Unfold/fold Transformations for
Automated Verification of
Parameterized Concurrent Systems

Abhik Roychoudhury\textsuperscript{1} and C.R. Ramakrishnan\textsuperscript{2}

\textsuperscript{1} School of Computing, National University of Singapore, Singapore. 
abhik@comp.nus.edu.sg

\textsuperscript{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 only finite state systems. 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. An algorithmic framework is 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 program. These include hardware controllers, operating systems, network protocols, and distributed applications e.g. air traffic control system. Intuitively, a reactive concurrent program 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 program 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 programs 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. A proof system is essentially a collection of proof rules corresponding to the operators of the languages chosen for expressing the specification and the implementation. 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 [26].

However, for finite state concurrent programs, 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 represented by the concurrent program. 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 concurrent 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. $\mathbb{N}$. 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 for only finite instances of the algorithm, i.e. for \( n = 3, n = 4, \ldots \), but never the entire algorithm for any \( n \). 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 (or even possible) to adapt them to verify parameterized systems.

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 [18,27]. Using this approach meaningful subclasses have been identified, such as token rings of similar processes [12] and classes of parameterized synchronous systems [15].

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 [44] and more recently for model checking of finite-state [28] / infinite-state systems [11]. In [35], 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 [19]. Recently, [10] used constraint logic programming to construct uniform 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 are in general not suitable for constructing induction proofs arising in the verification of parameterized systems. Essentially, we construct an automatic and programmable first order logic prover with limited deductive capability. Strategies can be programmed to guide the application of the proof rules. We discuss related work on other proof techniques based on program transformations in Section 9.

Our work provides a methodology for constructing such proofs by suitably extending the resolution based evaluation mechanism of logic programs [36,38]. 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 prob-
lem of determining the equivalence of predicates in this program. The predicate equivalences are then established by transforming them s.t. 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 $p(X)$ and $p'(X)$.

For transforming the predicates, we use the well-established unfold/fold transformations of logic programs which have been previously used for program optimization [24,32] and automated deduction [20,21,30]. 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.

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 discusses strategies 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 then 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.

\[
gen([1]).
gen([0;X]) :- \text{gen}(X).\]
\[
\text{thm}(X) :- \text{gen}(X), \text{live}(X).\]
\[
\text{trans}([0.1]; [1.0;T]).
\text{live}(X) :- X = [1].\]
\[
\text{trans}([1;T], [1;T]) :- \text{trans}(T;T1).
\text{live}(X) :- \text{trans}(X,Y), \text{live}(Y).\]

System description
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 \(\text{gen}\) 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 \([\text{Head}[\text{Tall}]\) of zeros and ones. The set of bindings of variable \(S\) upon evaluation of the query \(\text{gen}(S)\) is \([1], [0,1], [0,0,1], \ldots\). The predicate \(\text{trans}\) in the program encodes a single transition of the global automaton. The first clause in the definition of \(\text{trans}\) 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).

**Temporal Property** So far, we have illustrated how the parameterized system to be verified can be encoded using logic program predicates. 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 \(\varphi\) such that \(\varphi\) (or its negation) can be encoded as a definite logic program. This includes liveness properties such as \(\text{EFp}\) and invariant properties such as \(\text{AGp}\) where \(p\) is an atomic proposition about system states and \(A,E,F,G\) are operators of the branching time temporal logic CTL [14]. For our shift register example, we consider the following liveness property: eventually the 1 reaches the left most bit. This is encoded by the predicate \(\text{live}\) in Figure 1. The first clause of \(\text{live}\) succeeds for global states where the 1 is already in the left-most bit (a good state). The second (recursive) clause of \(\text{live}\) checks if a good state is reachable after a (finite) sequence of transitions.
Proof Obligation  Every member of the parameterized family satisfies the liveness property if and only if \( \forall X \, \text{gen}(X) \Rightarrow \text{live}(X) \). Moreover, this is the case if \( \forall X \, \text{thm}(X) \Leftrightarrow \text{gen}(X) \), i.e. if \( \text{thm} \) and \( \text{gen} \) are semantically equivalent. Thus, we have encoded the verification problem as a logic program 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 \( P_0 \vdash p \equiv q \) (\( p, q \) are predicates)  
3. Construct a transformation sequence \( P_0, P_1, \ldots, P_h \) s.t.
   (a) Semantics of \( P_0 \) = Semantics of \( P_h \)
   (b) from the syntax of \( P_h \) we infer \( P_h \vdash p \equiv q \)

