Inductively Verifying Invariant Properties of Parameterized Systems *

Abhik Roychoudhury (Contact Author) (abhik@comp.nus.edu.sg)
School of Computing, National University of Singapore, 3 Science Drive 2, Singapore 117543.

I.V. Ramakrishnan (ram@cs.sunysb.edu)
Department of Computer Science, State University of New York at Stony Brook, Stony Brook, NY 11794, USA.

Abstract.
Verification of distributed algorithms can be naturally cast as verifying parameterized systems, the parameter being the number of processes. In general, a parameterized concurrent system represents an infinite family (of finite state systems) parameterized by a recursively defined type such as chains, trees. It is therefore natural to verify parameterized systems by inducting over this type. However, construction of such proofs require combination of model checking with deductive capability. In this paper, we develop a logic program transformation based proof methodology which achieves this combination. One of our transformations (unfolding) represents a single resolution step. Thus model checking can be achieved by repeated application of unfolding. Other transformations (such as folding) represent deductive reasoning and help recognize the induction hypothesis in an inductive proof. Moreover the unfolding and folding transformations can be arbitrarily interleaved in a proof, resulting in a tight integration of decision procedures (such as model checking) with deductive verification.

Based on this technique, we have designed and implemented an invariant prover for parameterized systems. Our proof technique is geared to automate nested induction proofs which do not involve strengthening of induction hypothesis. The prover has been used to automatically verify invariant properties of parameterized cache coherence protocols, including broadcast protocols and protocols with global conditions. Furthermore, we have employed the prover for automatic verification of mutual exclusion in the Java Meta-Locking Algorithm. Meta-Locking is a distributed algorithm developed recently by designers in Sun Microsystems for ensuring secure access of Java objects by an arbitrary number of Java threads.

Keywords.
Parameterized Systems, Concurrent Systems, Induction proofs, Program Transformations, Logic Programming, Unfolding, Folding.

1. Introduction

There is a growing interest in verification of *parameterized* concurrent systems since they occur widely in computing *e.g.* in distributed algo-

* A preliminary version of this paper appeared in Proceedings of International Conference on Computer Aided Verification (CAV) 2001, LNCS 2102 [58].

rithms. Intuitively, a parameterized system is an infinite family of finite state systems parameterized by a recursively defined type *e.g.* chains, trees. Verification of distributed algorithms (with arbitrary number of constituent processes) can be naturally cast as verifying parameterized systems. For example, consider a distributed algorithm where n users share a resource and follow some protocol to ensure mutually exclusive access. Model checking [13, 40, 51] can verify mutual exclusion for only finite instances of the algorithm, *i.e.* for $n = 3$, $n = 4, \ldots$ but not for any n.

In general, automated verification of parameterized systems has been shown to be undecidable [3]. Thus, verification of parameterized networks is often accomplished via theorem proving [15, 35, 45], or by synthesizing network invariants [14, 37, 64]. A network invariant is a finite state process verifying which is sufficient for verifying the entire parameterized family. Alternatively, one can identify subclasses of parameterized systems for which verification is decidable [22, 23, 26, 30]. Another approach [18, 24, 25, 31, 36] finitely represents the state space of a parameterized system and applies (symbolic) model checking over this finite representation. A recently proposed approach [49] combines assertion synthesis techniques with abstraction for verification of parameterized systems.

Since a parameterized system represents an infinite family parameterized by a recursively defined type, it is natural to prove properties of parameterized systems by inducting over this type. In a recent paper [56] we outlined a methodology for constructing such proofs by suitably extending the resolution based evaluation mechanism of logic programs. In our approach, the parameterized system and the property to be verified are encoded as a logic program. The verification problem is reduced to the problem of determining relationships among predicates in this program such as $p \Rightarrow q$ where p, q are program predicates. The predicate implications are established by *transforming* the predicates p and q. The proof of $p \Rightarrow q$ then proceeds automatically by a routine induction on the structure of these transformed definitions.

