Verification of Parameterized Systems Using Logic-Program Transformations

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

[1] Dept. of Computer Science
SUNY Stony Brook
Stony Brook, NY 11794, USA
{abhik,cram,ram,sas}@cs.sunysb.edu

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

Abstract

We show how the problem of verifying parameterized systems can be reduced to the problem of determining the equivalence of goals in a logic program. We further show how goal equivalences can be established using induction-based proofs. Such proofs rely on a powerful new theory of *logic-program transformations* (encompassing unfold, fold and goal replacement over multiple recursive clauses), can be highly automated, and are applicable to a variety of network topologies, including uni- and bi-directional chains, rings, and trees of processes. Unfold transformations in our system correspond to algorithmic model-checking steps, fold and goal replacement correspond to program deductions, and all three types of transformations can be arbitrarily interleaved within a proof. Our framework thus provides a seamless integration of algorithmic and deductive verification at fine levels of granularity.

Contact Author:	C.R. Ramakrishnan
E-mail:	cram@cs.sunysb.edu
Phone:	+1 516 632 8218
Fax:	+1 516 632 8334

1 Introduction

Advances in Logic Programming technology are beginning to influence the development of new tools and techniques for the specification and verification of concurrent systems. For example, constraint logic programming has been used for the analysis and verification of hybrid systems [UR95, Urb96] and more recently for model checking infinite-state systems [DP99]. Closer to home, we have used a tabled logic-programming system to develop XMC, an efficient and flexible model checker for finite-state systems [RRR+97]. XMC is written in under 200 lines of tabled Prolog code, which constitute a declarative specification of CCS and the modal mu-calculus at the level of semantic equations. Despite the high-level nature of XMC's implementation, its performance is comparable to that of highly optimized model checkers such as Spin [Hol97] and Murφ [Dil96] on examples selected from the benchmark suite contained in the standard Spin distribution.

XMC's efficiency derives in part from its reliance on SLG resolution, the extension of Prolog-style SLD resolution with tabled resolution, implemented in the XSB logic programming system [XSB99]. Essentially, we have used XSB as a programmable fixed-point engine, motivated in part by the success of Steffen et al. in [SCK+95]. XMC's encoding centers around two predicates: `trans`, encoding the transitional semantics of CCS terms, and `models`, defining when a CCS term models a given modal mu-calculus formula. By appropriately redefining these predicates, model checkers for other process description languages and other temporal logics can be readily obtained. For example, we recently retargeted XMC to linear temporal logic (LTL) by redefining `models` in accordance with the proof system given in [BCG95].

We have also been actively investigating how XMC's model-checking capabilities can be extended *beyond finite-state systems*. Essentially, this can be done by enhancing the underlying resolution strategy appropriately, and such extensions need not involve modifying the internals of the XSB system. Rather, they can be obtained at the level of *meta-programming*, and without the undue performance penalties typically associated with the concept of meta-programming. In this sense, XMC can be viewed as a *programmable verification engine*. For example, we have shown in [DRS99] how an efficient model checker for real-time systems can be attained through the judicious use of a constraint package for the reals on top of tabled resolution.

In this paper, we expand on this theme even further. In particular, we examine how the tabled-resolution approach to model checking finite-state systems can be extended to the verification of *parameterized systems*. A parameterized system represents an *infinite* family of systems, each instance of which is finite state. For example, an n-bit shift register is a parameterized system, the parameter in question being n, the length of the shift register. In general, 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 parameterized systems.

The main idea underlying our approach to problem of verifying parameterized systems is to reduce it to the problem of determining the equivalence of goals in a logic program. We then establish goal equivalences using *induction-based proofs*. To derive such induction proofs we were required to substantially generalize the well-established theory of *logic-program transformations* encompassing unfold, fold and goal-replacement transformations. In particular in a recent paper [RKRR99b] we developed a powerful transformation system for folding using multiple recursive clauses which we will later show is crucial for proving properties of parameterized systems.

In our framework, unfold transformations, which replace instances of clause left-hand sides with corresponding instances of clause right-hand sides, represent resolution. They thereby represent a form of *algorithmic* model checking; viz. the kind of algorithmic model checking performed in XMC. 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.

Fold transformations, which replace instances of clause right-hand sides with corresponding instances of clause left-hand sides, and goal-replacement transformations, which replace a goal in an instance of a clause right-hand side with a semantically equivalent goal, 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 applicability of the goal-replacement transformation is governed by certain integer inequalities. An interesting aspect of our approach is the use of Integer Linear Programming to characterize when it is sound to apply goal replacement. In sum, we combine unfold, fold, and goal-replacement transformations into a framework for the automated derivation of induction proofs for the verification of parameterized systems.

Using this approach, we have been able to establish the correctness of a number of parameterized systems, with respect to both safety and liveness properties. Moreover, our approach does not seem limited to any particular kind of network topology, as the systems we considered have included uni- and bi-directional chains, rings, and trees of processes.

The primary benefits of our approach can be summarized as follows.

- *Uniform framework.* Our research has shown that finite-state systems, real-time systems, and, now, parameterized systems can be uniformly specified and verified within the framework of tabled logic programming.

- *Tighter integration of algorithmic and deductive model checking.* As our examples illustrate, unfold, fold, and goal-replacement steps can be arbitrarily interleaved within the inductive proof of a parameterized system. Thus our approach allows algorithmic model checking computation (unfold) to be seamlessly integrated with deductive reasoning (fold, goal replacement) at fine levels of granularity. Furthermore, since deductive steps are applied lazily in our approach, finite-state model checking emerges as a special case.

- *High degree of automation.* Although a fully automated solution to the model-checking problem for parameterized systems is not possible, for many cases of practical interest, including all the examples in this paper, we have identified certain heuristics that can be applied to our deduction system in order to completely automate the deductive process.

The idea of using logic program transformations for proving goal equivalences was first explored in [PP99] for logic program synthesis. Our work expands the existing body of work with more powerful transformations and algorithms that are central to verification of parameterized systems.

Regarding related work in the verification area, a myriad of techniques have been proposed during the past decade for verifying parameterized systems, and the related problem of verifying infinite-state systems. These include [BCG89, EN95, ID96], which reduce the problem of verifying a parameterized system to the verification of an "equivalent" finite-state system, and [WL89, KM95, LHR97], which seek to identify a "network invariant" that is invariant with respect to the given notion of parallel composition and stronger than the property to be established. The network-invariant approach is applicable to parameterized systems consisting of a number of copies of identical components (or components drawn from some finite set) that are composed in parallel.