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 \( \text{thm} \) and \( \text{gen} \) predicates in program \( P_0 \). We then apply program transformations to program \( P_0 \) to obtain a program \( P_h \) where \( \text{thm} \) and \( \text{gen} \) are defined as follows:

\[
\begin{align*}
\text{gen}([1]), & \quad \text{thm}([1]), \\
\text{gen}([0|X]) & \iff \text{gen}(X). \quad \text{thm}([0|X]) \iff \text{thm}(X).
\end{align*}
\]

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

Thus, since the transformed definitions of \( \text{thm} \) and \( \text{gen} \) are “isomorphic”, their semantic equivalence can be inferred from syntax. In general, we have a sufficient condition called syntactic equivalence s.t. if two predicates \( p \) and \( q \) are syntactically equivalent in program \( P_h \) then \( p \) and \( q \) are semantically equivalent in \( P_h \). Furthermore, we ensure that checking syntactic equivalence of two predicates in a given program is decidable. In the shift register example, the transformed definitions of \( \text{gen} \) and \( \text{thm} \) given in Figure 2 are syntactically equivalent. The formal definition of syntactic equivalence is presented in Section 5. The definitions of \( \text{gen} \) and \( \text{thm} \) 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 \( \text{thm}(X) \) and \( \text{gen}(X) \). 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 of $\text{gen}([0,0,1])$ and $\text{thm}([0,0,1])$ (using the above clauses of $\text{thm}$ and $\text{gen}$) are shown below.

$$
\begin{array}{c}
\text{gen}([0,0,1]) \\
\text{gen}([0,1]) \\
\text{gen}([1]) \\
\end{array}
\quad
\begin{array}{c}
\text{thm}([0,0,1]) \\
\text{thm}([0,1]) \\
\text{thm}([1]) \\
\end{array}
$$

Inferring the equivalence of $\text{thm}$ and $\text{gen}$ from the transformed definitions in Figure 2 involves an induction on the size of the proof trees of $\text{gen}(X)$ and $\text{thm}(X)$ 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 $p(X) \theta$ (where $X$ are variables and $\theta$ is any ground substitution of $X$) there exists a ground proof of $p'(X) \theta$. This follows by induction on the size of ground proofs of $p(X) \theta$.
- show that for every ground proof of $p'(X) \theta$ (where $X$ are variables and $\theta$ is any ground substitution of $X$) there exists a ground proof of $p(X) \theta$. This follows by induction on the size of ground proofs of $p'(X) \theta$.

Thus, transforming $\text{gen}$ and $\text{thm}$ 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 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.

4 A Proof System for Predicate Equivalences

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

---

1 In this particular example, these are the only ground proofs of $\text{gen}([0,0,1])$ and $\text{thm}([0,0,1])$. 
need to determine whether for all ground substitutions \( \theta \), \( p(\overline{x}) \theta \in M(P) \iff p'(\overline{x}) \theta \in M(P) \). Here \( M(P) \) denotes the least Herbrand model \([23]\) of program \( P \). Henceforth, whenever we refer to the “semantics” of a definite logic program \( P \), we mean its least Herbrand model \( M(P) \).

We develop a tableau-based proof system for establishing predicate equivalence. The proof system presented here can be straightforwardly extended to prove goal equivalences\(^2\) instead of predicate equivalences. Our process is analogous to SLD resolution. Recall that given a goal \( G \) and a definite logic program \( P \), SLD resolution is used to prove whether instances of \( G \) are in \( M(P) \). This proof is constructed recursively by deriving new goals via resolution. The truth of \( G \) 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 \( \Gamma \). It represents the proof obligation \( \forall 0 \leq j \leq i \ P_j \vdash p \equiv p' \) i.e., \( p,p' \) are semantically equivalent in each of the programs in \( \Gamma \).

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 \( \Gamma \) 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.

The axiom elimination rule \((\text{Ax})\) is applicable whenever the equivalence of the predicates \( p \) and \( p' \) can be established by some automatic mechanism, denoted in the rule by \( p \overset{\text{Ax}}{\equiv} p' \). Thus, \( \overset{\text{Ax}}{\equiv} \) 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 \((\text{Tx})\) attempts to simplify the program in order to expose the equivalence of predicates (which can then be inferred via an application of \text{Ax}). The program \( P_{t+1} \) is constructed from \( \Gamma \) using a semantics preserving program transformation. We use this rule whenever we