For transforming predicates, we use the well-established unfold/fold transformations of logic programs [46, 55, 57, 59, 61]. These transformations have been previously used for program optimization [7, 39, 48] and automated deduction [29, 32, 47]. One of these transformations (unfolding) represents an application of resolution. In particular, an application of the unfold transformation represents a single resolution step. Therefore, one can achieve on-the-fly explicit state *algorithmic* model checking by repeated unfolding of the verification proof obligation. In constructing induction proofs, unfold transformations are used to evaluate away the base case and the finite portions of the induction step. Folding on the other hand, represents a form of *deductive* reasoning. It is used to simplify the given program so that applications of the induction hypothesis in the induction proof can be recognized.

Since one of our transformations corresponds to a step in model checking and the others correspond to deductive reasoning, model checking emerges as a special case when the deductive steps are applied lazily. Note that application of unfolding and folding steps can be arbitrarily interleaved in the verification proof of a parameterized system. This allows our framework to tightly integrate algorithmic and deductive verification (as compared to integrating model checking as a decision procedure in a theorem prover).

The work in this paper is part of recent efforts to exploit logic programming technology for developing new tools and techniques to specify and verify concurrent systems. For example, constraint logic programming has been used for the analysis and verification of hybrid systems [62, 63] and more recently for model checking infinite-state systems [19]. In [53], the memoized logic programming system XSB [65] is used to develop the XMC model checker. XMC is an efficient and flexible model checker whose timings are comparable to that of highly optimized model checkers such as Spin [28] and Murφ [20]. Similar work on using logic program evaluation for model checking has also been reported in [44]. Recently, [18] used constraint logic programming to construct uniform proofs of safety properties of parameterized cache coherence protocols. Essentially, these techniques aim to use (constraint) logic program evaluation to efficiently construct verification proofs involving state space search (accomplished via resolution) and (possibly) constraint solving. These techniques are in general not suitable for constructing induction proofs arising in the verification of parameterized systems.

Summary of Contributions In this paper, we employ our logic program transformation based approach for inductive verification of real-life parameterized protocols. The specific contributions are:

1. First, we show how logic program evaluation based techniques for verifying finite state systems [53] can be flexibly extended to yield a program transformation framework for constructing inductive proofs of invariant properties of parameterized systems. One of our transformations corresponds to a resolution step and the others correspond to deductive reasoning. Since model checking can be accomplished by memoized resolution [53], model checking emerges as a special case when the deductive steps are applied lazily.

2. We clarify the connection between our transformations and inductive reasoning. In particular, repeated application of the unfolding transformation can achieve fully automated proof procedures such as model checking. The folding transformation is useful for deduction, in particular, uncovering the induction hypothesis in an induction proof.

3. Our technique is not restricted to specific network topologies. We have verified chain, ring, tree, star and complete graph networks. Furthermore by enriching the underlying language to Constraint Logic Programming (CLP), the technique can be extended to verify infinite families of infinite state systems such as parameterized real-time systems.

4. We construct an automatic and programmable first order logic based prover with limited deductive capability. This prover has been implemented on top of the XSB logic programming system [65]. It has been used to automatically verify safety properties of real-life parameterized protocols. The prover can also exploit knowledge of network topology (chain, tree etc) to facilitate convergence of proofs.

5. Besides verifying parameterized cache coherence protocols such as Berkeley RISC and Illinois, we also report the verification of mutual exclusion in Java meta-locking algorithm. It is a real-life distributed algorithm recently developed by Sun Microsystems to ensure mutual exclusion in accessing Java objects by an arbitrary number of Java threads. Previously, the designers of the protocol gave an informal correctness argument [1], and model checking of instances of the protocol were done [5]. This is the first mechanized proof of the algorithm which is parameterized by the number of threads.

Organization of the paper The rest of the paper is organized as follows. Section 2 clarifies the connection between program transformations and inductive reasoning. Section 3 presents our methodology for verifying invariant properties parameterized systems via program transformations along with an example (verifying mutual exclusion in an n process token ring). Section 4 discusses the strategies required to guide the program transformations. This achieves automated verification of parameterized protocols. Section 5 presents experimental results obtained by using our prover on various parameterized protocols *e.g.* cache coherence protocols. Section 6 presents a detailed case study: the use of our prover in verifying the Java meta-locking algorithm from Sun

Microsystems. Section 7 discusses related work in parameterized system verification and Section 8 presents concluding remarks.

