameterized system and the property to be verified is expressed as a logic program. The verification problem is reduced to the problem of determining the equivalence of predicates in this program. The predicate equivalences are then established by transforming the predicates such that their semantic equivalence can be inferred from the syntax of their transformed definitions. The proof of semantic equivalence of two transformed predicates p, p' then proceeds automatically by a routine induction on the size of the proofs of ground instances of $p(\overline{X})$ and $p'(\overline{X})$.

For transforming the predicates, we use the well-established *unfold/fold* transformations of logic programs [46] which have been previously used for program optimization [11, 38] and automated deduction [23, 25, 36]. The major transformations in such a transformation framework are unfolding, folding and goal replacement. One of these transformations (unfolding) represents an application of resolution. In particular, an application of the unfold transformation represents a single resolution step. Therefore, one can achieve on-the-fly explicit state *algorithmic* model checking by repeated unfolding of the verification proof obligation. In constructing induction proofs, unfold transformations are used to evaluate away the base case and the finite portions of the proof in the induction step of the induction argument. Folding and goal replacement, on the other hand, represent a form of *deductive* reasoning. They are used to simplify the given program so that applications of the induction hypothesis in the induction proof can be recognized. The reader should note that the folding transformation is more powerful that the memoing involved in traditional tabled resolution [7, 40, 48]. Folding remembers (disjunctions of) conjunctions of atoms and can be used to remember/recognize the induction hypothesis in an induction proof.

Contributions The contributions of this work can be summarized as follows.

- First, it shows how logic program evaluation based techniques for verifying finite state systems can be flexibly extended to yield a program transformation framework for constructing inductive proofs of temporal properties of parameterized systems. Since one of our transformations corresponds to a model checking step and the others correspond to deductive reasoning, model checking emerges as a special case when the deductive steps are applied lazily.[1]
- The program transformation framework presented here allows for tight integration of algorithmic and deductive verification steps in a proof. Note that application of unfolding and folding steps can be arbitrarily interleaved in the verification proof of a parameterized system. This constitutes a tighter integration of model checking computation with deductive reasoning as compared to the integration of model checking as a decision procedure in a theorem prover.
- Finally, we present terminating strategies for controlling the application of the transformation rules, thereby leading to the implementation of a programmable and fully automatic prover (which is incomplete). These strate-

[1] However, unlike model checking, our augmented proof technique does not generate counter-example evidence. We discuss this issue further in Section 9.

gies are used to construct safety proofs of parameterized systems of various "structures" including uni and bi-directional chains, rings and trees of processes. The prover has been used to construct automated proofs of safety properties of cache coherence protocols. It has also been used to automatically verify mutual exclusion in the Java Meta-Locking Algorithm (a recently developed algorithm to ensure secure access of Java objects by multiple Java threads) from Sun Microsystems [1].

Organization The rest of this chapter is organized as follows. In Section 2 we discuss how we encode the problem of verifying temporal properties of parameterized systems as a logic program. Section 3 presents an overview of our proof technique, while Section 4 presents the proof rules on which our technique is based. Section 5 discusses the automation of each application of a proof rule while Section 6 presents a framework to guide application of proof rules when several of them are applicable. Section 7 presents an example proof using our technique. Section 8 summarizes some applications of our proof technique along with experimental results. Finally, section 9 provides concluding remarks and comparisons to related work.

2 Encoding the Verification Problem

In this section, we discuss how to encode the problem of verifying parameterized concurrent systems as a logic program. Intuitively, a parameterized concurrent system can be viewed as a network of an unbounded number of finite state processes which communicate in a specific pattern. These finite state processes constituting the network have a finite number of process types, and their communication pattern is called the *network topology*. For example, an n bit shift register (for any n) is a parameterized system. It represents an unbounded number of finite state processes communicating along a chain. These finite state processes are "similar", each of them representing a single bit. To model a parameterized system as a logic program, the local states of the constituent finite state processes are represented by terms of finite size. The global state of the parameterized system is represented by a term of unbounded size consisting of these finite terms as sub-terms. The initial states and the transition relation of the parameterized system are then encoded as logic program predicates with such unbounded terms as arguments.

2.1 System Specification

In our encoding, the global states of the parameterized system to be verified are represented by unbounded terms. This is because (i) a global state is a composition of local states of the constituent processes and (ii) a parameterized system typically contains unbounded number of processes (recall that a parameterized system is an infinite family of finite state systems). The initial states and the transition relation of the parameterized system are specified as two logic program predicates `gen/1` and `trans/2` over these terms. Thus, for any ground term $\overline{\mathtt{t}}$, $\mathtt{gen}(\overline{\mathtt{t}})$ is true iff $\overline{\mathtt{t}}$ is an initial state of one member of the parameterized family being verified. Similarly for any two ground terms $\overline{\mathtt{t}}$, $\overline{\mathtt{t}}'$ we require that $\mathtt{trans}(\overline{\mathtt{t}}, \overline{\mathtt{t}}')$ is true iff $\overline{\mathtt{t}} \to \overline{\mathtt{t}}'$ is a transition in one of the members of the parameterized family. The recursive structure of `gen` and `trans` depends on the topology of the parameterized network being verified.

```
gen([1]).
gen([0|X]) :- gen(X).
trans([0,1|T], [1,0|T]).
trans([H|T], [H|T1]) :- trans(T,T1).
```

System description

```
thm(X) :- gen(X), live(X).
live(X) :- X = [1|_].
live(X) :- trans(X,Y), live(Y).
```

Property description

Fig. 1. Example: Liveness in an unbounded length shift register

For example, in an n bit shift register (for any n), the local states of the bit process are represented by the terms `0` and `1` (corresponding to the situations where the value stored in the bit is 0 and 1 respectively). A global state of the register is then represented by an unbounded list where each element of the list is `0` or `1`. Now, let us consider an n bit shift register where initially the rightmost bit of the chain contains 1 and all other bits contain 0. The system evolves by passing the 1 leftward. A logic program describing the system is given in Figure 1. The predicate `gen` generates the initial states of an n-process chain for all n. As mentioned above, a global state of the register is represented as an ordered list (a list in Prolog-like notation is of the form `[Head|Tail]`) of zeros and ones. The set of bindings of variable `S` upon evaluation of the query `gen(S)` is { `[1]`, `[0,1]`, `[0,0,1]`, ... }. The predicate `trans` in the program encodes a single transition of the global automaton. The first clause in the definition of `trans` captures the transfer of the 1 from right to left; the second clause recursively searches the state representation until the first clause can be applied. (i.e., when the 1 is not already in the left-most bit).

2.2 Temporal Property and Proof Obligations

So far, we have illustrated how the parameterized system to be verified can be encoded as a logic program. The temporal property to be verified can also be encoded as a logic program predicate over global states of the system. In this chapter, we only consider those properties φ *such that* φ *(or its negation) can be encoded as a definite logic program*. This includes weak liveness properties such as EFp and invariant properties such as AGp where p is an atomic proposition about system states and A, E, F, G are operators of the branching time temporal logic CTL [16]. These temporal properties have a single fixed point operator.

For our shift register example, we consider the CTL property EF [1|_], that is, eventually the 1 reaches the left most bit. This is encoded by the predicate `live` in Figure 1. The first clause of `live` succeeds for global states where the 1 is already in the left-most bit (a good state). The second (recursive) clause of `live` checks if a good state is reachable after a (finite) sequence of transitions. Every member of the parameterized family satisfies the liveness property if and only if $\forall$ `X gen(X)` $\Rightarrow$ `live(X)`. Moreover, this is the case if

$$\forall \mathtt{X}\ \mathtt{thm}(\mathtt{X}) \Leftrightarrow \mathtt{gen}(\mathtt{X})$$

i.e. if `thm` and `gen` are semantically equivalent. Thus, we have encoded the verification problem as a logic programs and reduced the proof obligation to establishing equivalence of program predicates.

3 Overview of our proof technique

We now illustrate how we can construct induction based proofs arising in parameterized system verification via logic program transformations. Essentially, this is accomplished using the following steps:

1. Encode the temporal property to be verified as well as the parameterized system as a logic program P_0.
2. Convert the verification proof obligation to predicate equivalence proof obligations of the form $P_0 \vdash \mathtt{p} \equiv \mathtt{q}$ ($\mathtt{p}, \mathtt{q}$ are predicates)
3. Construct a transformation sequence $P_0, P_1, \ldots, P_k$ s.t.
   (a) Semantics of P_0 = Semantics of P_k
   (b) from the syntax of P_k we infer $P_k \vdash \mathtt{p} \equiv \mathtt{q}$

The construction of a transformation sequence proceeds by repeated application of transformation rules. If several rules are applicable, strategies are used for rule selection. Inferring $P_k \vdash \mathtt{p} \equiv \mathtt{q}$ from syntax is achieved by a sufficient condition called *Syntactic Equivalence* which is formally defined in Section 5.1 (see Definition 4). Also, note that we are dealing with definite logic programs, and the semantics that we consider for a definite logic program is its least Herbrand model [10].