Another approach [CGJ95] aims to *finitely* represent the state space and transition relation of the entire family of finite-state systems comprising a given parameterized system, and has been used in [KMM^{+}97] to extend symbolic model checking [McM93] to the verification of parameterized systems. This method requires the construction of a uniform representation for each class of networks, and the property in question must have a proof that is uniform across the family of networks.

```
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

Figure 1: Example: Liveness in a unidirectional token-passing chain.

Perhaps the work most closely related to our own involves the use of inductive theorem provers for verifying parameterized systems. Rajan et al. [RSS95] have incorporated a finite-state model checker for the modal mu-calculus as a decision procedure within the PVS theorem prover [OSR92]. Inductive proofs can be established by the prover via calls to the model checker to verify finite subparts. Graf and Saidi [GS96] show how a custom-built specification-deduction system can be combined with PVS to formalize and carry out model checking of invariant properties using deduction.

The key difference between our approach and these is that we enhance model checking with deductive capabilities, rather than implement model checking as a decision procedure in a deductive system. In particular, the underlying evaluation mechanism for model checking in XMC is essentially unfolding, and we have enhanced this mechanism with fold, and goal-replacement transformations. These transformations complement the power of model checking with the ability to do deduction. Moreover, deductive steps are deployed only on demand and hence do not affect the efficacy of the algorithmic model-checking. More importantly our framework demonstrates that a tabled constraint logic-programming system can form the core of a verification engine that can be programmed to verify properties of various flavors of concurrent systems including finite-state, real-time, and parameterized systems.

2 Verification of Parameterized Systems as Goal Equivalence

In this section, we discuss how the problem of verifying temporal properties of parameterized systems can be reduced to that of checking the equivalence of goals in a logic program.

Modeling Infinite Families of Finite-State Systems: Consider the parameterized system consisting of a chain of n token-passing processes. In the system's initial state, the process in the right-most position of the chain has the token and no other process has a token. The system evolves by passing the token 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. A global state is represented as an ordered list[1] of zeros and ones, each bit corresponding to a local state, and the head of the list corresponding to the local state of the left-most process in the chain. Each process in the chain is a two-state automaton: one with the token (an entry of 1 in the list) and the other without the token (an entry of 0). 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 token from right to left; the second clause recursively searches the state representation until the first clause can be applied (i.e., when the token is not already in the left-most process).

Liveness Properties: The predicate `live` in Figure 1 encodes the temporal property we wish to verify: eventually the token reaches the left-most process. The first clause succeeds for global states

[1]A list in Prolog-like notation is of the form `[Head|Tail]`.

where the token is already in the left-most process (a good state). The second (recursive) clause checks if a good state is reachable after a (finite) sequence of transitions.

Thus, every member of the family satisfies the liveness property if and only if $\forall$ `X` `gen(X)` $\Rightarrow$ `live(X)`. Moreover, this is the case if $\forall$ `X` `thm(X)` $\Leftrightarrow$ `gen(X)`, i.e., if `thm` and `gen` are equivalent (have the same least model). Clearly, testing the equivalence of these goals is infeasible since the minimal model of the logic program is infinite. However, we present in Section 3 a proof methodology, based on program transformations, for proving equivalences between such goals in a logic program.

Safety Properties: We can model safety properties by introducing negation into the above formulation for liveness properties, using the temporal-logic identity $G\ \phi \equiv \neg F\ \neg\phi$. Although our program transformation systems have been recently extended to handle programs with negation [RKRR99a], we present an alternative formulation without negation, since the corresponding equivalence proofs are simpler. In particular, we define a predicate `bad` to represent states that violate the safety property, show that the start states are not `bad`, and, finally, show that `bad` states are reachable only from other `bad` states. For instance, mutual exclusion in the n-process chain can be verified using the following program:

```
bad([1|Xs]) :- one_more_token(Xs).
bad([_|Xs]) :- bad(Xs).

one_more_token([1|_]).
one_more_token([_|Xs]) :- one_more_token(Xs).
```

```
bad_start(X) :- gen(X), bad(X).

bad_src(X,Y) :- trans(X, Y), bad(X).
bad_dest(X,Y) :- trans(X, Y), bad(Y).
```

`bad` is true if and only if the given global state has more than one local state with a token. Showing `bad_start(X)` $\Leftrightarrow$ `false` establishes that the start states do not violate the safety property. Showing that `bad_src(X)` $\Leftrightarrow$ `bad_dest(X)` establishes that states that violate the safety property can be reached only from other states that violate the property. These two facts together imply that no reachable state in the infinite family is bad and thus establish the safety property for the entire family.

A Note on the Model: XMC [RRR$^+$97] provides a highly expressive process description language based on value-passing CCS [Mil89] for specifying parameterized systems (although only finite-state systems can actually be analyzed), and the modal mu-calculus for specifying temporal properties. The above simplified presentation (which we will continue to use in the rest of this paper) is used to prevent a proliferation of syntax from obscuring the key issues.

3 Goal Equivalence Proofs using Tableau

As observed in Section 2, to establish the liveness property of n-process chains (Figure 1), we must show that `gen(X)` and `thm(X)` are equivalent. In this section we describe the basic framework to construct such equivalence proofs. We begin by defining the notations used in formalizing the framework.

Notations: We assume familiarity with the standard notions of terms, models, substitutions, unification, and most general unifier (mgu) [Llo93]. A term having no variables is called a *ground* term. *Atoms* are terms with a predicate symbol at the root, and *goals* are conjunctions of atoms. Atoms whose subterms are distinct variables (i.e., atoms of the form $p(X_1, \ldots, X_n)$, where p is a predicate symbol of arity n) are called *open atoms.*

We use the following notation (possibly with primes and subscripts): p, q for predicate symbols; X, Y for variables; t, s for terms; $\overline{X}, \overline{Y}$ for sequences of variables; $\overline{t}, \overline{s}$ for sequences of terms; A, B for atoms; σ, θ for substitutions; C, D for Horn clauses; α, β for goals; and P for a logic program, which is a set of Horn clauses. A Horn clause C is written as $A \coloneq B_1, B_2, \ldots, B_n$. A, the consequent, is

$$\textbf{(Ax)} \quad \frac{\Gamma \ \vdash \ \alpha \equiv \beta}{} \qquad \text{where } \ \alpha \overset{P_i}{\cong} \beta$$

$$\textbf{(Tx)} \quad \frac{\Gamma \ \vdash \ \alpha \equiv \beta}{\Gamma, P_{i+1} \ \vdash \alpha \equiv \beta} \qquad \text{where } \ M(P_{i+1}) = M(P_i)$$

$$\textbf{(Gen)} \quad \frac{\Gamma \ \vdash \ \alpha \equiv \beta}{\Gamma, P_{i+1} \ \vdash \alpha \equiv \beta, \quad P_0 \ \vdash \alpha' \equiv \beta'} \qquad \text{where } \ M(P_{i+1}) = M(P_i) \text{ if } \alpha' \equiv \beta'$$

Figure 2: Rules for Constructing Equivalence Tableau.

called the *head* of C and the antecedent $B_1, B_2, \ldots, B_n$ the *body* of C. Note that we can write Horn clauses as $A :\!- \alpha$.

Semantics of a definite logic program P is given in terms of least Herbrand models, $M(P)$. Note that the body of Horn clauses do not contain negation, and our programs, defined above as set of Horn clauses, are in fact definite logic programs. Given a goal α and a program P, SLD resolution is used to prove whether instances of α are in $M(P)$. This proof is constructed recursively by deriving new (sub)goals by replacing an atom B in α with $\beta\theta$ where $B' :\!- \beta \in P$ and $\theta = mgu(B, B')$.

We use $P_0, P_1, \ldots, P_n$ to denote a transformation sequence where P_{i+1} is obtained from P_i by applying a single transformation. We call P_0 as the *original* program.

3.1 Tableau Construction

The *goal equivalence problem* is: given a logic program P and a pair of goals α and β, determine if α and β are semantically equivalent in P: i.e., whether for all ground substitutions θ, $\alpha\theta \in M(P) \Leftrightarrow \beta\theta \in M(P)$. This problem is undecidable in general and we attempt to provide a deductive system for identifying equivalence.

We now develop a tableau-based proof system for establishing goal equivalence. Our process is analogous to SLD resolution. Each node in the proof tree denotes a pair of goals. To establish their equivalence we must establish that the (sub)goals in the pair represented by each child node are equivalent. We formalize our tableau below.

Let $\Gamma = \langle P_0, P_1, \ldots, P_i \rangle$ be a sequence of logic programs such that P_{j+1} is obtained from P_j $(1 \leq j < i)$ by the application of a rule in our tableau. Further let $M(P_0) = M(P_1) = M(P_2) = \ldots = M(P_i)$. An *e-atom* is of the form $\Gamma \ \vdash \ \alpha \equiv \beta$ where α and β are goals, and represents our proof obligation: that $\alpha \equiv \beta$ are semantically equivalent in any (and hence in each) of the programs in Γ. An *e-goal* is a (possibly empty) sequence of e-atoms (e-atoms correspond to the atoms and e-goals to the goals in standard resolution).

The three rules used to construct equivalence tableau are shown in Figure 2. The *axiom elimination rule* (**Ax**) is applicable whenever the equivalence of goals α and β can be established by some automatic mechanism, denoted in the rule by $\alpha \overset{P_i}{\cong} \beta$. Axiom elimination is akin to the treatment of facts in SLD resolution.

The *program transformation rule* (**Tx**) attempts to simplify the program in order to expose the equivalence of goals. The program P_{i+1} is constructed from Γ using some 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.

The *sub-equivalence generation rule* (**Gen**) attempts to replace a pair of goals whose equivalences have to be established, with a new pair of (sub)goals whose equivalences are (hopefully) simpler to establish. This step corresponds to the standard SLD resolution step. The proof of $\alpha \equiv \beta$ is built