2. Program Transformations for Induction

In this section, we explain the intuition behind the use of logic program transformations for induction. For this purpose we need to use the logic programming notions of terms, atoms, substitutions, unification, most general unifier (mgu), Horn clause and definite logic program. We do not repeat the definition of these standard notions here. For a background on the necessary materials, the reader is referred to [41] or [54] (Chapter 2). We will also assume basic knowledge of a temporal logic such as the Computation Tree Logic (CTL). For a background on temporal logics, the reader is referred to the survey article [21]. We will use the following symbols (possibly with primes and subscripts): P to denote a definite logic program; C and D for clauses; A, B to denote atoms, p, q, r to denote predicates and σ, θ for substitutions and most general unifier (mgu).

Our proof obligations are always of the form $q \Rightarrow r$ where q, r are predicates in a given definite logic program P. Since the semantics of a definite logic program P is given in terms of its least Herbrand model $M(P)$ [17, 41], therefore this amounts to showing $\forall t \in HU(P)\ q(t) \in M(P) \Rightarrow r(t) \in M(P)$ where $HU(P)$ is the Herbrand Universe of P. In other words, we have to prove $\forall t \in S_{q,P}\ r(t) \in M(P)$ where $M(P)$ is the least Herbrand Model of P and $S_{q,P} = \{t \mid t \in HU(P) \wedge q(t) \in M(P)\}$ (the ground terms for which predicate q is true in P). The terms in $S_{q,P}$ are obtained by starting with the empty set of ground terms and repeatedly applying the rules defining predicate q in program P. This follows immediately from the least Herbrand model semantics of definite logic programs. Thus, if q is defined in program P using the clauses

```
q(0).
q(s(X)) :- q(X).
```

then we start with the empty set ϕ and successively apply the rules to get $\{\mathtt{0}\}$, $\{\mathtt{0}, \mathtt{s(0)}\}$, $\{\mathtt{0}, \mathtt{s(0)}, \mathtt{s(s(0))}\}$ and so on. This will (in the limit) yield $S_{q,P} = \{\mathtt{s}^{\mathtt{n}}\mathtt{(0)} \mid \mathtt{n} \in \mathbb{N}\}$.

Now, suppose predicate r is defined in program P via the clauses

```
r(0).
r(s(X)) :- r(X).
r(...) :- ...
```

Then clearly $\forall t \in S_{q,P}\ r(t) \in M(P)$. This is proved by induction on the terms in $S_{q,P}$. Furthermore, the induction schema is obtained by the recursive structure of the definition of q in P, namely: `0` for the base case and `s(X)` for the inductive step.

The above example shows how predicate implications such as $q \Rightarrow r$ in a program P can be shown by the structure of the definition of q in P. The reasoning corresponds to fixed-point induction; we prove a property about all terms in $S_{q,P}$ by inducting on the fixed-point iterations in computing the least Herbrand model of P. In particular, for a ground atom $q(t) \in M(P)$ we must prove $r(t) \in M(P)$ where we can assume $q(t') \in M(P) \Rightarrow r(t') \in M(P)$ provided $q(t')$ can be proved with fewer fixed point iterations than $q(t)$. All our inductive proofs employ this machinery.

Unfortunately, even if we decide to induct on the structure of clauses defining q in order to prove $q \Rightarrow r$, showing $q(t) \in M(P) \Rightarrow r(t) \in M(P)$ in the inductive step is non-trivial. In the above example this followed from the syntax of q and r. In general to show that $q \Rightarrow r$ in a program P we will need to transform q and r such that $q \Rightarrow r$ can be inferred from the syntax of their transformed definitions. As a simple example, consider the following definitions of q and r given below

```
q(0).
q(s(0)).
q(s(s(X))) :- q(X).
```

```
r(0).
r(s(X)) :- r(X).
```

To prove $q \Rightarrow r$, our proof technique will first transform predicate r to

```
r(0).
r(s(0)).
r(s(s(X))) :- r(X).
```

$q \Rightarrow r$ is then inferred from the syntax of q and the transformed definition of r.

Our transformation based proof technique can be summarized as follows. Given a proof obligation $P_0 \vdash q \Rightarrow r$ where P_0 is a definite logic program and q, r are predicates in P_0 we construct a transformation sequence $P_0, P_1, \ldots, P_k$ s.t.