In the shift register example, we have encoded the problem of verifying liveness in an n bit shift register as the logic program P_0 in Figure 1. We have reduced the verification proof obligation to establishing the equivalence of `thm` and `gen` predicates in program P_0. We then apply program transformations to program P_0 to obtain a program P_k where `thm` and `gen` are defined as follows:

In this example, the transformed definitions of `thm` and `gen` are identical modulo predicate (and variable) renaming. In general, we have a sufficient condition called *syntactic equivalence* s.t. if two predicates `p` and `q` are syntactically equivalent in program P_k then `p` and `q` are semantically equivalent in P_k. Furthermore, we ensure that checking syntactic equivalence of two predicates in a

```
gen([1]).                 thm([1]).
gen([0|X]) :- gen(X).  thm([0|X]) :- thm(X).
```

Fig. 2. Fragment of Transformed Program for Shift Register Example

given program is decidable. In the shift register example, the transformed definitions of `gen` and `thm` given in Figure 2 are syntactically equivalent. The formal definition of syntactic equivalence is presented in Section 5.1. The definitions of `gen` and `thm` given above both represent the infinite set $\{[0^n, 1] \mid n \in \mathbb{N}\}$. For each element `X` in this set, we can therefore construct a *ground proof* of `thm(X)` and `gen(X)`. Formally, we define a ground proof as:

Definition 1 (Ground Proof) *Let T be a tree, each of whose nodes is labeled with a ground atom. Then T is a ground proof in a definite program P, if every node A in T satisfies the condition : A :– $A_1, ..., A_n$ is a ground instance of a clause in P, where $A_1, ..., A_n$ $(n \geq 0)$ are the children of A in T.*

For example, a ground proof tree[2] of `gen([0,0,1])` and `thm([0,0,1])` (using the above clauses of `thm` and `gen`) are shown below.

Inferring the equivalence of `thm` and `gen` from the transformed definitions in Figure 2 involves an *induction* on the size of the proof trees of `gen(X)` and `thm(X)` for any ground term `X`. In general, to prove the equivalence of two predicates p, p' of same arity we first transform their definitions to syntactically equivalent forms. Then, the proof of semantic equivalence of two syntactically equivalent predicates p, p' proceeds (by definition of syntactic equivalence) as follows:

– show that for every ground proof of $p(\overline{X})\theta$ (where $\overline{X}$ are variables and θ is any ground substitution of $\overline{X}$) there exists a ground proof of $p'(\overline{X})\theta$. This follows by induction on the size of ground proofs of $p(\overline{X})\theta$.
– show that for every ground proof of $p'(\overline{X})\theta$ (where $\overline{X}$ are variables and θ is any ground substitution of $\overline{X}$) there exists a ground proof of $p(\overline{X})\theta$. This follows by induction on the size of ground proofs of $p'(\overline{X})\theta$.

Thus, transforming `gen` and `thm` to obtain the definitions of Figure 2 and then inferring the equivalence from these transformed definitions amounts to an induction proof of the liveness property. Note that even though we are actually

[2] In this particular example, these are the only ground proofs of `gen([0,0,1])` and `thm([0,0,1])`.

inducting on the size of ground proofs, here this is same as inducting on the process structure of the parameterized system: the length of the shift register.

We now formally describe our proof technique. Since we always prove equivalence of logic program predicates, we start by constructing a proof system for predicate equivalence proof obligations. Formally, the *predicate equivalence problem* is: given a definite logic program P and a pair of predicates p and p' of the same arity, determine if $P \models p \equiv p'$ *i.e.* whether p and p' are semantically equivalent in P. In other words, we need to determine whether for all ground substitutions θ, $p(\overline{X})\theta \in M(P) \Leftrightarrow p'(\overline{X})\theta \in M(P)$. Here $M(P)$ denotes the least Herbrand model [30] of program P. *Henceforth, whenever we refer to the "semantics" of a definite logic program P, we mean its least Herbrand model $M(P)$.*

4 A Proof System for Predicate Equivalences

We develop a tableau-based proof system for establishing predicate equivalence. The proof system presented here can be straightforwardly extended to prove goal equivalences[3] instead of predicate equivalences. Our process is analogous to SLD resolution. Recall that given a goal $\mathcal{G}$ and a definite logic program P, SLD resolution is used to prove whether instances of $\mathcal{G}$ are in $M(P)$. This proof is constructed recursively by deriving new goals via resolution. The truth of $\mathcal{G}$ is then shown by establishing the truth of these new goals. In contrast, each node in our proof tree denotes a pair of predicates (p, p'). To establish their equivalence we must establish that the predicates in the pair represented by each child node are equivalent. Note that the predicates in the child node are to be obtained from the syntax of the current definitions of p, p'. We now define:

Definition 2 (e-atom) *Let $\Gamma = P_0, P_1, \ldots, P_i$ be a sequence of programs. An e-atom is of the form $\Gamma \vdash p \equiv p'$ where p and p' are predicates of same arity appearing in each of the programs in Γ. It represents the proof obligation*

$$\forall 0 \leq j \leq i \;\; P_j \models p \equiv p'$$

Thus, an e-atom $\Gamma \vdash p \equiv p'$ represents the proof obligation that p, p' are semantically equivalent in each of the programs in Γ. We generalize the problem of establishing a single e-atom to that of establishing a sequence of e-atoms. We define an *e-goal* as a (possibly empty) sequence of e-atoms. We will often denote an e-goal by $\mathcal{E}$, possibly with primes and subscripts. Recall that SLD resolution proves a goal by unfolding an atom in the goal. Similarly, we proceed to prove an e-goal by transforming the relevant clauses of an e-atom (*i.e.* the clauses of the predicates appearing in the e-atom) in the e-goal.

The three rules used to construct an equivalence tableau are shown in Table 1. In the description of the proof rules Γ denotes a sequence of programs $P_0, \ldots, P_i$. Given a definite logic program P_0, and a pair of predicates of same arity p, p', we construct a tableau for the proof obligation $P_0 \vdash p \equiv p'$ by repeatedly applying the inference rules in Table 1.

[3] Recall that in a definite logic program, a goal is a conjunction of atoms.

Name	Top-down Inference (one step)	Side Condition
(Ax)	$\dfrac{\mathcal{E},\quad \Gamma \vdash p \equiv p',\quad \mathcal{E}'}{\mathcal{E},\quad \mathcal{E}'}$	$p \overset{P_i}{\cong} p'$
(Tx)	$\dfrac{\mathcal{E},\quad \Gamma \vdash p \equiv p',\quad \mathcal{E}'}{\mathcal{E},\quad \Gamma, P_{i+1} \vdash p \equiv p',\quad \mathcal{E}'}$	$M(P_{i+1}) = M(P_i)$
(Gen)	$\dfrac{\mathcal{E},\quad \Gamma \vdash p \equiv p',\quad \mathcal{E}'}{\mathcal{E},\quad \Gamma, P_{i+1} \vdash p \equiv p',\quad P_0 \vdash q \equiv q',\quad \mathcal{E}'}$	$P_0 \models q \equiv q'$ $\Rightarrow M(P_{i+1}) = M(P_i)$

Table 1. Proof System for Showing Predicate Equivalences

The *axiom elimination rule* (**Ax**) is applicable whenever the equivalence of the predicates p and p' can be established by some automatic mechanism, denoted in the rule by $p \overset{P_i}{\cong} p'$. Thus, $\overset{P_i}{\cong}$ is a decision procedure which infers the equivalence of p, p' in program P_i. Axiom elimination will typically be an application of what we call **syntactic equivalence**, a decidable equivalence of predicates based on the syntactic form of the clauses defining them.

The *program transformation rule* (**Tx**) attempts to simplify the program in order to expose the equivalence of predicates (which can then be inferred via an application of **Ax**). The program P_{i+1} is constructed from Γ using a *semantics preserving* program transformation. We use this rule whenever we apply an unfolding, folding, or any other (semantics-preserving) transformation that does not add any equivalence proof obligations. We give a brief presentation of these transformation rules in the next section.

The *equivalence generation rule* (**Gen**) proves an e-atom $\Gamma \vdash p \equiv p'$ by performing replacements in the clauses of p, p'. In particular, occurrence of some predicate q in the clauses of p, p' is replaced by occurrence of another predicate q'. The guarantee is that if the predicates q, q' are semantically equivalent then the program thus obtained is semantics preserving. This appears as the side condition of the **Gen** rule. The notation $P_0 \models q \equiv q'$ is a shorthand for the following: for all ground substitution θ, $q(\overline{X})\theta) \in M(P_0) \Leftrightarrow q'(\overline{X})\theta) \in M(P_0)$ where $M(P_0)$ is the least Herbrand model of P_0. Note the proof of semantic equivalence of p and p' is being constructed by using the semantic equivalence of q and q'. This allows us to simulate nested induction proofs. Typically, an application of the **Gen** rule corresponds to applying the goal replacement transformation.[4]

The notion of a tableau for a predicate equivalence proof obligation in a definite logic program P_0 is then defined in the usual way.

