Verification of Parameterized Systems Using Logic Program Transformations*

Abhik Roychoudhury[1], K. Narayan Kumar[2], C.R. Ramakrishnan[1], I.V. Ramakrishnan[1], and Scott A. 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 deductive steps, 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.

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 [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 in the standard Spin distribution.

More recently, we have been investigating how XMC's model-checking capabilities can be extended *beyond finite-state systems*. Essentially, this can be

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

done by enhancing the underlying resolution strategy appropriately 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. In general, the verification of parameterized systems lies beyond the reach of traditional model checkers and it is not at all trivial (or even possible) to adapt them to verify parameterized systems.

The main idea underlying our approach is to reduce the problem of verifying parameterized systems to one of determining equivalence of goals in a logic program. We then establish goal equivalences by inducting on the size of proofs of ground instances of goals. 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 new transformation system that allows folding using multiple recursive clauses, which seems essential 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, on-the-fly model checking performed in XMC. Unfold transformations are used to evaluate away the base case and the finite portions of the induction step in an induction proof. 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 a clause right-hand side with a semantically equivalent goal, represent *deductive* reasoning. They are used to simplify a given program so that applications of the induction hypothesis in the induction proof can be recognized.

Using our approach, we have been able to prove liveness and safety properties of a number of parameterized systems. 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 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 in the tabled logic programming framework.
- *Tighter integration of algorithmic and deductive model checking.* Unfold, fold, and goal-replacement steps can be arbitrarily interleaved within the verification proof of a parameterized system. Thus our approach allows algorithmic

model checking computation (unfold) to be integrated with deductive reasoning (fold, goal replacement) at fine levels of granularity. Also, since deductive steps are applied lazily in our approach, finite-state model checking emerges as a special case of verifying parameterized systems.

- *High degree of automation.* Although a fully automated solution to verification of parameterized systems is not possible, for many cases of practical interest, we have identified certain heuristics that can be applied to our proof system in order to completely automate the deduction involved.

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 in logic program transformations with more powerful transformation rules and strategies that are central to verification of parameterized systems. Note that our transformation rules are also applicable for proving general program properties.

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. [BCG89,EN95,ID99] reduce the problem of verifying a parameterized system to the verification of an "equivalent" finite-state system. [WL89,KM95,LHR97] 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.

Perhaps the work most closely related to our own involves the use of theorem provers for verifying parameterized systems. Rajan et al. [RSS95] have incorporated a mu-calculus model checker 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] combine a custom-built specification/deduction system with PVS to formalize and verify invariant properties of infinite-state systems.

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 folding and goal-replacement transformations. In our approach, 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 Parameterized System Verification as Goal Equivalence

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

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

System description

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

Property description

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

Modeling Parameterized 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 (a list in Prolog-like notation is of the form `[Head|Tail]`) 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.
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 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.

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], for simplicity of exposition we present here an alternative formulation without negation. 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(Xs).
bad([_|Xs]) :- bad(Xs).

one_more([1|_]).
one_more([_|Xs]) :- one_more(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.

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 XMC is guaranteed to terminate only for finite-state systems). The above simplified presentation (which we will continue to use in the rest of this paper) is used to prevent a proliferation of syntax.

3 Goal Equivalence Proofs using Tableau

In this section we describe the basic framework to construct such equivalence proofs. We begin by defining the relevant notations.

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 (`true` and `false` are special atoms), 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 *definite logic program*, which is a set of Horn clauses. A Horn clause C is written as $A :- B_1, B_2, \ldots, B_n$. A, the consequent, is 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, $M(P)$. 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 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 program transformation sequence where P_{i+1} is obtained from P_i by applying a 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, 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 proof system is analogous to SLD resolution. 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 α and β are semantically equivalent in any program in Γ. An *e-goal* is a (possibly empty) sequence of e-atoms (e-atoms and e-goals correspond to atoms and goals in standard resolution).

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

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

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

Fig. 2. Rules for constructing equivalence tableau.

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 by $\alpha \cong \beta$. Axiom elimination is akin to the treatment of facts in SLD resolution. The *program transformation rule* (**Tx**) attempts to simplify a program in order to expose the equivalence of goals. We use this rule when we apply a (semantics-preserving) transformation that does not add any equivalence proof obligations *e.g.* unfolding, folding. The *sub-equivalence generation rule* (**Gen**) replaces an e-atom with new e-atoms which are (hopefully) simpler to establish. This step is akin to standard SLD resolution step. 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 $E_0, E_1, \ldots, E_n$ where E_{i+1} is obtained from E_i by applying **Ax**/**Tx**/**Gen** and E_n is empty.