\(^2\) Recall that in a definite logic program, a goal is a conjunction of atoms.
<table>
<thead>
<tr>
<th>Name</th>
<th>Top-down Inference (one step)</th>
<th>Side Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Ax)</td>
<td>$\xi, \Gamma \vdash p \equiv p', \xi'$</td>
<td>$p \Leftrightarrow p'$</td>
</tr>
<tr>
<td>(Tx)</td>
<td>$\xi, \Gamma \vdash p \equiv p', \xi'$</td>
<td>$\text{M}(P_{i+1}) = \text{M}(P_i)$</td>
</tr>
<tr>
<td>(Gen)</td>
<td>$\xi, \Gamma, P_{i+1} \vdash p \equiv p', \xi'$</td>
<td>$P_0 \vdash q \equiv q', \xi' \Rightarrow \text{M}(P_{i+1}) = \text{M}(P_i)$</td>
</tr>
</tbody>
</table>

Table 1. Proof System for showing Predicate equivalences

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 $P_0 \vdash q \equiv q'$ is a shorthand for the following: for all ground substitution $\theta$, $q(X)\theta \in \text{M}(P_0) \Leftrightarrow q'(X)\theta \in \text{M}(P_0)$ where $\text{M}(P_0)$ 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.\(^3\)

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.

**Definition 3 (Equivalence Tableau)** An equivalence tableau of an e-goal $\xi_0$ is a finite sequence of e-goals $\xi_0, \xi_1, \ldots, \xi_n$ where $\xi_{i+1}$ is obtained from $\xi_i$ by applying one of the rules described in Table 1 and $\xi_n$ 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 $\xi_0 = (P_0 \vdash p \equiv p')$.

**Theorem 1 (Soundness of Proof System)** Let $\xi_0, \xi_1, \ldots, \xi_n$ be a successful tableau with $\xi_0 = (P_0 \vdash p \equiv p')$ for some (definite) logic program $P_0$. Then $P_0 \vdash p \equiv p'$ i.e. predicates $p$ and $p'$ are semantically equivalent in the least Herbrand model of $P_0$.

\(^3\) The Gen rule does not require $\{p, p'\} \cap \{q, q'\} = \emptyset$. When we synthesize an algorithmic framework for applying the proof rules we will keep track of the past history of equivalence proof obligations.
Proof: We prove a stronger result. For any successful tableau of an e-goal $E_0$ if $\Gamma \vdash p \equiv p'$ is an e-atom in $E_0$ where $\Gamma = P_0, \ldots, P_i$ then $P_i \vdash p \equiv p'$.

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 $E_0, E_1, \ldots, E_{k+1}$ of length $k + 1$. For all e-atoms of $E_0$ which are not modified in the step $E_0 \rightarrow E_1$, the result follows by induction hypothesis. Let $P_0, \ldots, P_i \vdash p \equiv p'$ be the e-atom in $E_0$ that is modified.

- Ax: If the rule applied to $E_0$ is Ax, then from the side condition of Ax we have $P_i \vdash p \equiv p'$.
- Tx: If the rule applied to $E_0$ is Tx, then $P_0, \ldots, P_i, P_{i+1} \vdash p \equiv p'$ is an e-atom in $E_i$. Since $E_i, \ldots, E_{k+1}$ is a successful tableau of $E_i$, therefore by induction hypothesis $P_{i+1} \vdash p \equiv p'$. By the side condition of Tx, we have $M(P_i) = M(P_{i+1})$ and therefore $P_i \vdash p \equiv p'$.
- Gen: If the rule applied to $E_0$ 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 $E_i$. Again $E_i, \ldots, E_{k+1}$ is a successful tableau of $E_i$. By induction hypothesis, we have $P_{i+1} \vdash p \equiv p'$ and $P_0 \vdash q \equiv q'$. From the side condition of Gen we have $M(P_i) = M(P_{i+1})$ and therefore $P_i \vdash p \equiv p'$.

If $E_0, \ldots, E_i$ is a successful tableau of $E_0 = P_0 \vdash p \equiv p'$ then $P_0 \vdash p \equiv p'$. \hfill \Box

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. It shows that for any e-atom $\Gamma \vdash \ldots$ appearing in an equivalence tableau, all programs in $\Gamma$ are semantically equivalent. The proof appears in [39].