[4] The **Gen** rule does not require $\{p, p'\} \cap \{q, q'\} = \phi$. When we synthesize an algorithmic framework for applying the proof rules we will keep track of the past history of equivalence proof obligations.

Definition 3 (Equivalence Tableau) *An* equivalence tableau *of an e-goal* $\mathcal{E}_0$ *is a finite sequence of e-goals* $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_n$ *where* $\mathcal{E}_{i+1}$ *is obtained from* $\mathcal{E}_i$ *by applying one of the rules described in Table 1 and* $\mathcal{E}_n$ *is empty.*

Now, let P_0 be a definite logic programs and p, p' be predicates of same arity appearing in P_0. Then we use our proof system to construct an equivalence tableau of $\mathcal{E}_0 = (P_0 \vdash p \equiv p')$.

Theorem 1 (Soundness of Proof System) *Let* $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_n$ *be a successful tableau with* $\mathcal{E}_0 = (P_0 \vdash p \equiv p')$ *for some definite logic program* P_0*. Then* $P_0 \models p \equiv p'$ *i.e. predicates* p *and* p' *are semantically equivalent in the least Herbrand model of* P_0*.*

Proof: We prove a stronger result. For any successful tableau of an e-goal $\mathcal{E}_0$ if $\Gamma \vdash p \equiv p'$ is an e-atom in $\mathcal{E}_0$ where $\Gamma = P_0, \ldots, P_i$ then $P_i \models p \equiv p'$.

The proof for this result is established by induction on the length of the tableau. For the base case, we have a tableau of length 1, which is formed by an application of the **Ax** rule. For such a tableau the result holds trivially since **Ax** is applied only when the semantic equivalence of p, p' can be automatically inferred in P_i. For the induction step, we consider a tableau $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_{k+1}$ of length $k+1$. For all e-atoms of $\mathcal{E}_0$ which are not modified in the step $\mathcal{E}_0 \to \mathcal{E}_1$, the result follows by induction hypothesis. Let $P_0, \ldots, P_i \vdash p \equiv p'$ be the e-atom in $\mathcal{E}_0$ that is modified.

- **Ax**: If the rule applied to $\mathcal{E}_0$ is **Ax**, then from the side condition of **Ax** we have $P_i \models p \equiv p'$.
- **Tx** : If the rule applied to $\mathcal{E}_0$ is **Tx**, then $P_0, \ldots, P_i, P_{i+1} \vdash p \equiv p'$ is an e-atom in $\mathcal{E}_1$. Since $\mathcal{E}_1, \ldots, \mathcal{E}_{k+1}$ is a successful tableau of $\mathcal{E}_1$, therefore by induction hypothesis $P_{i+1} \models p \equiv p'$. By the side condition of **Tx**, we have $M(P_i) = M(P_{i+1})$ and therefore $P_i \models p \equiv p'$.
- **Gen** : If the rule applied to $\mathcal{E}_0$ is **Gen**, then $P_0, \ldots, P_i, P_{i+1} \vdash p \equiv p'$ and $P_0 \vdash q \equiv q'$ are e-atoms in $\mathcal{E}_1$. Again $\mathcal{E}_1, \ldots, \mathcal{E}_{k+1}$ is a successful tableau of $\mathcal{E}_1$. By induction hypothesis, we have $P_{i+1} \models p \equiv p'$ and $P_0 \models q \equiv q'$. From the side condition of **Gen** we have $M(P_i) = M(P_{i+1})$ and therefore $P_i \models p \equiv p'$.

If $\mathcal{E}_0, \ldots, \mathcal{E}_n$ is a successful tableau of $\mathcal{E}_0 = P_0 \ \vdash \ p \equiv p'$ then $P_0 \models p \equiv p'$. $\square$

The tableau can be readily extended to use some transformations that may not preserve least models, but only ensure that the least models, with respect to the predicates in the original program, are same. A transformation that adds new predicates to a program has this property, and is often useful in predicate equivalence proofs. From the soundness of the proof system, we can also infer the following property for any equivalence tableau: for any e-atom $\Gamma \vdash \ldots$ appearing in an equivalence tableau, all programs in Γ are semantically equivalent. The proof appears in [41].

Lemma 2 *Let $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_n$ be an equivalence tableau of $\mathcal{E}_0 = P_0 \vdash p \equiv p'$. For every e-atom $(\Gamma \vdash \ldots)$ in the tableau, if $\Gamma = P_0, \ldots, P_i$ then we have $M(P_0) = \ldots = M(P_i)$.*

Note that the proof system given in Table 1 is not complete. There can be no such complete proof system as attested to by the following theorem.

Theorem 3 (Incompleteness) *Determining equivalence of predicates described by logic programs is not recursively enumerable.*

The theorem is easily proved using a reduction described in [3]. For a Turing machine M, we construct a program having two predicates, one that describes the natural numbers and the other that identifies an n such that M does not halt within n moves. These predicates are equivalent if and only if M does not halt. The non-halting problem is not recursively enumerable and so the predicate equivalence problem cannot be recursively enumerable.

5 Automated Instances of Proof Rules

In this section, we discuss the automation of each application of an **Ax**, **Tx** or **Gen** rule. The application of the **Tx** and **Gen** rules is achieved by unfolding, folding and goal replacement transformations (which we also discuss).

5.1 Automating the Ax Rule

The *axiom elimination rule* (**Ax**) infers the equivalence of two predicates p, p' in a semantics preserving program transformation sequence $\Gamma = P_0, \ldots, P_i$. In the light of Theorem 3, any such rule will be incomplete. Therefore, we will construct an effectively checkable sufficient condition for predicate equivalence. We call this sufficient condition as syntactic equivalence. Given a program transformation sequence $\Gamma = P_0, \ldots, P_i$ and two predicates p, p', we apply **Ax** if p, p' are syntactically equivalent in program P_i.

As an illustration, consider the program P (with clauses annotated with integer clause measures) in Figure 3. We can infer that $P \models \mathtt{r} \equiv \mathtt{s}$ since `r` and `s` have identical definitions. Using the equivalence of `r` and `s` we can infer that $P \models \mathtt{p} \equiv \mathtt{q}$, since the definitions of `p` and `q` are, in a sense, isomorphic.

```
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).
```

Fig. 3. Program with syntactically equivalent predicates.

We formalize this notion of equivalence in the following definition. The following definition partitions the predicate symbols of a program into equivalence classes. Each predicate is assumed to be assigned a *label*, the partition number of the equivalence class to which it belongs. The labels of all predicates belonging to the same equivalence class is thus the same, and each equivalence class has a unique label.

Definition 4 (Syntactic Equivalence) *A syntactic equivalence relation* $\overset{P}{\sim}$, *is an equivalence relation on the set of predicates of a program* P *such that for all predicates* p, q *in* P, *if* $p \overset{P}{\sim} q$ *then the following conditions hold:*

1. p *and* q *have same arity, and*
2. *Let the clauses defining* p *and* q *be* $\{C_1, \ldots, C_m\}$ *and* $\{D_1, \ldots, D_n\}$ *respectively. Let* $\{C'_1, \ldots, C'_m\}$ *and* $\{D'_1, \ldots, D'_n\}$ *be such that* C'_l (D'_l) *is obtained by replacing every predicate symbol* r *in* C_l (D_l) *by* s, *where* s *is the label of the equivalence class of* r *(w.r.t.* $\overset{P}{\sim}$*). Then there exist two functions* $f : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$ *and* $g : \{1, \ldots, n\} \rightarrow \{1, \ldots, m\}$ *such that*
   (i) $\forall 1 \leq i \leq m$ C'_i *is an instance of* $D'_{f(i)}$
   (ii) $\forall 1 \leq j \leq n$ D'_j *is an instance of* $C'_{g(j)}$.

Note that there is a largest syntactic equivalence relation. It can be computed by starting with all predicates in the same class, and repeatedly splitting the classes that violate properties (1) and (2) until a fixed point is reached. The existence of the mapping f ensures that for any ground substitution θ we have $p(\overline{X})\theta \in M(P) \Rightarrow q(\overline{X})\theta \in M(P)$ whereas the mapping g ensures $q(\overline{X})\theta \in M(P) \Rightarrow p(\overline{X})\theta \in M(P)$. The proof of the lemma proceeds by induction on size of ground proofs (see [41] for details).

Lemma 4 (Syntactic Equivalence $\Rightarrow$ Semantic Equivalence) *Let* $\overset{P}{\sim}$ *be the syntactic equivalence relation of the predicates of a program* P. *For all predicates* p, q, *if* $p \overset{P}{\sim} q$, *then* $P \models p \equiv q$.