```
p :- t, s.        p :- t, s.        p :- q.
q :- r, s.        q :- t, s.        q :- t, s.
r :- t.           r :- t.           r :- t.
...               ...               ...
 Program P0        Program P1        Program P2
```

Figure 3: Example of an unfold/fold transformation sequence.

using the proof of $\alpha' \equiv \beta'$. Note that the proof of $\alpha' \equiv \beta'$ may involve a transformation sequence different from, and not just an extension of, Γ.

A *successful tableau* for an e-goal E_0 is a finite sequence of e-goals $E_0, E_1, \ldots, E_n$ where E_{i+1} is obtained from E_i by applying one of the rules described above and E_n is empty.

Theorem 1 *Let $E_0, E_1 \ldots, E_n$ be a successful tableau with $E_0 = \langle P_0 \rangle \vdash \alpha \equiv \beta$ for some (definite) logic program P_0. Then for all ground substitutions θ, $\alpha\theta \in M(P_0) \Leftrightarrow \beta\theta \in M(P_0)$, i.e., goals α and β are equivalent in the least Herbrand model of P_0.*

The tableau, however, is not complete: there can be no such complete tableau as attested to by the following theorem.

Theorem 2 *The problem of determining equivalence of predicates described by logic programs is not recursively enumerable.*

The above theorem is easily proved using a reduction described in [AK86].

3.2 Program Transformations

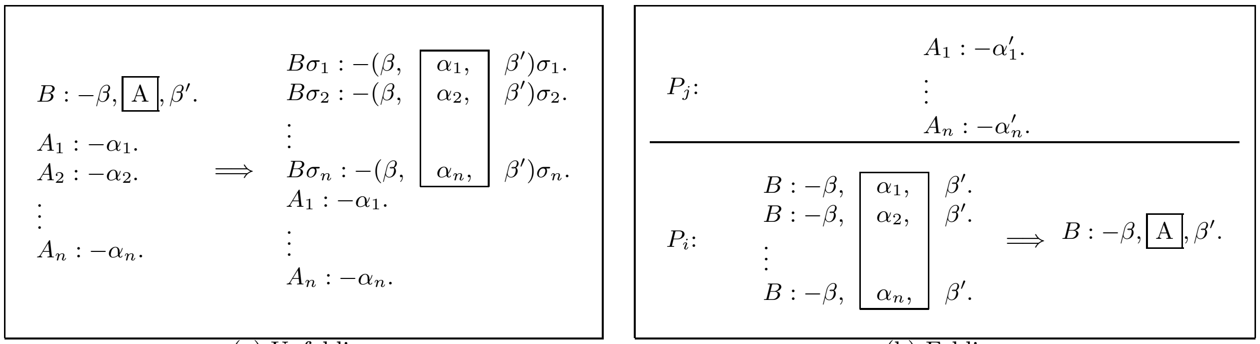
The tableau-based equivalence proof is constructed by repeatedly applying rules **Ax**, **Tx**, and **Gen**. These rules are realized concretely by logic-program transformations that include unfold, fold and goal replacement. To make this paper self contained we review them now.

3.2.1 Review of Logic Program Transformations

For a simple illustration of program transformations, consider Figure 3. There, program P_1 is derived from P_0 by *unfolding* the occurrence of `r` in the definition of `q`. P_2 is derived from P_1 by *folding* `t,s` in the definition of `p` using the definition of `q`.

While unfolding is always semantics preserving, indiscriminate folding may introduce circularity in definitions, thereby replacing finite proof paths with infinite ones. For example, folding `t,s` in the definition of `q` in P_2 using the definition of `p` in P_0 results in a program of the form `p :- q.  q :- p.  r :- t.  ...`, thereby removing `p` and `q` from the least model.

Fold transformations are guided by certain bookkeeping information that summarizes the effect of prior transformations. In [RKRR99b] we proposed a general bookkeeping mechanism and described an abstract framework for transformations which permits folding using clauses that contain recursion and disjunction: a fact that we will later see as a key to its application to automated verification. Below, we instantiate the framework of [RKRR99b] as follows. With each clause C in program P_i of the transformation sequence, we associate a pair of integer counters $\gamma^i_{lo}(C)$ and $\gamma^i_{hi}(C)$. The counters bound the size of a shortest proof for any ground goal in program P_i relative to the size of a shortest proof in P_0. Thus, counters associated with each clause keep track of potential reductions in proof lengths. Conditions on counters are then used to determine if a given application of folding will preserve all proofs.



$B :- \beta, \boxed{A}, \beta'.$
$A_1 :- \alpha_1.$
$A_2 :- \alpha_2.$
$A_n :- \alpha_n.$
$\Longrightarrow$
$B\sigma_1 :- (\beta, \alpha_1, \beta')\sigma_1.$
$B\sigma_2 :- (\beta, \alpha_2, \beta')\sigma_2.$
$B\sigma_n :- (\beta, \alpha_n, \beta')\sigma_n.$
$A_1 :- \alpha_1.$
$A_n :- \alpha_n.$
P_j:
$A_1 :- \alpha'_1.$
$A_n :- \alpha'_n.$
P_i:
$B :- \beta, \alpha_1, \beta'.$
$B :- \beta, \alpha_2, \beta'.$
$B :- \beta, \alpha_n, \beta'.$
$\Longrightarrow$
$B :- \beta, \boxed{A}, \beta'.$

(a) Unfolding (b) Folding

Figure 4: Schema for Unfold/Fold transformations.

We now present the program transformations formally. Unfolding is applied to an atom A in the body of a clause in P_i and is diagrammatically illustrated in Figure 4a.

Transformation 1 (Unfolding) Given a clause C and an atom A in P_i such that C is of the form $B :- \beta, A, \beta'$, unfolding A in C derives program P_{i+1} as follows. Let $C_1, ..., C_n$ be all the clauses in P_i such that C_j $(1 \leq j \leq n)$ is of the form $A_j :- \alpha_j$, and A and A_j unify with mgu σ_j. Then, $P_{i+1} := (P_i - \{C\}) \cup \{C'_1, ..., C'_n\}$, where C'_j is of the form $B\sigma_j :- (\beta, \alpha_j, \beta')\sigma_j$.
Also, $\gamma_{lo}^{i+1}(C'_j) = \gamma_{lo}^{i}(C) + \gamma_{lo}^{i}(C_j)$ and $\gamma_{hi}^{i+1}(C'_j) = \gamma_{hi}^{i}(C) + \gamma_{hi}^{i}(C_j)$. The counters of all other clauses in P_{i+1} are inherited from P_i. □

Note that the conditions of unifiability of A with A_j, and the subsequent application of the mgu σ_j for deriving the new clauses are taken directly from resolution.

A folding transformation replaces an occurrence of the body of a clause with its head. The clause where the replacement takes place is called the *folded* clause and the clauses used to perform the replacement are called the *folder* clauses. The folding schema is illustrated in in Figure 4b, where the clauses of B are the folded clauses, and the clauses of A are the folder clauses. The folder clauses may come from some earlier program in the transformation sequence, i.e., from P_j for some $j \leq i$.

Transformation 2 (Folding) Given clauses $C_1, ..., C_n$ in P_i such that C_l is of the form $B :- \beta, \alpha_l, \beta'$ for $1 \leq l \leq n$ and clauses $D_1, ..., D_n$ in P_j $(j \leq i)$ such that D_l is of the form $A_l :- \alpha'_l$ satisfying :
1. $\forall 1 \leq l \leq n\ \exists \sigma_l.\ \alpha_l = \alpha'_l \sigma_l$ where σ_l is a substitution.
2. there is an atom A such that $\forall 1 \leq l \leq n\ A_l\sigma_l = A$ and $D_1, ..., D_n$ are the only clauses in P_j whose heads unify with A.
3. $\forall 1 \leq l \leq n\ \sigma_l$ substitutes the internal variables of D_l to distinct variables that do not appear in $\{A, B, \beta, \beta'\}$
4. $\forall 1 \leq l \leq n\ \ \gamma_{hi}^{j}(D_l) < \gamma_{lo}^{i}(C_l) + p$.

Then, $P_{i+1} := (P_i - \{C_1, ..., C_n\}) \cup \{C'\}$ where C' is $B :- \beta, A, \beta'$. Also,
$\gamma_{lo}^{i+1}(C') = min_{1 \leq l \leq n}(\gamma_{lo}^{i}(C_l) - \gamma_{hi}^{j}(D_l))$, $\gamma_{hi}^{i+1}(C') = max_{1 \leq l \leq n}(\gamma_{hi}^{i}(C_l) - \gamma_{lo}^{j}(D_l))$. The counters of all other clauses in P_{i+1} are inherited from P_i. □

The goal-replacement transformation replaces a goal in the body of a clause in program P_i with a semantically equivalent goal. Note that such a replacement has the potential to change lengths of proofs arbitrarily. In order to maintain the counters associated with each clause, we need to estimate the change in proof lengths. We do so by using the notion of *atom measure* [RKRR99b]. A ground proof of an atom is an SLD proof that contains only ground terms. The atom measure of a ground atom A is the size of the smallest ground proof for A in P_0, the original program. We use $w(A)$ to denote the atom measure of A.