**Lemma 2** Let $E_0, E_1, \ldots, E_n$ be an equivalence tableau of $E_0 = P_0 \vdash p \equiv p'$. 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. In the next section we present an algorithmic framework for guiding the application of these rules. 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 $P \models r \equiv s$ since $r$ and $s$ have identical definitions. Using the equivalence of $r$ and $s$ we can infer that $P \models p \equiv q$, since the definitions of $p$ and $q$ are, in a sense, isomorphic.

\[
\begin{align*}
    p(X) &:= r(X). & q(X) &:= s(X). \\
    p(X) &:= e(X, Y), p(Y). & q(X) &:= e(X, Y), q(Y). \\
    r(X) &:= b(X). & s(X) &:= b(X).
\end{align*}
\]

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 $\equiv$, is an equivalence relation on the set of predicates of a program $P$ such that for all predicates $p, q$ in $P$, if $p \equiv q$ 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 $C'_i$ ($D'_i$) is obtained by replacing every predicate symbol $r$ in $C_i$ ($D_i$) by $s$, where $s$ is the label of the equivalence class of $r$ (w.r.t. $\equiv$). Then there exist two functions
\[ f: \{1, \ldots, m\} \rightarrow \{1, \ldots, n\} \] and 
\[ g: \{1, \ldots, n\} \rightarrow \{1, \ldots, m\} \text{ such that} \]

(i) \( \forall 1 \leq i \leq m \, C_i \text{ is an instance of } D_{f(i)} \)

(ii) \( \forall 1 \leq j \leq n \, D_j \text{ is an instance of } C_{g(j)} \).

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 \( \theta \) we have 
\( p(X)\theta \in M(P) \Rightarrow q(X)\theta \in M(P) \) whereas the mapping \( g \) ensures 
\( q(X)\theta \in M(P) \Rightarrow p(X)\theta \in M(P) \). The proof of the lemma proceeds by induction on size 
of ground proofs (see [39] for details).

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

### 5.2 Automating T\( x \): Transformations as Proof Rules

The transformation rule T\( x \) 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 [29]. 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 un- 
folding 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 instance of clause bodies (in some earlier program in the transformation 
sequence) with its head.

\[
\begin{align*}
\text{p(X)} & := \overline{\text{q(X)}}. \\
\text{q(0)} & := \overline{\text{q(0)}}. \\
\text{q(s(X))} & := \overline{\text{q(X)}}. \\
\text{Program } P_0 & \\
\text{p(0)} & := \overline{\text{q(0)}}. \\
\text{p(s(X)))} & := \overline{\text{q(X)}}. \\
\text{q(0):} & := \overline{\text{q(0)}}. \\
\text{q(s(X)))} & := \overline{\text{q(X)}}. \\
\text{Program } P_1 & \\
\text{p(0):} & := \overline{\text{q(0)}}. \\
\text{p(s(X)))} & := \overline{\text{q(X)}}. \\
\text{q(0):} & := \overline{\text{q(0)}}. \\
\text{q(s(X)))} & := \overline{\text{q(X)}}. \\
\text{Program } P_2 & \\
\end{align*}
\]

**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.
Transformation 5 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 6 Folding Let $\{C_1, \ldots, C_m\} \subseteq P_i$ where $C_i$ denotes the clause $A := A_{i,1}, \ldots, A_{i,n}, A', \ldots, A'_n$ and $\{D_1, \ldots, D_m\} \subseteq P_j$ ($j \leq i$) where $D_i$ is $B_i := B_{i,1}, \ldots, B_{i,n}$. Further, let:

1. $\forall l \leq m \exists \sigma_i \forall k \leq n_i A_{i,k} = B_{i,k} \sigma_i$
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. $\forall l \leq m, \sigma_l$ substitutes the internal variable$^4$ of $D_l$ to distinct variables which do not appear in $\{A, B, A', \ldots, A'_n\}$.

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

$D_1, \ldots, D_m$ 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 $P'_0$ in Figure 4; $P'_0$ 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 $P'_0$ serves as the folded clause and the second clause of $q$ in $P'_0$ serves as the folder clause. We get the program $P'_3$ of Figure 5. Now, let us fold again. We use the second clause of $q$ in $P'_3$ as the folded clause and the second clause of $p$ in $P'_0$ as the folder clause. This produces the program $P'_3$ of Figure 5. The program transformation sequence $P_0 \rightarrow P'_0 \rightarrow P'_2 \rightarrow P'_3$ is not semantics preserving since the least Herbrand model of $P'_3$ differs from that of $P'_0$.

\[
\begin{align*}
\text{Program } P_0 & \quad \text{Program } P'_0 & \quad \text{Program } P'_2 & \quad \text{Program } P'_3 \\
p(X) &::=q(X). & \quad p(a). & \quad p(a). & \quad p(a). \\
q(a). & \quad p(f(X))::=q(X). & \quad p(a). & \quad p(f(X))::=q(X). & \quad p(f(X))::=q(f(X)). \\
q(f(X))::=q(X). & \quad q(a). & \quad q(a). & \quad q(a). & \quad q(a).
\end{align*}
\]