5.2 Automating Tx: Transformations as Proof Rules

The transformation rule **Tx** corresponds to applying a program transformation which does not add any new equivalence proof obligations. Typically an application of this step is either unfolding or folding, or other standard transformations like generalization and equality introduction, deletion of subsumed clauses and deletion of failed clauses [35]. A single application of all these transformations can be automated.

We transform a logic program to another logic program by applying transformations that include unfolding and folding. A simple illustration of these transformations appears in Figure 4. Program P_1 is obtained from P_0 by unfolding the occurrence of `q(X)` in the definition of `p`. P_2 is obtained by folding `q(X)` in the second clause of `p` in P_1 using the definition of `p` in P_0 (an earlier program). Intuitively, unfolding is a step of clause resolution whereas folding replaces an instance of a clause body (in some earlier program in the transformation sequence) with its head.

```
p(X) :- q(X).
q(0).
q(s(X)) :- q(X).
```
Program P_0

$\xrightarrow{Unf.}$

```
p(0).
p(s(X)) :- q(X).
q(0).
q(s(X)) :- q(X).
```
Program P_1

$\xrightarrow{Fold}$

```
p(0).
p(s(X)) :- p(X).
q(0).
q(s(X)) :- q(X).
```
Program P_2

Fig. 4. Illustration of unfold/fold transformations

In the following, we present a (simplified) version of the unfolding and folding transformation rules. Note that each application of these rules is automated. We say that $P_0, P_1, \ldots, P_n$ is an *unfold/fold transformation sequence* if the program P_{i+1} is obtained from P_i $(i \geq 0)$ by an application of unfolding or a folding. We always assume that the clauses of each program in such a transformation sequence are "standardized apart". In other words, the variables of the clauses of P_i are suitably renamed such that no two clauses have any variables in common.

Transformation 1 Unfolding *Let C be a clause in P_i and A an atom in the body of C. Let $C_1, \ldots, C_m$ be the clauses in P_i whose heads are unifiable with A with most general unifier $\sigma_1, \ldots, \sigma_m$. Let C'_j be the clause that is obtained by replacing $A\sigma_j$ by the body of $C_j\sigma_j$ in $C\sigma_j$ $(1 \leq j \leq m)$. Assign $(P_i - \{C\}) \cup \{C'_1, \ldots, C'_m\}$ to P_{i+1}.* □

Transformation 2 Folding *Let $\{C_1, \ldots, C_m\} \subseteq P_i$ where C_l denotes the clause*

$$A :- A_{l,1}, \ldots, A_{l,n_l}, A'_1, \ldots, A'_n$$

and $\{D_1, \ldots, D_m\} \subseteq P_j$ $(j \leq i)$ where D_l is $B_l :- B_{l,1}, \ldots, B_{l,n_l}$. Further, let:

1. $\forall 1 \leq l \leq m\ \exists \sigma_l\ \forall 1 \leq k \leq n_l\ A_{l,k} = B_{l,k}\sigma_l$
2. $B_1\sigma_1 = B_2\sigma_2 = \cdots = B_m\sigma_m = B$
3. *$D_1, \ldots, D_m$ 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[5] of D_l to distinct variables which do not appear in $\{A, B, A'_1, \ldots A'_n\}$.*

Then $P_{i+1} := (P_i - \{C_1, \ldots, C_m\}) \cup \{C'\}$ where $C' \equiv A :- B, A'_1, \ldots, A'_n$. □

$D_1, \ldots, D_m$ are the *folder* clauses, $C_1, \ldots, C_m$ are the *folded* clauses, and B is the *folder* atom.

Semantics preservation While unfolding is semantics preserving, folding may introduce circularity and change the program semantics. Recall that we are dealing with definite logic programs and we consider the least Herbrand model semantics. For example consider the program P_1 in Figure 5; P_1 is obtained from P_0 by unfolding the occurrence of `q(X)` in the body of `p`'s second clause. We perform folding where the second clause of `p` in P_1 serves as the folded clause

[5] Variables appearing in the body of a clause, but not its head

and the second clause of `q` in P_0 serves as the folder clause. We get the program P_2 of Figure 5. Now, let us fold again. We use the second clause of `q` in P_2 as the folded clause and the second clause of `p` in P_1 as the folder clause. This produces the program P_3 of Figure 5. The program transformation sequence $P_0 \to P_1 \to P_2 \to P_3$ is not semantics preserving since the least Herbrand model of P_3 differs from that of P_0.

Program P_0	Program P_1	Program P_2	Program P_3

`p(X):-q(X).`
`q(a).`
`q(f(X)):-q(X).`
	`p(a).`
	`p(f(X)):-q(X).`
	`q(a).`
	`q(f(X)):-q(X).`
		`p(a).`
		`p(f(X)):-q(f(X)).`
		`q(a).`
		`q(f(X)):-q(X).`
			`p(a).`
			`p(f(X)):-q(f(X)).`
			`q(a).`
			`q(f(X)):-p(f(X)).`

Fig. 5. An example of incorrect unfold/fold transformation sequence

Due to this problem of semantics preservation, existing unfold/fold transformation systems have restricted the folding rule. Thus, in a program transformation sequence $P_0, P_1, \ldots, P_i$, folding of clause(s) in P_i is restricted [20, 26, 46, 47]. The restrictions are of two kinds: (a) based on the unfold/fold steps used to derive the transformation sequence $P_0, \ldots, P_i$, and (b) based on the syntax of the folder clauses used. In [43] we have shown that restrictions on the syntax of folder clauses is unnecessary for semantics preservation. As a consequence of this result, in a folding step we can use multiple clauses as folder; furthermore some of these clauses may be recursive.

The additional power of our transformation rules is useful in our transformation based proofs of temporal properties. Note that temporal properties contain fixed point operators. These properties are typically encoded as a logic program predicate with multiple recursive clauses *e.g.* a least fixed point property containing disjunctions is encoded using multiple recursive clauses. A simple reachability property EFp (which specifies that a state in which proposition p holds is reachable) [9] will be encoded as a logic program as follows:

```
ef(X) :- p(X).
ef(X) :- trans(X,Y), ef(Y).
```

where the predicate `trans` captures the transition relation of the system being verified, and `p(X)` is true if the proposition p holds in state `X`. This encoding contains two clauses, one of which is recursive.

5.3 Automating the Gen Rule

The **Gen** rule attempts to prove the e-atom $\Gamma \vdash p \equiv p'$ by proving the e-atoms $\Gamma, P_{i+1} \vdash p \equiv p'$ and $P_0 \vdash q \equiv q'$ where $\Gamma = P_0, \ldots, P_i$ is a program transformation sequence. It generates a new lemma $P_0 \vdash q \equiv q'$ whose proof is used to ensure that $M(P_i) = M(P_{i+1})$. An application of **Gen** corresponds to an application of the Goal Replacement transformation (given in the following). Here, we replace an occurrence of q with q' in a clause of p or p' as shown below.

$$\begin{array}{ll} \ldots & \ldots \\ C : p(\bar{t}) :- \mathcal{G},\, q(\bar{s}),\, \mathcal{G}'. & C' : p(\bar{t}) :- \mathcal{G},\, q'(\bar{s}),\, \mathcal{G}'. \\ \ldots & \ldots \\ \\ \text{Program } P_i & \text{Program } P_{i+1} \end{array}$$

This requires us to show $P_0 \models q \equiv q'$ and therefore we obtain a new proof obligation $P_0 \vdash q \equiv q'$. We prove $P_0 \vdash q \equiv q'$ by constructing a different transformation sequence $P_0, P_1', \ldots, P_k'$ s.t. $q \overset{P_k'}{\sim} q'$ *i.e.* q, q' are syntactically equivalent in P_k'. Note that since we are replacing q with q' in program P_i, the goal replacement rule requires $P_i \models q \equiv q'$. However for any e-atom $\Gamma \vdash \ldots$ appearing in a successful tableau, $M(P_0) = \ldots = M(P_i)$ where $\Gamma = P_0, \ldots, P_i$ (refer Lemma 2). Thus, $P_0 \models q \equiv q'$ implies $P_i \models q \equiv q'$.

A (simplified) definition of the Goal Replacement Transformation is given below. Again, to ensure semantics preservation, the transformation rule needs to impose *additional restrictions* on the transformation sequence $P_0, P_1, \ldots, P_i$. We omit these restrictions here (refer [43] for details). For a conjunction of atoms $A_1, ..., A_n$, we use the notation $vars(A_1, ..., A_n)$ to denote the set of variables in $A_1, ..., A_n$.

Transformation 3 Goal Replacement Let C be a clause $A :- A_1, \ldots, A_k, G$ in P_i, and G' be an atom such that $vars(G) = vars(G') \subseteq vars(A, A_1, ..., A_k)$. Suppose for all ground instantiation θ of G, G' we have $G\theta \in M(P_i) \Leftrightarrow G'\theta \in M(P_i)$. Then $P_{i+1} := (P_i - \{C\}) \cup \{C'\}$ where $C' \equiv A :- A_1, \ldots, A_k, G'$. □

6 An Algorithmic Framework for Proof Strategies

We describe an algorithmic framework for creating strategies to automate the construction of the equivalence tableau of an e-atom. The objective is to: (a) find equivalence proofs that arise in verification with little or no user intervention, and (b) apply deduction rules lazily, i.e. for finite state systems a proof using the strategy is equivalent to algorithmic verification.