```
p(X) :- r(X).             (γ1, γ1')     q(X) :- s(X).             (μ1, μ1')
p(X) :- e(X,Y), p(Y).     (γ2, γ2')     q(X) :- e(X,Y), q(Y).     (μ2, μ2')
r(X) :- b(X).             (γ3, γ3')     s(X) :- b(X).             (μ3, μ3')
```

Figure 5: Program with syntactically equivalent predicates.

Transformation 3 (Goal Replacement) Given a clause C of the form $A :- \alpha, B, \beta$ in P_i, an atom B' such that $vars(B) = vars(B') \subseteq vars(A) \cup vars(\alpha) \cup vars(\beta)$, and two integers δ and δ' obeying the following conditions:
1. for all ground substitutions θ: *(i)* $P_i \vdash B\theta \Leftrightarrow P_i \vdash B'\theta$, and *(ii)* $\delta \leq w(B\theta) - w(B'\theta) \leq \delta'$
2. $\gamma^i_{lo}(C) + \delta + k > 0$

Then $P_{i+1} := (P_i - \{C\}) \cup \{C'\}$ where C' is $A :- \alpha, B', \beta$, and
$\gamma^{i+1}_{lo}(C') = \gamma^i_{lo}(C) + \delta$ and $\gamma^{i+1}_{hi}(C') = \gamma^i_{hi}(C) + \delta'$. □

The applicability of goal replacement transformation depends on conditions that are undecidable in general: (i) the equivalence of two atoms B and B'. (ii) existence of values δ and δ' that bound the atom measures of $B\theta$ and $B'\theta$ for any substitution θ. We will use (recursively) the equivalence tableau to prove $B \equiv B'$. We later describe a technique to estimate bounds on atom measures based on that proof.

Theorem 3 ([RKRR99b]) *Let $P_0, P_1, \ldots, P_N$ be a sequence of definite logic programs where P_{i+1} is obtained from P_i by an application of unfolding, folding, or goal replacement (Transformations 1, 2, and 3). Then $\forall 1 \leq i \leq N$, $M(P_i) = M(P_0)$.*

Definition-introduction transformation adds a clause defining a new predicate to a program. This transformation is used to generate "names" for goals. By introducing definitions and folding subsequently, we can replace indirect and multiple recursion by single, direct recursion.

Transformation 4 (Definition Introduction) Given a goal α, and a predicate symbol p that does not appear in P_i, let C be $(p(\overline{X}) :- \alpha)$, where $\overline{X} = vars(\alpha)$.
Then, $P_{i+1} := P_i \cup \{C\}$ and $\gamma^{i+1}_{lo}(C) = \gamma^{i+1}_{hi}(C) = 1$. □

Note that, after definition introduction $M(P_{i+1}) \neq M(P_i)$ since a new predicate is added in P_{i+1}. But for every predicate p in P_i, and all ground terms $\overline{t}$, $p(\overline{t}) \in M(P_i) \Leftrightarrow p(\overline{t}) \in M(P_{i+1})$. The tableau presented earlier can be readily extended to include such transformations. In addition, a number of other standard transformations, such as deletion of subsumed clauses can be readily adapted to our system of bookkeeping with counters. (Details are omitted.)

3.2.2 Computational Aspects of Goal Equivalence

Recall that the axiom elimination rule (**Ax**) is applicable whenever we can mechanically establish the equivalence of two goals. We now develop a *syntax-based* technique to establish the equivalence of two *open atoms*, i.e., atoms of the form $p(\overline{X})$ and $q(\overline{X})$. This technique will, by itself, fail to find equivalence of more general goals; in Section 4, we describe methods which use this simpler test to show general goal equivalence.

A syntactic test for goal equivalence: We first introduce the notion of syntactic equivalence between predicates. Consider the program (with clauses annotated with counter values) in Figure 5. We can infer that `r(X)` $\equiv$ `s(X)` since `r` and `s` have identical definitions. In addition, we can infer that `q(X)` and `p(X)` are equivalent, since their definitions are, in a sense, isomorphic. We formalize this notion of equivalence in the following definition.

Definition 5 (Syntactic Equivalence of Predicates) *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 name 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 $D'_{f(i)}$, and
   (ii) $\forall 1 \leq j \leq n$ D'_j is an instance of $C'_{g(j)}$.

Note that the largest syntactic equivalence relation 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 soundness of syntactic equivalence with respect to semantic equivalence is ensured by the following lemma.

Lemma 4 *Let P be a program and $\overset{P}{\sim}$ be the syntactic equivalence relation. For all predicates p, q, if $p \overset{P}{\sim} q$, then $p(\overline{X}) \equiv q(\overline{X})$.*

This lemma can be proved by induction on the length of ground proofs T. We define syntactic equivalence between goals as: $\alpha \overset{P}{\cong} \beta$ if and only if $\alpha = p(\overline{X})$ and $\beta = q(\overline{X})$ where p, q are predicates such that $p \overset{P}{\sim} q$.

Recall that every application of goal replacement requires a computation of bounds on atom measures. We now develop a technique based on Integer Linear Programming to estimate these bounds.

Test for applicability of goal replacement: Assume that we need to replace $p(\overline{X})$ with $q(\overline{X})$ using Transformation 3. Let $\Delta(p, q)$ and $\Delta'(p, q)$ denote the lower and upper bounds of atom measures as specified by the transformation. That is,

$$\forall \text{ ground substitutions } \theta \quad \Delta(p, q) \leq w(p(\overline{X})\theta) - w(q(\overline{X})\theta) \leq \Delta'(p, q)$$

We first establish $p(\overline{X}) \equiv q(\overline{X})$ by constructing a transformation sequence $P_0, P_1, \ldots, P_k$ such that $p \overset{P_k}{\sim} q$. We can establish the following theorem relating the values of Δ and Δ' for equivalent predicates.

Theorem 5 (Values of Δ and Δ') *Let p and q be two distinct predicates in program P such that $p \overset{P}{\sim} q$. Let the clauses of p in P be $C_1, \ldots, C_m$ with clause annotations $(\gamma_1, \gamma'_1), \ldots, (\gamma_m, \gamma'_m)$. Let the clauses of q in P be $D_1, \ldots, D_n$ with clause annotations $(\mu_1, \mu'_1), \ldots, (\mu_n, \mu'_n)$. Moreover, let each clause C_i be $p(\overline{t}) \coloneq p_{i,1}(\overline{t_{i,1}}), \ldots, p_{i,k_i}(\overline{t_{i,k_1}})$, and D_j be $q(\overline{s}) \coloneq q_{j,1}(\overline{s_{j,1}}), \ldots, q_{j,l_j}(\overline{s_{j,l_j}})$. Finally, let f and g be the mappings (from Definition 5) used to show $p \overset{P}{\sim} q$. Then,*

$$\begin{aligned}
\Delta(p,p) &= \Delta(q,q) = \Delta'(p,p) = \Delta'(q,q) = 0 \\
\Delta(p,q) &\leq \min_{1 \leq i \leq m} [(\gamma_i - \mu'_{f(i)}) + \sum_{1 \leq l \leq k_i} \Delta(p_{i,l}, q_{f(i),l})] \\
\Delta'(p,q) &\geq \max_{1 \leq j \leq n} [(\gamma'_{g(j)} - \mu_j) + \sum_{1 \leq l \leq l_j} \Delta'(p_{g(j),l}, q_{j,l})]
\end{aligned}$$

Theorem 5 immediately suggests a method to *compute* Δ and Δ'. Observe that the constraints on Δ and Δ' for different pairs of equivalent predicates form a set of linear inequalities. We can therefore use integer linear programming to maximize Δ values and minimize Δ' values to arrive at the tightest bounds on atom measures. If the system of inequalities for Δ (or Δ') is unsolvable, we can set the corresponding Δ (Δ') to $-\infty$ ($+\infty$).

For example, consider again the program in Figure 5. We have: $\Delta(p,q) \leq (\gamma_1 - \mu'_1) + \Delta(r,s)$; $\Delta(p,q) \leq (\gamma_2 - \mu'_2) + \Delta(p,q)$; and $\Delta(r,s) \leq \gamma_3 - \mu'_3$. Note that when $(\gamma_2 - \mu'_2) < 0$ the inequalities on $\Delta(p,q)$ are unsolvable. Otherwise, the non-recursive inequation gives the optimal value. Thus, applying minimization, we get $\Delta(p,q) = (\gamma_1 - \mu'_1) + (\gamma_3 - \mu'_3)$ if $(\gamma_2 - \mu'_2) \geq 0$, and $-\infty$ otherwise.

4 Automated Construction of Equivalence Tableau

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

Our framework specifies the order in which the different tableau rules and 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).

The tableau rules and associated transformations are applied in the following order. Given an e-atom $\Gamma \vdash \alpha \equiv \beta$, the proof is complete whenever the axiom elimination rule (**Ax**) is applicable. Hence, we first choose to apply **Ax**. When the choice is between the **Tx** and **Gen** rules, we choose the former since **Tx** corresponds to unfolding, i.e., resolution. This will ensure that our strategies will perform algorithmic verification for finite-state systems. For infinite-state systems, however, uncontrolled unfolding will diverge. To create finite unfolding sequences we impose the following finiteness condition *FIN* on transformation sequences:

Definition 6 (Finiteness condition) *A program transformation sequence* $\Gamma = \langle P_0, \ldots, P_i, \ldots \rangle$ *satisfies the finiteness condition FIN*(Γ) *if and only if the clause C and atom A selected for unfolding at* P_i*: (i) are distinct modulo variable renaming, and (ii) the term size of A is bounded a priori by a constant.*

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** that $\alpha \equiv \beta$ in P_{i+1} implies $\alpha \equiv \beta$ in P_i only if we can prove a new equivalence $\alpha' \equiv \beta'$ in P_0. Since **Gen** itself does not specify the goals 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. Applying goal replacement in P_0 under the assumption that that `q1(X)` $\equiv$ `q2(X)` enables the subsequent folding which transforms P_1 into P_2.

Hence, when no further unfoldings are possible, we look to do any possible folding. If no foldings are enabled, we check if there are new atom equivalences that will enable a folding step. For instance, in program P_0 of Figure 6, equivalence of `q1(X)` and `q2(X)` enables folding. Note that atom equivalences may be of the form $p(\overline{t}) \equiv q(\overline{s})$, where t and s are sequences of arbitrary *terms*, whereas the test for syntactic equivalence is only done on open atoms. We therefore introduce new definitions to convert them into open atoms. Finally, we look for new goal equivalences, which, if valid, can lead to syntactic equivalence. For instance, suppose in program P_2 (in Figure 6), there are two additional predicates