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

$^4$ Variables appearing in the body of a clause, but not its head.
Due to this problem of semantics preservation, existing unfold/fold transformation systems have restricted the folding rule. Thus, in a program transformation sequence $R_0, R_1, \ldots, R_i$, folding of clause(s) in $R_i$ is restricted [41, 17, 22]. The restrictions are of two kinds: (a) based on the unfold/fold steps used to derive the transformation history $R_0, \ldots, R_i$, and (b) based on the syntax of the folder clauses used. In [37] we have shown that restrictions on the syntax of folder clauses is unnecessary for semantics preservation in unfold/fold transformation sequences of definite logic programs. 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:

\[
edf(X) := \neg p(X).
\]

\[
edf(X) := \text{trans}(X,Y), \ edf(Y).
\]

where the predicate \text{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. Therefore, one cannot assume restrictions that are imposed by existing transformation systems [17, 22, 41, 42] on the syntax of clauses encoding a temporal property.

5.3 Automating the Gen Rule

The \textit{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 \textit{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.

\[
\ldots
C : p(\overline{E}) \leftarrow \mathcal{G}, q(\overline{x}), \mathcal{G}'.
\ldots
\]

\[
\ldots
C' : p(\overline{E}) \leftarrow \mathcal{G}, q'(\overline{x}), \mathcal{G}'.
\ldots
\]

Program $P_i$ Program $P_{i+1}$

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. $q \equiv q'$ 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 needs to impose additional restrictions on the transformation history. We omit these restrictions here (refer [37] for details). For a conjunction of atoms \( A_1, \ldots, A_n \), we use the notation \( \text{vars}(A_1, \ldots, A_n) \) to denote the set of variables in \( A_1, \ldots, A_n \).

**Transformation 7 (Goal Replacement)** Let \( C \) be a clause \( A \leftarrow A_1, \ldots, A_k, G \) in \( P_i \), and \( G' \) be an atom such that \( \text{vars}(G) = \text{vars}(G') \subseteq \text{vars}(A, A_1, \ldots, A_k) \).

Suppose for all ground instantiation \( \theta \) 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 \( C' \equiv A \leftarrow A_1, \ldots, A_k, G' \).

\[\square\]

### 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).

\[\text{p1(a).} \]
\[\text{p1(}f(x)\text{):- p1(x), s1(x).} \]
\[\text{p1(}f(x)\text{):- p1(x), t1(x), q1(x).} \]
\[\text{r1(x):- s1(x).} \]
\[\text{r1(x):- t1(x), q2(x).} \]

**Fig. 6.** Goal replacements to facilitate other transformations.

The tableau rules and associated transformations are applied in the following order. As would be expected, the axiom elimination rule (\( A\text{x} \)) is used whenever it is applicable. When the choice is between the \( \text{Tx} \) and \( \text{Gen} \) rules, we choose the former since the default transformation employed by \( \text{Tx} \) is unfolding, i.e., resolution. This will ensure that our strategies will perform on-the-fly
model checking, a’ la XMC [35] for finite-state systems. For infinite-state systems, however, uncontrolled unfolding may diverge. To create finite unfolding sequences we impose a finiteness condition\textit{FIN} on transformation sequences. We do not give an exact implementation of \textit{FIN} but only a sufficient condition s.t. the resultant unfolding sequences terminate.

**Definition 8 (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 \( \text{FIN}(\Gamma, k) \) if for the clause \( C \) and atom \( A \) selected for unfolding at every \( P_j \); (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 \( \text{FIN}(\Gamma) \). Condition 1 prohibits infinite unfolding sequences of the form: unfolding \( p(X) \) using the clause \( p(X) : \leftarrow p(X) \) \( i.e. \) unfolding sequences where the same atom is infinitely unfolded. Condition 2 prohibits infinite unfolding sequences of the form: unfolding \( p(X) \) using the clause \( p(a(X)) \) \( i.e. \) where a different atom is unfolded every time, but there are infinitely many atoms to unfold.