Our framework specifies the order in which the different program transformations (corresponding to each tableau rule) will be applied. If multiple transformations of the same kind (e.g., two folding steps) are possible at any point in the proof, the framework itself does not specify which transformations to apply. That is done by a separate selection function (analogous to literal selection in SLD resolution). *Thus we only present a framework for constructing strategies, rather than concrete strategies. Concrete strategies can be constructed by instantiating this framework.*

The tableau rules and associated transformations are applied in the following order. As would be expected, the axiom elimination rule (**Ax**) is used whenever it is applicable. When the choice is between the **Tx** and **Gen** rules, we choose the former since the default transformation employed by **Tx** is unfolding, *i.e.*

```
p1(a).
p1(f(X)):- p1(X),s1(X).
p1(f(X)):- p1(X),t1(X),q1(X).
r1(X):- s1(X).
r1(X):- t1(X),q2(X).
```

$\boldsymbol{P_0}$

```
p1(a).
p1(f(X)):- p1(X),s1(X).
p1(f(X)):- p1(X),t1(X), q2(X).
r1(X):- s1(X).
r1(X):- t1(X),q2(X).
```

$\boldsymbol{P_i}$

```
p1(a).
p1(f(X)):- p1(X),r1(X).
r1(X):- s1(X).
r1(X):- t1(X),q2(X).
```

$\boldsymbol{P_{i+1}}$

Fig. 6. Goal replacements to facilitate other transformations.

resolution. This will ensure that our strategies will perform on-the-fly model checking, a' la XMC [40] for finite-state systems. To create finite unfolding sequences we impose a finiteness condition *FIN* on transformation sequences. We do not give an exact definition of *FIN* but only a sufficient condition such that the resultant unfolding sequences terminate.

Definition 5 (Finiteness condition) *Given an a-priori fixed constant $k \in \mathbb{N}$, an unfolding program transformation sequence $\Gamma = P_0, \ldots, P_i, \ldots$ satisfies the finiteness condition $FIN(\Gamma, k)$ if for the clause C and atom A selected for unfolding at every P_i: (1) A is distinct modulo variable renaming from any atom B which was selected in unfolding some clause $D \in P_j (j < i)$ where C is obtained by repeated unfolding of D (2) the term depth of each argument of A is $\leq k$.*

Typically, we will assume a suitable choice of k and write the finiteness condition simply as $FIN(\Gamma)$. Condition 1 prohibits infinite unfolding sequences of the form: unfolding `p(X)` using the clause `p(X) :- p(X)` *i.e.* unfolding sequences where the same atom is infinitely unfolded. Condition 2 prohibits infinite unfolding sequences of the form: unfolding `p(X)` using the clause `p(X) :- p(s(X))` *i.e.* where a different atom is unfolded every time, but there are infinitely many atoms to unfold.

We note that various online techniques for ensuring termination of unfolding sequences have been studied in the context of partial deduction [28, 32]. These techniques proceed by establishing a well-founded / well-quasi order among the atoms unfolded. This order may be fixed before hand, or refined online as the unfolding proceeds. Such techniques could be adapted for controlling unfolding in our predicate equivalence prover.

If *FIN* prohibits any further unfolding we either apply the folding transformation associated with **Tx** or use the **Gen** rule. Care must be taken, however, when **Gen** is chosen. Recall from the definition of **Gen** (refer Table 1) that $\Gamma, P_{i+1} \vdash p \equiv p'$ implies $\Gamma \vdash p \equiv p'$ only if we can prove a new equivalence $P_0 \vdash q \equiv q'$. In other words, $P_{i+1} \models p \equiv p'$ implies $P_i \models p \equiv p'$ only if

$P_0 \models q \equiv q'$. Since **Gen** itself does not specify the goals q and q' in the new equivalence, its application is highly nondeterministic. We limit the nondeterminism by using **Gen** only to enable **Ax** or **Tx** rules. For instance, consider the transformation sequence in Figure 6 (the intermediate programs $P_1, \ldots, P_{i-1}$ in the program transformation sequence $P_0, P_1, \ldots, P_{i-1}, P_i$ are not shown). Applying goal replacement in P_0 under the assumption that $P_0 \models \texttt{q1} \equiv \texttt{q2}$ enables the subsequent folding which transforms P_i into P_{i+1}.

Thus, when no further unfoldings are possible, we apply any possible folding. If no foldings are enabled, we check if there are new goal equivalences that will enable a folding step. We call this a *conditional folding* step. For instance, in program P_0 of Figure 6, equivalence of `q1(X)` and `q2(X)` enables folding. Note that the test for syntactic equivalence is only done on predicates, whereas a goal is a conjunction of atoms. However, we can reduce a goal equivalence check to a predicate equivalence check by introducing new predicate names for the goals. A keen point needs to be noted here. When we introduce new predicate names to a program, clearly the least Herbrand model cannot be preserved. As is common in program transformation literature [46, 20], we rectify this apparent anomaly by assuming that all new predicate names introduced are present in the initial program P_0 of a program transformation sequence.

Finally, we look for new goal equivalences, which, if valid, can lead to syntactic equivalence. This is called a *conditional equivalence* step. For instance, suppose in program P_{i+1} (in Figure 6), there are two additional predicates `p2` and `r2` and further assume that `p2` is defined using clauses

```
p2(a).
p2(f(Y)):- p2(Y), r2(Y).
```

Now if `r2` and `r1` are semantically equivalent, we can perform this goal replacement to obtain a program P_{i+2} where `p1` and `p2` are defined as follows. Thus, in P_{i+2} we can conclude that $\texttt{p1} \overset{P_{i+2}}{\sim} \texttt{p2}$.

```
p1(a).                          p2(a).
p1(f(X)):- p1(X), r1(X).        p2(f(Y)):- p2(Y), r1(Y).
```

The above intuitions are formalized in algorithmic framework *Prove* (see Figure 7). Given a program transformation sequence Γ, and a pair of predicates p, p', *Prove* attempts to prove that $\Gamma \vdash p \equiv p'$. *Prove* searches nondeterministically for a proof: if multiple cases of the nondeterministic choice are enabled, then they will be tried in the order specified in *Prove*. If none of the cases apply, then evaluation fails, and backtracks to the most recent unexplored case. There may also be nondeterminism within a case; for instance, many fold transformations may be applicable at the same time. We again select nondeterministically from this set of applicable transformations. By providing selection functions to pick from these applicable transformations, one can implement a variety of concrete strategies. Note that *Prove* uses two different markings in the process of constructing a proof for $\Gamma \vdash p \equiv p'$. The marking *proved* remembers predicate equivalences which have been already proved. This marking allows us to cache subproofs in a proof. The marking *proof_attempt* keeps track of predicate pairs whose equivalence has not yet been established, but is being attempted by *Prove* via transformations. This marking is essential for ensuring termination of *Prove*. The proof of $P_0 \vdash p \equiv p'$ may (via a conditional equivalence step) generate the (sub)-equivalence $P_0 \vdash p \equiv p'$. *Prove* deems this proof path as failed and explores other proof paths.

Our algorithmic framework *Prove* uses the following functions. Functions *unfold*(P), *fold*(P) apply unfolding and folding transformations respectively to program P and return a new program. Whenever conditional folding is possible, the function *new_goal_equiv_for_fold*(P) finds a pair of goals whose replacement is necessary to do a fold transformation. Similarly, when conditional equivalence is possible, *new_goal_equiv_for_equiv*(p, p', P) finds a pair of goals $\mathcal{G}, \mathcal{G}'$ s.t. syntactic equivalence of p and p' can be established after replacing $\mathcal{G}$ with $\mathcal{G}'$ in P.

Finally, *replace_and_prove* constructs nested proofs for sub-equivalences created by applying the **Gen** rule. Thus, *replace_and_prove*$(p, p', \mathcal{G}, \mathcal{G}', \Gamma)$ performs the following sequence of steps (where $\Gamma = P_0, \ldots, P_i$):

1. first introduces new predicate definitions q and q' for goals $\mathcal{G}$ and $\mathcal{G}'$ respectively (if such definitions do not already exist),
2. proves the equivalence $P_0 \vdash q \equiv q'$ by invoking *Prove*,
3. replaces goal $\mathcal{G}$ by goal $\mathcal{G}'$ in clauses of p or p' in program P_i to obtain program P_{i+1}, and
4. finally invokes *Prove* to dispense the obligation $\Gamma, P_{i+1} \vdash p \equiv p'$. This completes the proof of $\Gamma \vdash p \equiv p'$.