Theorem 1 *Let $E_0, E_1 \ldots, E_n$ be a successful tableau, P_0 be a (definite) logic program and $E_0 = \langle P_0 \rangle \vdash \alpha \equiv \beta$. For all ground substitutions θ, $\alpha\theta \in M(P_0) \Leftrightarrow \beta\theta \in M(P_0)$, i.e. 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 (which can be proved using a reduction in [AK86]).

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

3.2 Program Transformations

The **Tx** and **Gen** rules of our proof system require us to transform a program P_i into a program P_{i+1}. This is accomplished by applying logic program transformations that include unfolding, folding, goal replacement and definition introduction.

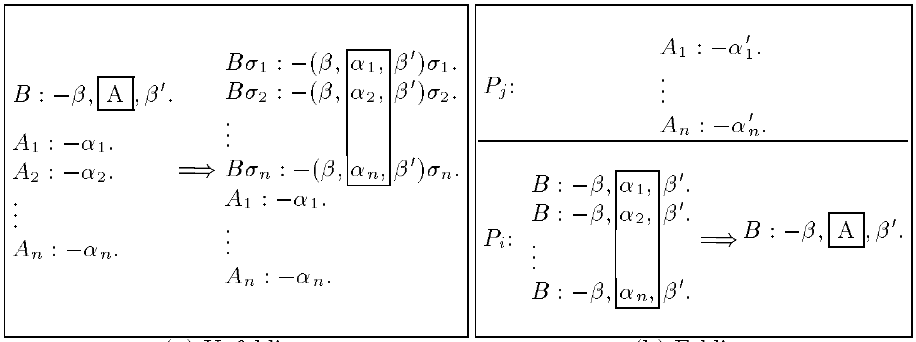
For a simple illustration of program transformations, consider Figure 3. There, program P_1 is derived from P_0 by *unfolding* the occurrence of `r` in the definition of `q`. P_2 is derived from P_1 by *folding* `t,s` in the definition of `p` using the definition of `q`. While unfolding is semantics preserving, indiscriminate folding may introduce circularity, thereby removing finite proof paths. *e.g.* folding `t,s` in the definition of `q` in P_2 using the definition of `p` in P_0 results in a program `p :- q. q :- p. r :- t. ...`. This removes `p` and `q` from the least model.

Program P_0	Program P_1	Program P_2
`p :- t, s.`	`p :- t, s.`	`p :- q.`
`q :- r, s.`	`q :- t, s.`	`q :- t, s.`
`r :- t.`	`r :- t.`	`r :- t.`
`...`	`...`	`...`

Fig. 3. Example of an unfold/fold transformation sequence.

We now present the program transformations informally. For a formal description, the reader is referred to [RKRR99b]. With each clause C in program P_i of the transformation sequence, we associate a pair of integer counters that bound the size of a shortest proof of any ground atom A derived using C in program P_i relative to the size of a shortest proof of A in P_0. Thus the counters keep track of potential reductions in proof lengths. Conditions on counters are then used to determine if a given application of folding is semantics preserving.

Unfolding of an atom A in the body of a clause in P_i is shown in Figure 4a. The conditions for applying the transformation are : (i) $A_1, \ldots, A_n$ are the only clause heads in P_i which unify with A, and (ii) σ_j is the mgu of A and A_j for all $1 \leq j \leq n$. Note that these conditions are taken directly from resolution, which means that unfolding is essentially a resolution step.



(a) Unfolding (b) Folding

Fig. 4. Schema for unfold/fold transformations.

Folding 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 $P_j (j \leq i)$ in the transformation sequence. The conditions for applying the transformation are[1]: (i) α_l is an instance of α'_l with substitution σ_l for all $1 \leq l \leq n$ (ii) there is an atom A such that $\forall 1 \leq l \leq n \; A_l \sigma_l = A$ and the folder clauses are the only clauses in P_j whose heads unify with A.