If \( \text{FIN} \) prohibits any further unfolding we either apply the folding transformation associated with \textit{Tx} or use the \textit{Gen} rule. Care must be taken, however, when \textit{Gen} is chosen. Recall from the definition of \textit{Gen} (refer Table 1) that \( \Gamma, P_{i+1} \models p \equiv p' \) implies \( \Gamma \models p \equiv p' \) only if we can prove a new equivalence \( P_0 \models q \equiv q' \). 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 \textit{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 \textit{Gen} only to enable \textit{Ax} or \textit{Tx} rules. For instance, consider the transformation sequence in Figure 6. Applying goal replacement in \( P_0 \) under the assumption that that \( P_0 \models q_1 \equiv q_2 \) enables the subsequent folding which transforms \( P_1 \) into \( P_2 \).

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 \( q_1(X) \) and \( q_2(X) \) 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 can never be preserved. As is common in program transformation literature [41,17], 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 \( P_2 \) (in Figure 6), there are two additional predicates \( p_2 \) and \( x_2 \) and further assume that \( p_2 \) 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 the program P3 where p1 and p2 are defined as follows. Thus, in P3 we can conclude that p1 P2 p2.

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

The above intuitions are formalized in Algorithm Prove (see Figure 7). Given a program transformation sequence Γ, and a pair of predicates p, p′, algorithm Prove attempts to prove that Γ ⊢ p ≡ p′. Algorithm Prove searches non-deterministically for a proof; if multiple cases of the non-deterministic 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 non-determinism within a case; for instance, many fold transformations may be applicable at the same time. We again select non-deterministically 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 Algorithm Prove uses two different markings in the process of constructing a proof for Γ ⊢ p ≡ 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 Algorithm Prove via transformations. This marking is essential for ensuring termination of the algorithm. The proof of P0 ⊢ p ≡ p′ may (via a conditional equivalence step) generate the (sub-)equivalence P0 ⊢ p ≡ p′. Algorithm Prove deems this proof path as failed and explores the other proof paths.

Algorithm Prove uses the following functions. Functions unfold(P) and 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 G, G′ s.t. syntactic equivalence of p and p′ can be established after replacing G with 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′, G, G′, Γ) performs the following sequence of steps (where Γ = P0, ..., Pt):

1. first introduces new predicate definitions q and q′ for goals G and G′ respectively (if such definitions do not already exist),
2. proves the equivalence P0 ⊢ q ≡ q′ by invoking Prove,
3. replaces goal G by goal G′ in clauses of p or p′ in program P, to obtain program P+1, and
4. finally invokes Prove to dispense the obligation Γ, P+1 ⊢ p ≡ p′. This completes the proof of Γ ⊢ p ≡ p′.
algorithm Prove(p, p': predicates, \(\Gamma\):prog. seq.)
begin
let \(\Gamma = P_0, \ldots, P_i\)
mark proof_attempt(p, p')
(* Ax rule *)
if \((p \not\equiv p' \lor \text{proved}(p, p'))\) then
return true
else if proof_attempt(p, p') is not marked
non-deterministic choice
(* Tx rule *)
case \(\text{FIN}((\Gamma, \text{unfold}(P_i)))\): (* Unfolding *)
return Prove(p, p', \(\Gamma, \text{unfold}(P_i))\)
case Folding is possible in \(P_i\):
return Prove(p, p', \(\Gamma, \text{fold}(P_i))\)
(* Gen rule *)
case Conditional folding is possible in \(P_i\):
let \((\mathcal{G}, \mathcal{G}') = \text{new_goal_equiv_for_fold}(P_i)\)
return replace_and_prove(p, p', \(\mathcal{G}, \mathcal{G}', \Gamma\))
case Conditional equivalence is possible in \(P_i\):
let \((\mathcal{G}, \mathcal{G}') = \text{new_goal_equiv_for_equiv}(p, p', P_i)\)
return replace_and_prove(p, p', \(\mathcal{G}, \mathcal{G}', \Gamma\))
end choices
mark proved(p, p')
unmark proof_attempt(p, p')
end

Fig. 7. Algorithmic framework for equivalence tableau construction.

Fig. 8. Liveness Proof of n-bit shift register
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 8 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** Algorithm Prove (refer Figure 7) terminates provided the number of definitions introduced (i.e. new predicate symbols added) is finite.

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

\[ \forall \theta \ \text{ground substitutions} \quad p(X)\theta \in M(P_0) \Rightarrow p'(X)\theta \in M(P_0) \]

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 5) which formulates a liveness property about token-passing chains, namely, that the token eventually reaches the leftmost process in any arbitrary length chain. We obtain \( P_0 \), the starting point of our transformation sequence, by annotating each clause of the program in Figure 1 with counter values of (1,1). To establish the liveness property, we prove that \( \text{thm}(X) \equiv \text{gen}(X) \), by invoking Prove(\( \text{thm}, \text{gen}, (P_0) \)). The proof is illustrated in Figure 8.