Termination of ***Prove*** It can be verified that only finite unfolding sequences satisfy *FIN*. This is because in any unfolding sequence of clauses $C_1, \ldots, C_n$ where C_{i+1} is obtained from C_i via unfolding, condition 1 of Definition 5 ensures that the selected atom each C_i is distinct, and condition 2 ensures that there are only finitely many atoms which can ever be selected for unfolding.

Therefore, the length of each predicate equivalence proof itself is finite (assuming folding always reduces program size which can be ensured). However, a proof for $p \equiv p'$ may require $q \equiv q'$ as a lemma, whose proof in turn may require $r \equiv r'$ as a lemma, and so on. Since the number of distinct equivalences are quadratic in the number of predicate symbols in the program, the number of subproofs is finite if the number of new predicates names introduced is finite. Thus, we have :

Lemma 5 *Prove (refer Figure 7) terminates provided the number of definitions introduced (*i.e. *new predicate symbols added) is finite.*

Efficiency of unfolding/folding The algorithmic framework *Prove* does not clarify how we implement $unfold(P)$ and $fold(P)$, *i.e.* the heuristics for choosing unfolding/folding steps. These heursitics are extremely important for the purposes of efficient proof construction via program transformation. Full details of these heuristics appear in [41, 44].

algorithm $\mathit{Prove}(p, p'$: predicates, Γ:prog. seq.)
begin
  if $\mathit{proof\_attempt}(p, p')$ is marked **then return** false
  mark $\mathit{proof\_attempt}(p, p')$
  let $\Gamma = P_0, \ldots, P_i$
  (* **Ax** rule *)
  if $(p \stackrel{P_i}{\sim} p' \vee \mathit{proved}(p, p'))$ **then**
    return true
  else nondeterministic choice
    (* **Tx** rule *)
      case $\mathit{FIN}(\langle \Gamma, \mathit{unfold}(P_i) \rangle)$: (* Unfolding *)
        return $\mathit{Prove}(p, p', \langle \Gamma, \mathit{unfold}(P_i) \rangle)$
      case Folding is possible in P_i:
        return $\mathit{Prove}(p, p', \langle \Gamma, \mathit{fold}(P_i) \rangle)$
    (* **Gen** rule *)
      case Conditional folding is possible in P_i:
        let $(\mathcal{G}, \mathcal{G}') = \mathit{new\_goal\_equiv\_for\_fold}(P_i)$
        return $\mathit{replace\_and\_prove}(p, p', \mathcal{G}, \mathcal{G}', \Gamma)$
      case Conditional equivalence is possible in P_i:
        let $(\mathcal{G}, \mathcal{G}') = \mathit{new\_goal\_equiv\_for\_equiv}(p, p', P_i)$
        return $\mathit{replace\_and\_prove}(p, p', \mathcal{G}, \mathcal{G}', \Gamma)$
  end choices
  mark $\mathit{proved}(p, p')$
  unmark $\mathit{proof\_attempt}(p, p')$
end

Fig. 7. Algorithmic framework for equivalence tableau construction.

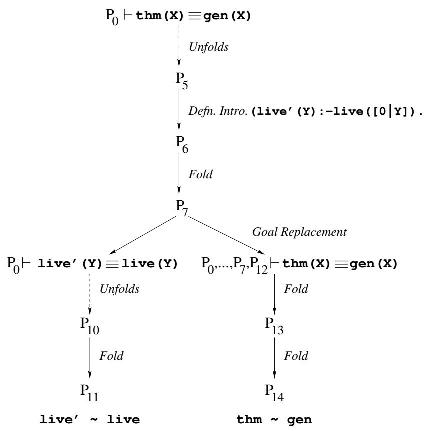

Fig. 8. Liveness Proof of n-bit shift register

Proving Predicate Implications Note that the proof system given in Table 1, the algorithmic framework *Prove* and the strategies to guide the transformations in *Prove* are aimed at proving equivalence of program predicates. Our proof technique can be readily extended to prove predicate implications *i.e.* proof obligations of the form

$$\forall \text{ ground substitutions } \theta \;\; p(\overline{X})\theta \in M(P_0) \Rightarrow p'(\overline{X})\theta \in M(P_0)$$

This extension involves (1) relaxing the definition of syntactic equivalence (Definition 4) to test for implications only, and (2) generating conditions of the form $q \Rightarrow q'$ by applying conditional folding and conditional equivalence.

7 An Example Proof

Recall the logic program of Figure 1 (page 6) which formulates a liveness property about token-passing chains, namely, that the token eventually reaches the left-most process in any arbitrarily long chain. We obtain P_0, the starting point of our transformation sequence from the encoding of the verification problem in Figure 1. To establish the liveness property, we prove that $P_0 \vdash$ `thm(X)` $\equiv$ `gen(X)`, by invoking *Prove*(`thm`, `gen`, $\langle P_0 \rangle$). The proof is illustrated in Figure 8.

Proof of $P_0 \vdash$ `thm` $\equiv$ `gen`: Since `thm` $\stackrel{P_0}{\not\sim}$ `gen`, we must transform the predicates. By repeatedly unfolding the definition of `thm` in P_0, we obtain program P_5 where `thm` is defined as:

```
thm([1]).
thm([0|X]) :- gen(X), X = [1|_].
thm([0|X]) :- gen(X), trans(X,Y), live([0|Y]).
```

Further unfolding in P_5 is not possible since it involves unfolding an atom which is already unfolded in the sequence $P_0, \ldots, P_5$, thereby risking non-termination. In addition no folding transformation is applicable at this stage. However, if $\forall$`Y live([0|Y])` $\Leftrightarrow$ `live(Y)` we can fold the last two clauses of `thm`. Thus, *conditional folding* is true at P_5, and hence *replace_and_prove* is invoked with $\mathcal{G}$ = `live([0|Y])` and $\mathcal{G}'$ = `live(Y)`. Since `live([0|Y])` is not an open atom, a new name:

```
live'(Y) :- live([0|Y]).
```

is added to P_5 to yield P_6. This simply converts the goal equivalence problem of showing $\forall$`Y live([0|Y])` $\Leftrightarrow$ `live(Y)` to a predicate equivalence problem. We fold the third clause of `thm` above using the newly introduced clause as folder, obtaining P_7:

```
thm([1]).
thm([0|X]) :- gen(X), X = [1|_].
thm([0|X]) :- gen(X), trans(X,Y), live'(Y).
```

We then proceed to show $P_0 \vdash$ `live'` $\equiv$ `live`. This subproof is shown in the left branch of the tree in Figure 8). Then we replace `live'(X)` with `live(X)` in the definition of `thm` in P_7 (right branch in Figure 8).

Proof of $P_0 \vdash$ `live'` $\equiv$ `live`: *Prove*(`live'`, `live`, $\langle P_0 \rangle$) performs a series of unfoldings, yielding programs P_8, P_9 and P_{10}. Any further unfolding involves unfolding an atom already unfolded in the sequence P_0, P_8, P_9, P_{10} and risks non-termination. In P_{10}, `live'` is defined by the following clauses:

```
live'([1|Z]).
live'(X) :- trans(X,Z), live([0|Z]).
```

Folding is applicable is P_{10}, in the second clause of `live'`, yielding P_{11} with

```
live'([1|Z]).
live'(X) :- trans(X,Z), live'(Z).
```

Now, `live'` $\overset{P_{11}}{\sim}$ `live` and hence *Prove*(`live'`, `live`, $\langle P_0 \rangle$) terminates. We assume that all occurrences of the equality predicate in the clause bodies are removed (via unification) prior to any syntactic equivalence check.

Resuming Proof of $P_0 \vdash$ `thm` $\equiv$ `gen`: Now *replace_and_prove* replaces `live'(X)` with `live(X)` in the definition of `thm` in P_7, yielding P_{12} with:

```
thm([1]).
thm([0|X]) :- gen(X), X = [1|_].
thm([0|X]) :- gen(X), trans(X,Y), live(Y).
```

We can now fold the last two clauses of `thm` using the definition of `live` in P_0. Note that the folding uses a recursive definition of a predicate with multiple clauses. The program-transformation system developed by us in [43] was the first to permit such folding. Thus we obtain P_{13}:

```
thm([1]).
thm([0|X]) :- gen(X), live(X).
```

This completes the conditional folding step (which had invoked *replace_and_prove* and thereby constructed `live'` $\equiv$ `live` as a subproof). We can fold again using the definition of `thm` in P_0, giving P_{14} where `thm` is defined as:

```
thm([1]).
thm([0|X]) :- thm(X).
```

We now have `thm` $\overset{P_{14}}{\sim}$ `gen`, thereby completing the equivalence proof.

It is interesting to observe in Figure 8 that the unfolding steps that transform P_0 to P_5 and P_7 to P_{10} are interleaved with folding steps. In other words, algorithmic and deductive verification steps are interleaved in the proof of the equivalence $P_0 \vdash$ `thm` $\equiv$ `gen`.