Goal replacement replaces an atom B in a clause $A :- \alpha, B\beta$ in program P_i with a semantically equivalent atom B' to obtain the clause $A :- \alpha, B', \beta$. Note that such a replacement can change lengths of proofs of A arbitrarily. To obtain the counters associated with the new clause we need to estimate the changes in proof lengths. In practice, we do so by using techniques based on Integer Linear Programming. Details appear in [Roy99].

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. Then $M(P_i) = M(P_0)$, $1 \leq i \leq N$.*

Definition-introduction transformation adds clauses defining a new predicate to a program P_i. This transformation is used to generate "names" for goals. Note that after definition introduction, $M(P_{i+1}) \neq M(P_i)$ since a new predicate is added to 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.

[1] In addition, certain other conditions need to be imposed including conditions on the counters of the folder and folded clauses; we do not mention them here.

3.3 Checking Goal Equivalence from Syntax

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 $q(\overline{X})$.

```
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).
```

Consider the example program given above. We can infer that `r(X)` $\equiv$ `s(X)` since `r` and `s` have identical definitions. Then, we can infer `q(X)` $\equiv$ `p(X)`, since their definitions are "isomorphic". Formally:

Definition 1 (Syntactic Equivalence) *A syntactic equivalence relation, $\overset{P}{\sim}$, is an equivalence relation on the set of predicates of a program P such that for all predicates p, q in P, if $p \overset{P}{\sim} q$ then:*
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 $f : \{1, \ldots, m\} \to \{1, \ldots, n\}$ and $g : \{1, \ldots, n\} \to \{1, \ldots, m\}$ such that:
(a) $\forall 1 \le i \le m$ C'_i is an instance of $D'_{f(i)}$, and
(b) $\forall 1 \le j \le n$ D'_j is an instance of $C'_{g(j)}$.

The largest syntactic equivalence relation can be computed by starting with all predicates in the same class, and repeatedly splitting the classes until a fixed point is reached. Syntactic equivalence is sound w.r.t. semantic equivalence, *i.e.*

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 $p(\overline{X}) \equiv q(\overline{X})$.*

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 limited user intervention, and (b) apply deduction rules lazily, i.e. a proof using the strategy is equivalent to algorithmic verification for finite-state systems.

In our framework, 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** allows unfolding, i.e. resolution. This will ensure that our

```
algorithm Prove(A, B: open atoms, Γ:prog. seq.)
begin
  let Γ = ⟨P_0, ..., P_i⟩
  (* Ax rule *)
  if (A = p(X̄) ∧ B = q(X̄) ∧ p ~^{P_i} q) then
    return true
  else nondeterministic choice
      (* Tx rule *)
        case FIN(⟨Γ, unfold(P_i)⟩): (* Unfolding *)
          return Prove(A, B, ⟨Γ, unfold(P_i)⟩)
        case Folding is possible in P_i:
          return Prove(A, B, ⟨Γ, fold(P_i)⟩)
      (* 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, ⟨A', B'⟩, Γ)
        case Conditional equivalence is possible in P_i:
          let (α, β) = new_goal_equiv_for_equiv(A, B, P_i)
          return replace_and_prove(A, B, ⟨α, β⟩, Γ)
    end choices
end
```

Fig. 5. Algorithmic framework for automated construction of tableau.

strategies will perform algorithmic verification, a' la XMC, for finite-state systems. For infinite-state systems, however, uncontrolled unfolding will diverge. To create finite unfolding sequences we impose the finiteness condition *FIN* in Definition 2. 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.

Definition 2 (Finiteness condition) *An unfolding transformation sequence* $\Gamma = \langle P_0, \ldots, P_i, \ldots \rangle$ *satisfies* $FIN(\Gamma)$ *if and only if for the clause* C *and atom* A *selected for unfolding at* P_i*: (i)* 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 *(ii) the term size of* A *is bounded a-priori by a constant.*

Hence, when no further unfoldings are possible, we apply any possible folding. If no foldings are enabled, we check if there are new atom equivalences that will enable a folding step. We call this a *conditional folding* step. 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. This is called as a *conditional equivalence* step. In such a step, an equivalence proof on arbitrary goals is first converted into equivalence between open atoms by introducing new definitions.