```
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(Y).
```

$\boldsymbol{P_0}$

```
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(Y).
```

$\boldsymbol{P_1}$

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

$\boldsymbol{P_2}$

Figure 6: Goal replacements to facilitate other transformations.

`p2` and `r2` and further assume that `p2` is defined using clauses `p2(a).` and `p2(f(Y)):-p2(Y),r2(X)`. Now if `r2` $\equiv$ `r1`, then we can perform this goal replacement to obtain the program P_3 and in P_3 we can conclude that `p1` $\overset{P_3}{\sim}$ `p2`. Herein also, an equivalence proof on arbitrary goals is first converted into equivalence between open atoms by introducing new definitions.

The above intuitive ideas are formally spelled out in Algorithm *Prove* (see Figure 7). Given a program transformation sequence Γ, and a pair A and B of open atoms, Algorithm *Prove* attempts to construct a proof for $A \equiv B$. If $A \equiv B$ is a subproof that needs to be done (because of a goal replacement step), the corresponding bounds on atom measures are needed. If *Prove* succeeds in finding a proof, it returns the bounds on the corresponding atom measures. Algorithm *Prove* uses a number of functions that are described below. The function *replace_and_prove* constructs proofs for sub-equivalences created by applying the **Gen** rule. Functions *unfold*(P) and *fold*(P) apply unfolding and folding transformations respectively to program P and return a new program; function *defn_intro*(C, P) adds a new definition clause C to P; and *goal_replace*$(P, (A, B), deltas)$ replaces atom A by B in P, using *deltas* as the bounds on atom measures.

Whenever conditional folding is possible, the function *new_atom_equiv_for_fold*(P) finds the pair of atoms whose replacement is necessary to do the fold operation. Similarly, when conditional equivalence is possible, *new_goal_equiv_for_equiv*(A, B, P) finds a pair of goals α, β such that syntactic equivalence of A and B can be established after replacing α with β in P.

We use *new_defn* to convert goal and atom equivalences into equivalences between open atoms. The function *new_defn*(α, Γ) finds and returns a clause C from some program P_j in Γ such that α is the body of C and the head of C does not unify with any other clause head in P_j. If no such C exists, then *new_defn* generates a new predicate symbol r not in Γ and returns the clause $(r(\overline{X}) :- \alpha)$, where $\overline{X} = vars(\alpha)$.

Deriving concrete strategies from *Prove*: Algorithm *Prove* searches nondeterministically for a proof: if multiple cases of 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.

Search for equivalence proofs can be controlled by the design of appropriate selection functions. For instance, when multiple unfold transformations are applicable, we only unfold those clauses *relevant* to A and B. The set of clauses relevant to an atom A, denoted by $\mathcal{R}(A)$, is the smallest set such that $C \in \mathcal{R}(A)$ if the head of C unifies either with A, or with some atom A' such that A' is in the body of a clause $D \in \mathcal{R}(A)$. (The examples described below use such a selection function.) It can be shown that this selection function is “lossless” in the sense that it preserves all equivalences that can be established by any strategy implemented using *Prove*. Moreover, selection functions can be used