8 Experiments

So far, we have presented a tableau based proof system for proving equivalence of predicates in a logic program. Furthermore, we presented an algorithmic framework *Prove* for guiding the application of the rules in the proof system. However, this algorithmic framework *Prove* is nondeterministic since at each step several transformations may be applicable. Hence it is necessary to develop appropriate selection functions to distill concrete strategies from the algorithmic framework. Indeed we have implemented such strategies in a predicate equivalence prover for verifying parameterized protocols of different *network topologies* (the communication pattern between the different constituent processes of a parameterized network is called its network topology). Given a parameterized system and a liveness/invariant property to be proved, our prover extracts the predicate equivalences that need to be established. It tries to use the network topology of the parameterized system being verified to construct concrete proof strategies. These strategies then guide the proof search which proceeds without any user intervention. The proof search is terminating, sound but incomplete (*i.e.* the prover may fail to establish a correct property). *A full-fledged discussion of the concrete proof strategies (obtained by instantiating the algorithmic framework Prove of the last section) appears in [41, 44].*

In this section, we present the experimental results obtained using our predicate equivalence prover. The prover is built on top of the XSB tabled logic programming system [50] which supports top-down memoized evaluation of logic programs. We report results on parameterized cache coherence protocols, including (a) single bus broadcast protocols *e.g.* Mesi, (b) single bus protocols with global conditions *e.g.* Illinois and (b) multiple bus hierarchical protocols. We also report experimental results for the Java Meta-locking algorithm [1], a distributed algorithm to ensure secure access of shared objects by various Java threads. The benchmarks cover various network topologies including star, tree and complete graph networks.

Results In Table 2, *Meta-lock* denotes the Java meta-locking algorithm from Sun Microsystems. The Java Meta-Locking Algorithm is a distributed algorithm recently proposed by Sun Microsystems to ensure mutually exclusive access of shared Java objects by Java threads. A proof of correctness of the algorithm involves proving mutual exclusion in the access of a Java object by arbitrary number of Java threads. Previously, model checking has been used to verify mutual exclusion for different instances of the protocol, obtained by fixing the number of threads [5]. We have used our program transformation based prover to automatically construct a proof of mutual exclusion for the entire infinite family. The sources of infiniteness in the Meta-locking algorithm include (a) unbounded number of Java threads, and (b) data variables of infinite domain in the shared Java object.

Mesi and *Berkeley RISC* are single bus broadcast protocols [4, 15, 18]. *Illinois* is a single bus cache coherence protocol with global conditions which cannot be modeled as a broadcast protocol [12]. *Tree-cache* is a binary tree network which simulates the interactions between the cache agents in a hierarchical cache coherence protocol [41].

Table 2 presents experimental results obtained using our prover: a summary of the invariants proved along with the time taken, the number of unfolding steps and the number of deductive steps (*i.e.* folding, conditional equivalence

Protocol	Invariant	Time(sec) in [12]	Our time(secs)	# Unf	#Ded
Meta-Lock	#owner + #handout < 2	-	129.8	1981	311
Mesi	#m + #e < 2	1	3.2	325	69
	#m + #e = 0 ∨ #s = 0	0.5	2.9	308	63
Illinois	#dirty < 2	5.3	35.7	2501	137
Berkeley	#dirty < 2	0.6	6.8	503	146
	#ex + #sh < 2	-	Fails	-	-
Tree-cache	#bus_with_data < 2	-	9.9	178	18

Table 2. Summary of protocol verification results

etc.) performed in constructing the proof. The total time involves time taken by (a) unfolding steps (b) deductive steps, and (c) the time to invoke nested proof obligations. All experiments reported here were conducted on a Sun Ultra-Enterprise workstation with two 336 MHz CPUs and 2 GB of RAM. In the table, we have used the following notational shorthand: $\#s$ denotes the number of processes in local state s. In column 3 of the table, we have shown the timings for the same proofs using the constraint logic program evaluation based checker of [12]. The work of [12] was aimed at verifying parameterized cache coherence protocols. Note that the timings of [12] were obtained using a Pentium 133 with Linux 2.0.32.

Comparison with CLP-based Verifier [12] The running times of our prover are slower than the times for verifying single bus cache coherence protocols reported in [12]. In fact, there is up to an order of magnitude difference between the time taken by our prover and the time taken by the CLP-based verifier. One source of the relative inefficiency of our prover arises from the way the proof steps are applied. The prototype implementation of our prover implements both the unfolding and folding steps via meta-programming. While meta-programming appears inevitable to perform the folding steps, the unfolding steps can be performed directly at the level of the underlying evaluation engine. Such an implementation would improve running times, can be directly compared to the CLP-based verifier, and can be used to evaluate whether the overheads due to folding steps in our prover exceed the overheads due to constraint solving in the CLP-based verifier. Also, the abstraction technique of [12] is not suitable for parameterized tree networks such as *Tree-cache*, which can be verified by our inductive proof technique.

Comparison across Benchmarks Note that the number of deductive steps in a proof is consistently small compared to the number of unfolding steps. This is owing to our proof search strategy which repeatedly applies unfolding steps until none are applicable. Furthermore, note that the tree network example consumes larger running time with fewer unfolding and deductive steps as compared to other cache coherence protocols like the Mesi protocol. Due to its network topology, the state representation in the tree network has a different term structure than the other protocols (where the global states are typically represented as lists). This partially accounts for the increase in the running time. In addition, certain deductive steps (such as conditional equivalence) employ more expensive search heuristics for the tree topology. Finally, the Java meta-locking algorithm represents global states as lists, but involves nested induction over both control and the data of the protocol thereby increasing the number of predicate equivalence proof obligations. Extra proof obligations are incurred due to nested induction on the infinite data domain thereby increasing the time to construct the proof.

9 Discussion

In this chapter, we have presented a technique for proving predicate equivalences in a definite logic program. This is used for verifying infinite-state concurrent systems, in particular the class of parameterized concurrent systems. We have described how the parameterized system verification problem can be reduced to proving equivalence of logic program predicates. First we review related work on using logic program transformations to construct proofs.

9.1 Related Work

Unfold/fold transformations of logic programs have been widely used for program specialization and optimization. Relatively little work has been done on using these transformations for constructing *proofs*. Unfold/fold transformations can be used to construct induction proofs of program properties. In such proofs, unfolding accomplishes the base case and the finite part of the induction step, and folding roughly corresponds to application of induction hypothesis. This observation has been exploited in [23, 25, 27, 36, 37] to construct inductive proofs of program properties.

Hsiang and Srivas in [23] extended Prolog's evaluation with "limited forward chaining" to perform inductive theorem proving. This limited forward chaining step is in fact a very restricted form of folding: only the theorem statement (which is restricted to be conjunctive) can be used as a folder clause. The works of [25, 27] is closer to ours. They proved certain first order theorems about the Least Herbrand Model of a definite logic program via induction. In particular, they observed that the least fixed point semantics of logic programs could be exploited to employ fixed point induction. Our usage of the transformations is similar. Given a program P we intend to prove $p \equiv q$ in the Least Herbrand Model of P. To do this proof by induction, we transform p and q to obtain a program P'. If the transformed definitions of p and q in P' are "syntactically equivalent" (Definition 4) then our proof is finished. Note that *the syntactic equivalence check is in fact an application of fixed point induction.* It allows us to show $p \equiv q$ in $M(P')$ (the least Herbrand model of P'). Furthermore, since $M(P') = M(P)$ this amounts to showing $p \equiv q$ in program P. Thus, in our work predicates are transformed to facilitate the construction of induction schemes (for proving predicate equivalence). [25] also exploits transformations for similar purposes. However, their method performs *conjunctive* folding using only a single non-recursive clause. Apart from the restriction in their folding rule, they also do not employ goal replacement in their induction proofs.

The idea of using logic program transformations for proving goal equivalences was explored by Pettorossi and Proietti in [36, 37]. These works employ more restricted unfold/fold transformation rules *e.g.* folding using non-recursive clauses. In [19], a proof technique based on transformation of constraint logic programs was proposed. It is used to verify safety properties of systems with arbitrary number of (potentially infinite state) process. Unlike our work, the proof technique of [19] is not based on mathematical induction. Instead it produces uniform proofs by abstracting away the number of processes. The work of [12] proves safety properties of parameterized systems via evaluation of constraint logic programs (the evaluation includes acceleration techniques to ensure termination). Partial deduction and abstract interpretation of logic programs is used in [29] for proving safety properties of infinite state systems; these techniques can be applied to parameterized families as well.

The reader might notice similarities between a proof system based on unfold/fold transformations and a proof system based on tabled resolution [7, 48]. Tabled resolution combines resolution proofs with memoing of calls and answers. Since folding corresponds to remembering the original definition of predicates, there is some correspondence between folding and memoing. However, folding can remember conjunctions and/or disjunctions of atoms as the definition of a predicate. This is not possible in tabled resolution. Furthermore, in tabled resolution when a tabled call C is encountered, the answers produced so far for C are used to produce new answers for C. In folding, when the clause bodies in old definition of a predicate is encountered, it is replaced by the clause head.