The above intuitions are formalized in Algorithm *Prove* (see Figure 5). Given a program transformation sequence Γ, and a pair of open atoms A, B, algorithm *Prove* attempts to prove that $\Gamma \vdash A \equiv B$. Algorithm *Prove* uses the following functions. Function *replace_and_prove* constructs proofs for sub-equivalences created by applying the **Gen** rule. *replace_and_prove*$(A, B, \langle\alpha, \beta\rangle, \Gamma)$ first introduces definitions for α and β, then proves the equivalence $\langle P_0 \rangle \vdash \alpha \equiv \beta$ by invoking *Prove*, then replaces α by β and finally invokes *Prove* to complete the proof of $\Gamma \vdash A \equiv B$. 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_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 s.t. syntactic equivalence of A and B can be established after replacing α with β in P.

Note that *Prove* terminates as long as the number of definitions introduced (i.e., new predicate symbols added) is finite. If multiple cases of the nondeterministic choice are enabled, then *Prove* tries them in the order specified in Figure 5. 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. By providing selection functions to pick from the applicable transformations, one can implement concrete strategies from *Prove*. Details appear in [Roy99].

4.1 Example: Liveness Property in Chains

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. To establish the liveness property, we prove `thm(X)` $\equiv$ `gen(X)` by invoking *Prove*(`thm(X)`, `gen(X)`, $\langle P_0 \rangle$). The proof tree is illustrated in Figure 6 (dashed arrows in the figure denote multiple applications of the transformation annotating the arrow). *Prove* first unfolds the clauses of `thm` to obtain:

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

Since no unfolding or folding is applicable, conditional folding is done giving rise to the (sub)-equivalence `live([0|Y])` $\equiv$ `live(Y)`. Since `live([0|Y])` is not an open atom, a new definition `live'(Y) :- live([0|Y])` is added to P_5 to yield P_6. Then *Prove* folds the third clause of `thm` using this definition and recursively invokes *Prove*(`live'(X)`, `live(X)`, $\langle P_0 \rangle$) to establish `live'(X)` $\equiv$ `live(X)`. This

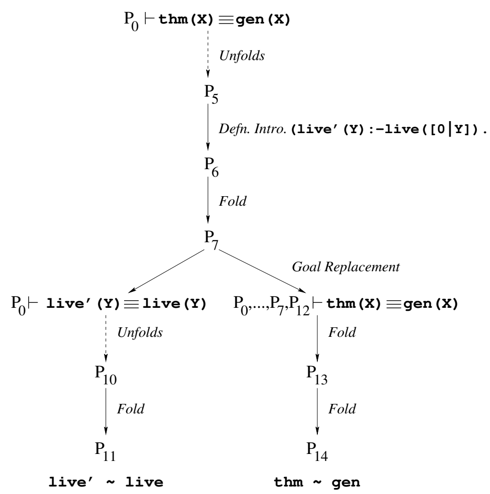


Fig. 6. Proof tree for liveness property in chains.

subproof appears in the left branch of Figure 6. Finally, *Prove* replaces `live'(X)` with `live(X)` in the clauses of `thm` and completes the proof of `thm(X)` $\equiv$ `gen(X)` by applying two folding steps.

It is interesting to observe in Figure 6 that the unfolding steps that transform P_0 to P_5 and P_7 to P_{10} are interleaved with folding steps. This illustrates how we interleave algorithmic model-checking steps with deduction steps.

4.2 Example: 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 (see Section 2), we represent the global state of a ring as a list of local states. 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, a safety 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 a safety proof can be completed by showing `bad_start` $\equiv$ `false` and `bad_src` $\equiv$ `bad_dest`. Figure 7 illustrates the proofs generated by *Prove* to demonstrate these equivalences.

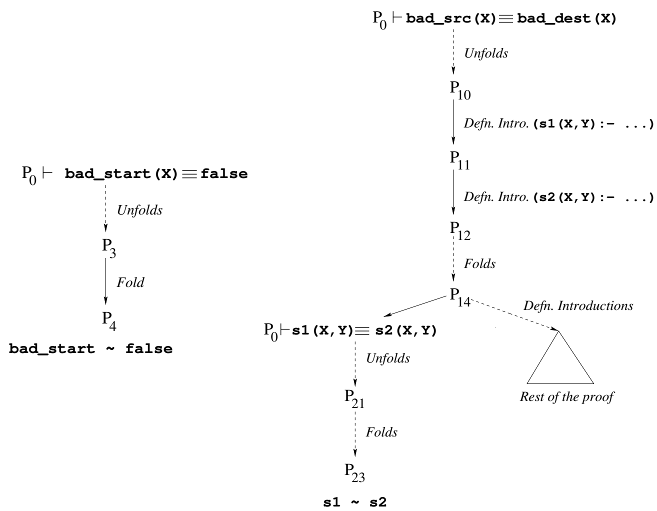


Fig. 7. Proof trees for mutual exclusion in token rings.