*Proof of \( \text{thm} \equiv \text{gen} \):* Since \( \text{thm} \not\equiv \text{gen} \), we must transform the predicates. By repeatedly unfolding the definition of \( \text{thm} \) in \( P_0 \), we obtain program \( P_5 \) where \( \text{thm} \) is defined as:

\[
\begin{align*}
\text{thm}([1]). \\
\text{thm}([0|X]) &\leftarrow \text{gen}(X), X = [1|]. \\
\text{thm}([0|X]) &\leftarrow \text{gen}(X), \text{trans}(X,Y), \text{live}([0|Y]).
\end{align*}
\]
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 \( \forall Y \, \text{live}([0|Y]) \Leftrightarrow \text{live}(Y) \) we can fold the last two clauses of \( \text{thm} \). Thus, conditional folding is true at \( P_5 \), and hence \( \text{replace_and_prove} \) is invoked with \( G = \text{live}([0|Y]) \) and \( G' = \text{live}(Y) \). Since \( \text{live}([0|Y]) \) is not an open atom, a new name:

\[
\text{live'}(Y) : - \text{live}([0|Y]).
\]

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

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

We then proceed to prove \( \text{live'} \equiv \text{live} \). This subproof is shown in the left branch of the tree in Figure 8. Then we replace \( \text{live'}(X) \) with \( \text{live}(X) \) in the definition of \( \text{thm} \) in \( P_7 \) (right branch in Figure 8).

**Proof of \( \text{live'} \equiv \text{live} \):** Prove \( \text{live'}(\text{live'}, \text{live}, (P_0)) \) 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 \( P_{10}, \text{live'} \) is defined by the following clauses:

\[
\text{live'}([1|Z]).
\text{live'}(X) : - \text{trans}(X,Z), \text{live}([0|Z]).
\]

Folding is applicable is \( P_{10} \), in the second clause of \( \text{live'} \), yielding \( P_{11} \) with

\[
\text{live'}([1|Z]).
\text{live'}(X) : - \text{trans}(X,Z), \text{live'}(Z).
\]

Now, \( \text{live'} \overset{R}{\Rightarrow} \text{live} \) and hence \( \text{Prove}(\text{live'}, \text{live}, (P_0)) \) 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 \( \text{thm} \equiv \text{gen} \):** Now \( \text{replace_and_prove} \) replaces \( \text{live'}(X) \) with \( \text{live}(X) \) in the definition of \( \text{thm} \) in \( P_7 \), yielding \( P_{12} \) with:

\[
\text{thm}([1]).
\text{thm}([0|X]) : - \text{gen}(X), X = [1|].
\text{thm}([0|X]) : - \text{gen}(X), \text{trans}(X,Y), \text{live}(Y).
\]

We can now fold the last two clauses of \( \text{thm} \) using the definition of \( \text{live} \) 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 [37] was the first to permit such folding. Thus we obtain \( P_{13} \):
\texttt{thm([1])}.
\texttt{thm([0]|X]) :- gen(X), live(X).

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

\texttt{thm([1])}.
\texttt{thm([0]|X]) :- thm(X).

We now have \( \texttt{thm} \stackrel{P_{14}}{=} \texttt{gen} \), 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 \( \texttt{thm} \equiv \texttt{gen} \).

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 \textit{Prove} for guiding the application of the rules in the proof system. However, this algorithmic framework \textit{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 of the last section) is involved; details appear in [39].

In this section, we present the experimental results obtained using our predicate equivalence prover. The prover is built on top of the XSB tabular logic programming system [45] which supports top-down memoized evaluation of logic programs. We report results on parameterized cache coherence protocols, including (a) single bus broadcast protocols \textit{e.g.} Mesi, (b) single bus protocols with global conditions \textit{e.g.} Illinois and (b) multiple bus hierarchical protocols. We also report experimental numbers 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 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, 13, 16]. Illinois is a single bus cache coherence protocol with global conditions which cannot be modeled as a broadcast protocol [10]. Tree-cache is a binary tree network which simulates the interactions between the cache agents in a hierarchical cache coherence protocol [39]. Finally German’s client server is a client-server protocol proposed by Steve German. It involves unbounded number of client accessing a central server (a star network topology) and we need to prove coherence of cached copies across the clients. It was proposed as a benchmark for parameterized system verification by Ruah, Pnueli and Zuck in [33].