We note that there is a lot of research work on using logic program transformations for optimization and/or partial evaluation [35]. Furthermore, the area of automated inductive theorem proving has substantial literature of its own [6]. These works are not discussed here. Instead, we have concentrated only on techniques which extend logic program evaluation for proving program properties.

9.2 Summary

In a broader perspective, our proof technique is geared to automate nested induction proofs, where each induction proceeds without hypothesis strengthening. Furthermore, the induction schema as well as the requisite lemmas should be implicitly encoded in the logic program itself. We have employed our lightweight inductive proof technique for verifying a specific class of infinite state concurrent systems: *parameterized systems.* Such systems occur widely in computing since many distributed algorithms in telecommunication and information processing applications constitute a parameterized concurrent system. We have used our proof technique to verify parameterized networks of various interconnection patterns: chain, ring, tree, star and complete graph networks. A prover based on our technique has been used to verify design invariants of real-life distributed algorithms such as the recently developed Java meta-locking algorithm from Sun Microsystems [1].

Our program transformation based proof technique unifies algorithmic and deductive verification steps (*i.e.* model checking and theorem proving steps) in a framework. Essentially the proof technique amounts to integrating limited deductive steps by enhancing the search based evaluation of a model checker. This is different from the traditional way of integrating model checking and theorem proving where a model checker is incorporated as a decision procedure into a theorem prover [39].

The reader should however note that unlike model checking, our inductive proof technique currently *does not generate counter-example evidence*. Due to the undecidability of parameterized system verification, the problem of counter-example generation is more involved. This is because a proof attempt may fail due to either the temporal property being false or the inability of the proof system to construct a proof. The problem of navigating and explaining proof attempts has been studied for interactive theorem provers [24]. For our transformation based automated prover, we can a-posteriori provide explanation of success/failure of a proof attempt. In particular, we can provide to the user the tree of predicate equivalence proof obligations constructed. Once a node in this tree is selected (by the user), we can provide snapshots of the transformation sequence constructed which led to the success/failure of the proof attempt of that predicate equivalence. Developing tools and techniques for explaining transformation based proof runs is an attractive topic of future work.

In conclusion, we would like highlight some interesting aspects of our proposed integration (of algorithmic and deductive verification). First, the proof technique thus obtained allows arbitrary interleaving of algorithmic and deductive steps in a proof. In contrast, by incorporating model checking as a decision procedure into a theorem prover, the model checker is always invoked as a subroutine. Secondly, the integration is not only *tight* but also *extensible* for verification of different flavors of concurrent systems. Our transformation based proof technique is a flexible extension of model checking via logic program evaluation (since one of our transformations correspond to logic program evaluation). By extending the underlying programming language to constraint logic programs one can verify families of timed systems with similar proof techniques. Finally, note that the proof technique supports zero overhead theorem proving [45]. Concurrent systems which can be verified without deductive reasoning (such as finite state and data independent systems) are verified via model checking since the deductive transformations are applied lazily.

Acknowledgments The work reported in this chapter was conducted as part of the first author's Ph.D. thesis at SUNY Stony Brook [41]. Many discussions with K. Narayan Kumar, I.V. Ramakrishnan and Scott A. Smolka shaped the work.

This work was partially supported by NSF grants CCR-9711386, CCR-9876242, CDA-9805735 and EIA-9705998.

References

1. O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999.
2. R. Alur and T. A. Henzinger, editors. *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, July 1996. Springer-Verlag.
3. K. Apt and D. Kozen. Limits for automatic verification of finite-state systems. *Information Processing Letters*, 15:307–309, 1986.
4. J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.
5. S. Basu, S.A. Smolka, and O.R. Ward. Model checking the Java meta-locking algorithm. In *IEEE International Conference on the Engineering of Computer Based Systems*. IEEE Press, April 2000.
6. A. Bundy. *The Automation of Proof by Mathematical Induction*, volume 1 of *Handbook of Automated Reasoning*, pages 845–911. Elsevier and MIT Press, 2001.
7. W. Chen and D.S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.
8. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2), 1986.
9. E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. S.K. Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.
11. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.
12. G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *International Conference on Computer Aided Verification (CAV), LNCS 1855*, 2000.
13. G. Delzanno and A. Podelski. Model checking in CLP. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume LNCS 1579, pages 74–88. Springer-Verlag, 1999.
14. E. Emerson and K.S. Namjoshi. Reasoning about rings. In *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*, pages 85–94, 1995.
15. E. Emerson and K.S. Namjoshi. On model checking for non-deterministic infinite state systems. In *IEEE Annual Symposium on Logic in Computer Science (LICS)*, pages 70–80, 1998.
16. E.A. Emerson. *Temporal and Modal Logic*, volume B of *Handbook of Theoretical Computer Science*, pages 995–1072. Elsevier/North-Holland, 1990.
17. E.A. Emerson and K.S. Namjoshi. Automated verification of parameterized synchronous systems. In Alur and Henzinger [2].

18. J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *IEEE Annual Symposium on Logic in Computer Science (LICS)*, pages 352–359, 1999.
19. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Logic-Based Program Synthesis and Transformation (LOPSTR '01)*, LNCS 2372, 2002.
20. M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In *International Symposium on Programming Language Implementation and Logic Programming (PLILP), LNCS 844*, pages 340–354, 1994.
21. S. German and A. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39:675–735, 1992.
22. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
23. J. Hsiang and M. Srivas. Automatic inductive theorem proving using Prolog. *Theoretical Computer Science*, 54:3–28, 1987.
24. INRIA Rocquencourt, URL : `http://pauillac.inria.fr/coq/doc/main.html`, Paris, France. *The Coq Proof Assistant : Reference Manual*, 1999.
25. T. Kanamori and H. Fujita. Formulation of Induction Formulas in Verification of Prolog Programs. In *International Conference on Automated Deduction (CADE)*, pages 281–299, 1986.
26. T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. In *USA-Japan Seminar on Logics of Programs*, 1987.
27. T. Kanamori and H. Seki. Verification of prolog programs using an extension of execution. In *International Conference on Logic Programming (ICLP)*, 1986.
28. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.
29. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In Annalisa Bossi, editor, *Logic-Based Program Synthesis and Transformation (LOPSTR '99)*, LNCS 1817, pages 63–82, Venice, Italy, 2000.
30. J.W. Lloyd. *Foundations of Logic Programming, Second Edition*. Springer-Verlag, 1993.
31. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1991.
32. B. Martens, D. De Schreye, and T. Horváth. Sound and complete partial deduction with unfolding based on well-founded measures. *Theoretical Computer Science*, 122:97–117, 1994.
33. K.S. Namjoshi. *Ameliorating the State Explosion Problem*. PhD thesis, University of Texas at Austin, 1998.
34. U. Nilsson and J. Lubcke. Constraint logic programming for local and symbolic model checking. In *Computational Logic, LNCS 1861*, 2000.
35. 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.
36. A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming*, 41(2–3):197–230, 1999.
37. A. Pettorossi and M. Proietti. Perfect model checking via unfold/fold transformations. In *Computational Logic, LNCS 1861*, 2000.

38. A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. In *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*, pages 414–427, 1997.
39. S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *International Conference on Computer Aided Verification (CAV), LNCS 939*, 1995.
40. Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.
41. A. Roychoudhury. *Program Transformations for Verifying Parameterized Systems.* PhD thesis, State University of New York at Stony Brook, Available from `http://www.comp.nus.edu.sg/~abhik/papers.html`, 2000.
42. A. Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Verification of parameterized systems using logic program transformations. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume LNCS 1785, pages 172–187. Springer-Verlag, 2000.
43. A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, To appear. Preliminary version appeared in International Conference on Principles and Practice of Declarative Programming (PPDP) 1999, LNCS 1702.
44. A. Roychoudhury and I.V. Ramakrishnan. Inductively verifying invariant properties of parameterized systems. *Automated Software Engineering Journal*, 2004. Preliminary version appeared in International Conference on Computer Aided Verification (CAV) 2001, LNCS 2102.
45. Carl Seger. Combining functional programming and hardware verification. In *ACM SIGPLAN International Conference on International Conference on Functional Programming, Invited Talk*, 2000.
46. H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of International Conference on Logic Programming*, pages 127–138, 1984.
47. H. Tamaki and T. Sato. A generalized correctness proof of the unfold/ fold logic program transformation. Technical report, Ibaraki University, Japan, 1986.
48. H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third International Conference on Logic Programming*, pages 84–98, 1986.
49. L. Urbina. Analysis of hybrid systems in CLP(R). In *Constraint Programming (CP'96)*, volume LNCS 1102. Springer-Verlag, 1996.
50. XSB. The XSB logic programming system v2.2, 2000. Available for downloading from `http://xsb.sourceforge.net/`.