Invocation of $\mathit{Prove}($`bad_start(X)`$,$ `false`$, \langle P_0\rangle)$ performs unfoldings to obtain program P_3 where `bad_start` is defined using a single clause, namely: `bad_start([0|X]) :- gen(X), bad(X).` *Prove* now folds using the original definition of `bad_start` to obtain P_4 where `bad_start` is defined by the clause: `bad_start([0|X]) :- bad_start(X).` Since `bad_start` is defined by a single self-recursive clause, it is detected as failed, and hence `bad_start` $\equiv$ `false`.

An invocation of $\mathit{Prove}($`bad_src(X)`$,$ `bad_dest(X)`$, \langle P_0\rangle)$ performs unfoldings, to get program P_{10} where the definitions of `bad_src` and `bad_dest` are:

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

Now, to show `bad_src` $\equiv$ `bad_dest`, *Prove* applies conditional equivalence steps, generating the following (sub)-equivalences:

`trans1(X,Y), one_more(X)` $\equiv$ `trans1(X,Y), one_more(Y)`
`trans1(X,Y), bad(X)` $\equiv$ `trans1(X,Y), bad(Y)`
`trans2(X,Y), one_more(Y)` $\equiv$ `trans2(X,Y)`
`trans2(X,Y), one_more(X)` $\equiv$ `trans2(X,Y), bad(Y)`

We now show the proof of the first of the above. Proofs of the other three (sub)-equivalences proceed similarly, and are omitted. Since the goals are not open atoms, the following definitions are created to obtain program P_{12}.

```
s1(X, Y) :- trans1(X,Y), one_more(X).
s2(X, Y) :- trans1(X,Y), one_more(Y).
```

Since no new unfolding is applicable at P_{12}, the clauses of `bad_src` and `bad_dest` are folded using the above two clauses to obtain P_{14}. $\mathit{Prove}(\texttt{s1(X)}, \texttt{s2(X)}, \langle P_0 \rangle)$ is then invoked by *Prove* as a subproof. This subproof is completed after a sequence of unfoldings (to reach program P_{21}) and two foldings, yielding P_{23}:

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

$\texttt{s1} \stackrel{P_{23}}{\sim} \texttt{s2}$ and hence $\texttt{s1(X)} \equiv \texttt{s2(X)}$.

5 Concluding Remarks

A preliminary prototype implementation of our transformation system, built on top of our XSB tabled logic-programming system [XSB99], 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 Java meta-locking protocol [BSW00].

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.

[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.

[BSW00] S. Basu, S.A. Smolka, and O.R. Ward. Model checking the Java meta-locking algorithm. In *IEEE International Conference on the Engineering of Computer Based Systems*. IEEE Press, April 2000.

[CGJ95] E. Clarke, O. Grumberg, and S. Jha. Verifying parametrized networks using abstraction and regular languages. In *CONCUR, LNCS 962*, 1995.

[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, `http://www.cs.sunysb.edu/~vicdu/papers`, 1999.

[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 *POPL*, pages 85–94, 1995.

[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.

[ID99] C. N. Ip and D. L. Dill. Verifying systems with replicated components in Murφ. *Formal Methods in System Design*, 14(3), May 1999.

[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 *CAV, LNCS 1254*, 1997.

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

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

[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*, 41, 1999.

[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, LNCS 1742*, pages 322–333, 1999.

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

[Roy99] A. Roychoudhury. Program transformations for automated verification of parameterized concurrent systems. Technical report, Department of Computer Science, State University of New York at Stony Brook, `http://www.cs.sunysb.edu/~abhik/papers`, 1999. Dissertation proposal.

[RRR+97] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient model checking using tabled resolution. In *CAV, LNCS 1254*, 1997.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In *CAV, LNCS 939*, 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 *Automatic Verification Methods for Finite State Systems, LNCS 407*, 1989.

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