```
algorithm $Prove(A, B$: open atoms, $\Gamma$:prog. seq.)
begin
  let $\Gamma = \langle P_0, \ldots, P_i \rangle$
  (* Ax rule *)
  if $(A = p(\overline{X}) \wedge B = q(\overline{X}) \wedge p \overset{P_i}{\sim} q)$ then
    return $(\Delta(p, q), \Delta'(p, q))$ (* page 9 *)
  else
    nondeterministic choice
      (* Tx rule *)
        case $FIN(\langle \Gamma, unfold(P_i) \rangle)$: (* Unfolding *)
          return $Prove(A, B, \langle \Gamma, unfold(P_i) \rangle)$
        case Folding is possible in $P_i$:
          return $Prove(A, B, \langle \Gamma, fold(P_i) \rangle)$
      (* Gen rule *)
        case Conditional folding is possible in $P_i$:
          let $(A', B') = new\_atom\_equiv\_for\_fold(P_i)$
          return $replace\_and\_prove(A, B, \langle A', B' \rangle, \Gamma)$
        case Conditional equivalence is possible in $P_i$:
          let $(\alpha, \beta) = new\_goal\_equiv\_for\_equiv(A, B, P_i)$
          return $replace\_and\_prove(A, B, \langle \alpha, \beta \rangle, \Gamma)$
    end choices
end
```

```
algorithm $replace\_and\_prove(A, B$: open atoms,
                         $\langle \alpha, \beta \rangle$:goals, $\Gamma$:prog. seq.)
begin
  let $\Gamma = \langle P_0, \ldots, P_i \rangle$
  if ($\alpha$ is not an open atom) then
    $C = new\_defn(\alpha, \Gamma)$
    return $Prove(A, B, \langle \Gamma, defn\_intro(C, P_i) \rangle)$
  else
    let $A' = \alpha$
  if ($\beta$ is not an open atom) then
    $D = new\_defn(\beta, \Gamma)$
    return $Prove(A, B, \langle \Gamma, defn\_intro(D, P_i) \rangle)$
  else
    let $B' = \beta$
  let $deltas = Prove(A', B', P_0)$
  let $P_{i+1} = goal\_replace(P_i, (A', B'), deltas)$
  return $Prove(A, B, \langle \Gamma, P_{i+1} \rangle)$
end
```

Figure 7: Algorithmic framework for automated construction of tableau

to implement *fair* searches in the space of proofs. For instance, using a breadth-first search, we can ensure that any finite equivalence proof derivable using *Prove* will eventually be found.

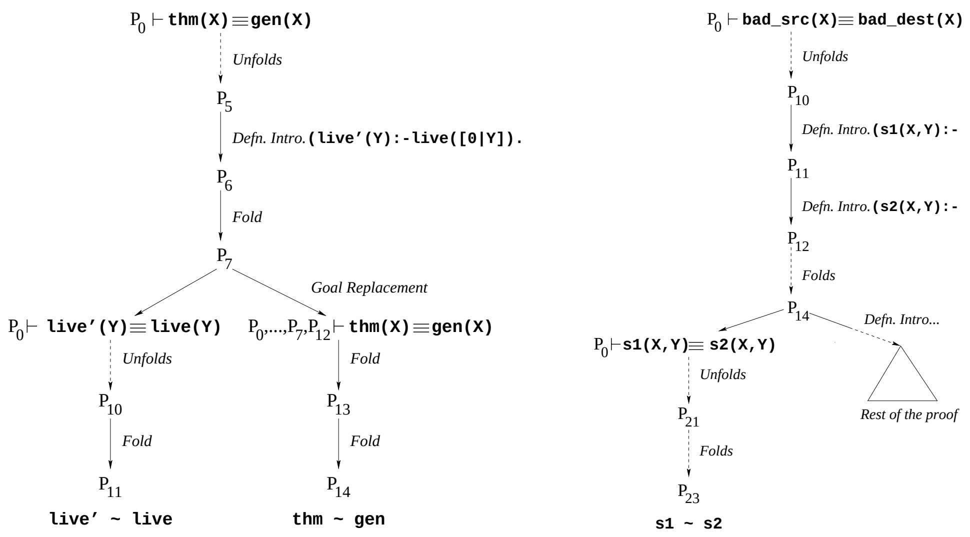
We use techniques from tabled resolution [TS84] to ensure that *Prove* does not repeatedly attempt to prove the same equivalences using the same transformation sequences. We do this by deeming a proof path as having failed if it attempts to add a program P to a transformation sequence Γ when $P \in \Gamma$. Moreover, note that when subproofs are generated due to conditional folding or equivalence, we start the transformation sequence from program P_0. Together with selection functions for unfolding sketched above, this ensures that only relevant transformation sequences are used in any equivalence proof.

Properties of *Prove*: It can be easily verified that only finite unfolding sequences satisfy *FIN*. In fact, the condition (ii) of Definition 6 can be relaxed to select norms other than term size (following work in termination analysis, see [DD94]). We use term size norm since it is easy to measure in practice.

Prove terminates as long as the number of definitions introduced (i.e., new predicate symbols added) is finite. This follows immediately from the fact that there are only a finite number of terms of bounded size (consistent with the requirement of *FIN*) if the set of predicate symbols is finite.

4.1 Liveness Properties in Chains: An Example

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



(a) Liveness Proof (b) Safety Proof

Figure 8: Fragments of Liveness and Safety Proofs using *Prove*.

Since `thm` $\overset{P_0}{\not\sim}$ `gen`, *Prove* unfolds clauses in P_0, and reaches program P_5 where `thm` is defined as:

```
thm([1]).                                          (3,3)
thm([0|X]) :- gen(X), X = [1|_].                   (5,5)
thm([0|X]) :- gen(X), trans(X,Y), live([0|Y]).     (4,4)
```

Further unfolding in P_5 will violate *FIN*. In addition no folding transformation is applicable at this stage. However, if `live([0|Y])` $\equiv$ `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 $\alpha =$ `live([0|Y])` and $\beta =$ `live(Y)`. Since `live([0|Y])` is not an open atom, a new definition of the form

```
live'(Y) :- live([0|Y]).                           (1,1)
```

is added to P_5 to yield P_6, and *Prove* is invoked again. Unconditional folding is applicable in P_6 in the third clause of `thm` above using the newly introduced clause as folder, generating P_7 where the third clause of `thm` is:

```
thm([0|X]) :- gen(X), trans(X,Y), live'(Y).        (3,3)
```

Conditional folding is applicable in P_7 and *replace_and_prove* is invoked, with $\alpha =$ `live'(Y)` and $\beta =$ `live(Y)`. Since these α and β are open atoms, we first invoke *Prove*(`live'(X)`, `live(X)`, $\langle P_0 \rangle$) (left branch in the tree in Figure 8a). Then using the bounds on atom measures returned by this call, we replace `live'(X)` with `live(X)` in the definition of `thm` in P_7 (right branch in Figure 8a).

Prove(`live'(X)`, `live(X)`, $\langle P_0 \rangle$) performs a series of unfoldings, yielding programs P_8, P_9 and P_{10}. (*FIN* prohibits any further unfolding in P_{10}.) In P_{10}, `live'` is defined by the following clauses:

```
live'([1|Z]).                                      (4,4)
live'(X) :- trans(X,Z), live([0|Z]).               (3,3)
```

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

```
live'([1|Z]).                                      (4,4)
live'(X) :- trans(X,Z), live'(Z).                  (2,2)
```

Now, `live'` $\overset{P_{11}}{\sim}$ `live` and hence *Prove*(`live'(X)`, `live(X)`, $\langle P_0 \rangle$) terminates. Also, $\Delta($`live'`, `live`$) = 4 - 1 = 3$, and $\Delta'($`live'`, `live`$) = \infty$; these bounds are returned by *Prove*.

Having completed the lemma, *replace_and_prove* replaces `live'(X)` with `live(X)` in the definition of `thm` in P_7, yielding P_{12} with:

```
thm([1]).                                          (3,3)
thm([0|X]) :- gen(X), X = [1|_].                   (5,5)
thm([0|X]) :- gen(X), trans(X,Y), live(Y).         (6,∞)
```

No further unfolding is possible without violating *FIN*. However, we can 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 we developed in [RKRR99b] was the first to permit such folding. Thus we obtain P_{13}:

```
thm([1]).                                          (3,3)
thm([0|X]) :- gen(X), live(X).                     (4,∞)
```

We can fold again using the definition of `thm` in P_0, giving P_{14} where `thm` is defined as:

```
thm([1]).                                          (3,3)
thm([0|X]) :- thm(X).                              (3,∞)
```

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

It is interesting to observe in Figure 8a that the unfolding steps that transform P_0 to P_5 and P_7 to P_{10} are interleaved with folding steps. This example thus illustrates how we interleave algorithmic model-checking steps with deduction steps.

4.2 Mutual exclusion in token rings

Algorithm *prove* generates a proof for mutual exclusion in a n-process token ring. The token ring is described by the following logic program:

```
gen([0,1]).                                  trans(X, Y) :- trans1(X, Y).
gen([0|X]) :- gen(X).                        trans([1|X], [0|Y]) :- trans2(X, Y).
trans1([0,1|T], [1,0|T]).                    trans2([0], [1]).
trans1([H|T], [H|T1]) :- trans1(T, T1).      trans2([H|X], [H|Y]) :- trans2(X, Y).
```

As in the case of chains, we represent the global state of a ring as a list of local states, by arbitrarily choosing an process in the ring as the first in the list. Processes with tokens are in local state 1 while processes without tokens are in state 0. `trans` is now divided into two parts: `trans1` which transfers the token to the left neighbor in the list, and `trans2` which transfers the token form the front of the list to the back, thereby completing the ring.

Mutual exclusion property is modeled using the predicates `bad`, `bad_start`, etc., as discussed in Section 2. These predicates, along with those listed above, form the initial program P_0. Recall that the safety proof can be completed by showing `bad_start` $\equiv$ `false` and `bad_src` $\equiv$ `bad_dest`. We now show snapshots of the proof generated by *prove* to demonstrate these equivalences.

An invocation of *prove*(`bad_start(X)`, `false`, $\langle P_0 \rangle$) performs a series of unfoldings and reaches a program P_3 where `bad_start` is defined by

```
bad_start([0|X]) :- gen(X), bad(X).   (3,3)
```

In P_3, *prove* folds `gen(X), bad(X)` using the original definition of `bad_start` to obtain P_4 with:

```
bad_start([0|X]) :- bad_start(X).   (2,2)
```

Since `bad_start` is defined by a single self-recursive clause, can be detected as failed, and hence `bad_start(X)` $\equiv$ `false`.

The proof of `bad_src(X)` $\equiv$ `bad_dest(X)` is illustrated in Figure 8b. We sketch some salient aspects of this proof below[2]. An invocation of *prove*(`bad_src(X)`, `bad_dest(X)`, $\langle P_0 \rangle$) performs a sequence of unfoldings, reaching a program P_{10} where the definitions of `bad_src` and `bad_dest` both expand to 6 clauses. Only conditional equivalence is applicable at P_{10}, generating 4 new goal equivalences, among them `trans1(X,Y), one_more_token(X)` $\equiv$ `trans1(X,Y), one_more_token(Y)`.

The replacement of these two goals is itself performed by *replace_and_prove*. Since the goals are not open atoms, *replace_and_prove* will create a new definition for each goal, reaching a program P_{12} containing clauses of the form:

```
s1(X, Y) :- trans1(X,Y), one_more_token(X)   (1,1)
s2(X, Y) :- trans1(X,Y), one_more_token(Y)   (1,1)
```

Note that, although the definitions of `s1` and `s2` are very similar, the variable bindings are different, and hence they are not syntactically equivalent. No new unfolding is applicable at P_{12}, and *prove* folds using the above two rules as folders into respective clauses of *bad_src* and *bad_dest*, to obtain P_{14}. It then invokes *prove*(`s1(X)`, `s2(X)`, $\langle P_0 \rangle$) as a subproof. This proof is completed after a sequence of unfoldings (to reach program P_{21}) and two foldings, yielding program P_{23} containing:

```
s1([0,1|X], [1,0|X]).                (4,4)      s2([0,1|X], [1,0|X]).                (3,3)
s1([1|X], [1|Y]) :- trans1(X, Y).    (3,3)      s2([1|X], [1|Y]) :- trans1(X, Y).    (3,3)
s1([H|X], [H|Y]) :- s1(X, Y).        (2,2)      s2([H|X], [H|Y]) :- s2(X, Y).        (2,2)
```

`s1` $\overset{P_{23}}{\sim}$ `s2` and hence `s1(X)` $\equiv$ `s2(X)`. Proofs of other goal equivalences generated by the conditional equivalence in program P_{10} proceed similarly, and are omitted.

5 Concluding Remarks

We have shown how the combination of tabled resolution (unfold with tabling) and certain logic-program transformations (fold and goal replacement) yields a general and highly automated framework for proving safety and liveness properties of parameterized systems.

A preliminary prototype implementation of our transformation system, built on top of our XSB tabled logic-programming system, has been completed. So far we have been able to automatically verify a number of examples including the ones described in this paper. Our plan now is to investigate the scalability of our system on more complex problems such as parameterized versions of the Rether protocol [DSC99] and the Gnu i-protocol [DDR$^+$99].

Towards that end we will need to enhance the proof strategies discussed in this paper. In particular, we are investigating techniques to make our proof search more "goal oriented" through the notion of a "preferred" form for the definition of an atom (or predicate). The criteria for the preferred form are so chosen that semantic equivalence of two atoms (or predicates) is more likely to be detected through syntactic equivalence if their definitions are in preferred form. Our proof strategies will be geared to transform atoms to such preferred forms.

References

[AH96] 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.

[2] The fragment of the proof corresponding to the figure appears in the appendix. For complete proof, see [RKR$^+$99].

[AK86] K. Apt and D. Kozen. Limits for automatic verification of finite-state systems. *Information Processing Letters*, 15:307–309, 1986.

[BCG89] M. Browne, E. Clarke, and O. Grumberg. Reasoning about networks with many identical finite-state processes. *Information and Computation*, 81(1):13–31, 1989.

[BCG95] G. S. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *LICS'95*, pages 388–397, San Diego, July 1995. IEEE Computer Society Press.

[CGJ95] E. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In Lee and Smolka [LS95].

[DD94] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19 & 20:199–260, May 1994.

[DDR$^+$99] Y. Dong, X. Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I.V. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, and D. S. Warren. Fighting livelock in the i-Protocol: A comparative study of verification tools. In *TACAS'99*, volume LNCS 1579. Springer-Verlag, March 1999.

[Dil96] D. L. Dill. The Murφ verification system. In Alur and Henzinger [AH96], pages 390–393.

[DP99] G. Delzanno and A. Podelski. Model checking in CLP. In *TACAS'99*, volume LNCS 1579, pages 74–88. Springer-Verlag, March 1999.

[DRS99] X. Du, C.R. Ramakrishnan, and S.A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. Technical report, Dept. of Computer Science, SUNY Stony Brook, 1999. URL: `http://www.cs.sunysb.edu/~vicdu/papers`.

[DSC99] X. Du, S. A. Smolka, and R. Cleaveland. Local model checking and protocol analysis. *Software Tools for Technology Transfer*, 1999.

[EN95] E. Emerson and K.S. Namjoshi. Reasoning about rings. In *Proceedings of 22nd POPL*, pages 85–94, 1995.

[Gru97] O. Grumberg, editor. *Computer Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, Haifa, Israel, June 1997. Springer-Verlag.

[GS96] S. Graf and H. Saidi. Verifying invariants using theorem proving. In Alur and Henzinger [AH96].

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[ID96] C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. In Alur and Henzinger [AH96], pages 147–158.

[KM95] R.P. Kurshan and K. Mcmillan. A structural induction theorem for processes. *Information and Computation*, 117:1–11, 1995.

[KMM$^+$97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In Grumberg [Gru97], pages 424–435.

[LHR97] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parametrized linear networks of processes. In *Proceedings of POPL 97*, pages 346–357, 1997.

[Llo93] J.W. Lloyd. *Foundations of Logic Programming, Second Edition.* Springer-Verlag, 1993.

[LS95] I. Lee and S. A. Smolka, editors. *CONCUR '95*, volume 962 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.

[Mil89] R. Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, 1989.

[OSR92] S. Owre, N. Shankar, and J. Rushby. PVS: A Prototype Verification System. *Proceedings of CADE*, 1992.

[PP99] A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. *Journal of Logic Programming, to appear*, 1999.

[RKR$^+$99] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, and S.A. Smolka. Verification of parameterized systems using logic program transformations. Technical report, Dept. of Computer Science, SUNY Stony Brook, 1999. URL: `http://www.cs.sunysb.edu/∼abhik/papers/`.

[RKRR99a] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. In *Asian Computer Science Conference (ASIAN)*, 1999. To appear.

[RKRR99b] A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. A parameterized unfold/fold transformation framework for definite logic programs. In *Principles and Practice of Declarative Programming (PPDP), LNCS 1702*, pages 396–413, 1999.

[RRR$^+$97] 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 Grumberg [Gru97].

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Computer Aided Verification (CAV '95)*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liége, Belgium, July 1995. Springer-Verlag.

[SCK$^+$95] B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In Lee and Smolka [LS95], pages 72–87.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of International Conference on Logic Programming*, pages 127–138, 1984.

[UR95] L. Urbina and G. Riedewald. Simulation and verification in constraint logic programming. In *2nd European Workshop on Real Time and Hybrid Systems*. Springer-Verlag, 1995.

[Urb96] L. Urbina. Analysis of hybrid systems in CLP(R). In *Constraint Programming (CP'96)*, volume LNCS 1102. Springer-Verlag, 1996.

[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 66–80, Grenoble, June 1989. Springer-Verlag.

[XSB99] XSB. The XSB logic programming system v2.01, 1999. Available by anonymous ftp from `www.cs.sunysb.edu/∼sbprolog`.

APPENDIX

A Proof of mutual exclusion in a n-process token ring

An invocation of *prove*(`bad_src(X)`, `bad_dest(X)`, $\langle P_0 \rangle$) performs a sequence of unfoldings (as long as the finiteness conditions hold) and reaches a program P_{10} containing the following clauses:

```
bad_src([0,1,1|X], [1,0,1|X]).                                (6,6)
bad_src([0,1,H|T], [1,0,H|T]) :- one_more_token(T).           (6,6)
bad_src([1|X],[1|Y]) :- trans1(X,Y), one_more_token(X).       (4,4)
bad_src([H|X],[H|Y]) :- trans1(X,Y), bad(X).                  (4,4)
bad_src([1,1|X],[0,1|Y]) :- trans2(X,Y).                      (5,5)
bad_src([1,H|X],[0,H|Y]) :- trans2(X,Y), one_more_token(X).   (5,5)
bad_dest([0,1,1|X], [1,0,1|X]).                               (6,6)
bad_dest([0,1,H|T], [1,0,H|T]) :- one_more_token(T).          (6,6)
bad_dest([1|X],[1|Y]) :- trans1(X,Y), one_more_token(Y).      (4,4)
bad_dest([H|X],[H|Y]) :- trans1(X,Y), bad(Y).                 (4,4)
bad_dest([1,1|X],[0,1|Y]) :- trans2(X,Y), one_more_token(Y).  (5,5)
bad_dest([1,H|X],[0,H|Y]) :- trans2(X,Y), bad(Y).             (5,5)
```

Conditional equivalence alone is applicable in P_{10}, creating new goal equivalences between the right hand sides of rules for `bad_src` and `bad_dest`. Here, we consider one of the new equivalences: `trans1(X,Y), one_more_token(X)` $\equiv$ `trans1(X,Y), one_more_token(Y)`. The replacement of these two goals is itself performed by *replace_and_prove*. Since the goals are not open atoms, *replace_and_prove* will create a new definition for each goal, reaching a program P_{12} containing clauses of the form:

```
s1(X, Y) :- trans1(X,Y), one_more_token(X)   (1,1)
s2(X, Y) :- trans1(X,Y), one_more_token(Y)   (1,1)
```

No new unfolding is applicable at P_{12}, and *prove*. It first folds using the above two rules as folders into respective clauses of *bad_src* and *bad_dest* respectively, yielding P_{14}. It then invokes *prove*(`s1(X)`, `s2(X)`, $\langle P_0 \rangle$). This invocation reaches a program P_{21} containing:

```
s1([0,1|X], [1,0|X]).                               (4,4)
s1([1|X], [1|Y]) :- trans1(X, Y).                   (3,3)
s1([H|X], [H|Y]) :- trans1(X, Y), one_more_token(X).  (3,3)
s2([0,1|X], [1,0|X]).                               (3,3)
s2([1|X], [1|Y]) :- trans1(X, Y).                   (3,3)
s2([H|X], [H|Y]) :- trans1(X, Y), one_more_token(Y).  (3,3)
```

No further unfolding is possible in P_{21}, but folding using the definitions of `s1` and `s2`, we reach a program P_{23} containing:

```
s1([0,1|X], [1,0|X]).              (4,4)
s1([1|X], [1|Y]) :- trans1(X, Y).  (3,3)
s1([H|X], [H|Y]) :- s1(X, Y).      (2,2)
s2([0,1|X], [1,0|X]).              (3,3)
s2([1|X], [1|Y]) :- trans1(X, Y).  (3,3)
s2([H|X], [H|Y]) :- s2(X, Y).      (2,2)
```

Now `s1` $\overset{P_{23}}{\sim}$ `s2` and hence `s1(X)` $\equiv$ `s2(X)`. Proofs of other goal equivalences generated by the conditional equivalence in program P_{10} proceed similarly, and are omitted.