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Invariant</th>
<th>Time(sec)</th>
<th>Our # Unf</th>
<th># Ded</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>in [10]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meta-Lock</td>
<td>#owner +</td>
<td></td>
<td>129.8</td>
<td>1981</td>
</tr>
<tr>
<td></td>
<td>#handout &lt; 2</td>
<td></td>
<td>311</td>
<td></td>
</tr>
<tr>
<td>Mesi</td>
<td>#m + #e &lt; 2</td>
<td>1</td>
<td>3.2</td>
<td>325</td>
</tr>
<tr>
<td></td>
<td>#m + #e - 0 V</td>
<td>0.5</td>
<td>2.9</td>
<td>308</td>
</tr>
<tr>
<td></td>
<td>#s - 0</td>
<td></td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>Illinois</td>
<td>#dirty &lt; 2</td>
<td>5.3</td>
<td>35.7</td>
<td>2501</td>
</tr>
<tr>
<td>Berkeley</td>
<td>#dirty &lt; 2</td>
<td>6.6</td>
<td>6.8</td>
<td>303</td>
</tr>
<tr>
<td>German’s</td>
<td>ex &lt; 2</td>
<td></td>
<td>8.8</td>
<td>404</td>
</tr>
<tr>
<td>client-server</td>
<td>#ex + #ah &lt; 2</td>
<td></td>
<td>Fails</td>
<td>-</td>
</tr>
<tr>
<td>Tree-cache</td>
<td>#bus_with_data &lt; 2</td>
<td></td>
<td>9.9</td>
<td>178</td>
</tr>
</tbody>
</table>

Table 2. Summary of protocol verification results

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, and comparison of predicate definitions) 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 [10]. To the best of our knowledge, [10] is the only other work which employs logic programming technology for parameterized system verification, and reports detailed experimental results. Note that the timings of [10] on a Pentium 133 with Linux 2.0.32.

Comparison with [10] The running times of our prover are slower than the times for verifying single bus cache coherence protocols reported in [10]. Indeed, there is up to an order of magnitude difference between the time taken by our prover and the time taken by the prover of [10]. The reason for this can be explained as follows. The constraint logic program evaluation based model checker of [10] proceeds essentially by unfolding which is performed by the underlying abstract machine. On the other hand, the prototype implementation of our transformation based prover implements both the unfolding and folding steps via meta-programming. Even though our folding steps should ideally be implemented by meta-programming, the proof search conducted by the unfolding steps can be implemented at the level of the underlying abstract machine (which should substantially reduce our running times). It would be more interesting to compare the running times of such a prover with the Constraint logic program execution in [10]. Such a comparison would determine whether the time overhead due to folding steps exceeds the time overhead due to constraint solving in Constraint logic program evaluation. Also, note that the abstraction based technique of [10] 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. Note that for one of the benchmarks (German’s client server protocol), our prover fails to construct a proof of the invariant that at most one one client can be in shared or exclusive state. In this case our prover terminates.
and reports its failure to find a proof (even though one exists). This needs to be remedied by manually strengthening the invariant to be proved. However, the desired strengthening turns out to be non-obvious in this case.

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

Relatively little work has been done on using unfold/fold 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 [20, 21, 23, 30, 31] to construct inductive proofs of program properties.

Hsiang and Srice in [20] 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 [21, 23] 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), [21] 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 [30, 31]. These works employ more restricted Tamaki-Sato style unfold/fold transformations, which are not suitable in general for constructing induction proofs of temporal properties. This is because
temporal properties employ fixed point operators, and are typically encoded using multiple recursive clauses. As shown in Section 5.2, a simple reachability property $EFp$ (which specifies that a state in which proposition $p$ holds is reachable) [9] can be encoded via a predicate $ef$; the definition of $ef$ contains two clauses, one of which is recursive. The current unfold/fold transformation systems for definite logic programs do not allow such clauses to be used as folder in a folding step. Our work relaxes restrictions on the applicability of transformation rules (in particular the folding rule), enabling their use in proving temporal properties.

The reader might notice similarities between a proof system based on unfold/fold transformations and a proof system based on tabled resolution [7,43]. 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 called 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 [29]. 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 [34].

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 we can verify (families of) timed systems with the same proof technique. Finally, note that the proof technique supports zero overhead theorem proving [40]. 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 [39]. We would like to thank the following people for actively collaborating in the research reported here: K. Narayan Kumar, I.V. Ramakrishnan and S.A. Smolka. This work was partially supported by NSF grants CCR-9711386, CCR-9876242, CDA-9805735 and EIA-9705998.

References