- $M(P_0) = M(P_k)$ (the transformations are semantics preserving)
- $q \Rightarrow r$ can be inferred from syntax of P_k

We now present the transformation rules, and the mechanism for inferring predicate implications from syntax.

$$
\begin{array}{lcllcl}
 & & \texttt{p(0).} & & & \texttt{p(0).} \\
\texttt{p(X) :- }\boxed{\texttt{q(X)}}\texttt{.} & \xrightarrow{\textit{Unf.}} & \texttt{p(s(X)) :- }\boxed{\texttt{q(X)}}\texttt{.} & \xrightarrow{\textit{Fold}} & & \texttt{p(s(X)) :- p(X).} \\
\texttt{q(0).} & & \texttt{q(0).} & & & \texttt{q(0).} \\
\texttt{q(s(X)) :- q(X).} & & \texttt{q(s(X)) :- q(X).} & & & \texttt{q(s(X)) :- q(X).} \\
\text{Program } P'_0 & & \text{Program } P'_1 & & & \text{Program } P'_2
\end{array}
$$

Figure 1. Illustration of unfold/fold transformations

$$
\begin{array}{lcl}
 & & B\sigma_1 :-(\mathcal{G}, \ \mathcal{F}_1, \ \mathcal{G}')\sigma_1. \\
B :-\mathcal{G}, \boxed{\text{A}}, \mathcal{G}'. & & B\sigma_2 :-(\mathcal{G}, \ \mathcal{F}_2, \ \mathcal{G}')\sigma_2. \\
A_1 :-\mathcal{F}_1. & & \vdots \\
A_2 :-\mathcal{F}_2. & \longrightarrow & B\sigma_n :-(\mathcal{G}, \ \mathcal{F}_n, \ \mathcal{G}')\sigma_n. \\
\vdots & & A_1 :-\mathcal{F}_1. \\
A_n :-\mathcal{F}_n. & & \vdots \\
 & & A_n :-\mathcal{F}_n.
\end{array}
$$

Figure 2. Schema for Unfolding Transformation

2.1. Program transformation rules

We transform a logic program to another logic program by applying transformations that include unfolding and folding. A simple illustration of these transformations appears in Figure 1. Program P'_1 is obtained from P'_0 by unfolding the occurrence of `q(X)` in the definition of `p`. P'_2 is obtained by folding `q(X)` in the second clause of `p` in P'_1 using the definition of `p` in P'_0 (an earlier program). Intuitively, unfolding is a step of clause resolution whereas folding replaces instance of clause bodies (in some earlier program in the transformation sequence) with its head.

We now present the program transformations informally. We call a sequence of program $P_0, P_1, \ldots, P_n$ as a program transformation sequence if for all $0 \leq i < n$, program P_{i+1} is obtained from program P_i by applying a program transformation.

Unfolding of an atom A in the body of a clause in P_i is shown in Figure 2. 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 most general unifier (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.

Folding replaces an occurrence of the body of a clause with its head. The clauses where the replacement takes place are called the *folded* clauses and the clauses used to perform the replacement are called the *folder* clauses. The folding schema is illustrated in Figure 3, where the clauses of B in program P_j are the folded clauses, and the clauses of A

$$P_j: \quad \begin{array}{l} A_1 :- \mathcal{F}'_1. \\ \vdots \\ A_n :- \mathcal{F}'_n. \end{array}$$

$$P_i: \quad \begin{array}{l} B :- \mathcal{G}, \ \mathcal{F}_1, \ \mathcal{G}'. \\ B :- \mathcal{G}, \ \mathcal{F}_2, \ \mathcal{G}'. \\ \vdots \\ B :- \mathcal{G}, \ \mathcal{F}_n, \ \mathcal{G}'. \end{array} \quad \longrightarrow \ B :- \mathcal{G}, \boxed{\text{A}}, \mathcal{G}'.$$

Figure 3. Schema for Folding Transformation

in program P_i 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) $\mathcal{F}_l$ is an instance of $\mathcal{F}'_l$ 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. i.e., from P_j for some $j \leq i$.

In the following, we present a (simplified) version of the unfolding and folding transformation rules. Note that each application of these rules is automated. We say that $P_0, P_1, \ldots, P_n$ is an unfold/fold transformation sequence if the program P_{i+1} is obtained from P_i $(i \geq 0)$ by an application of unfolding or a folding.

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

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

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

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

1. $\forall 1 \leq l \leq m \ \exists \sigma_l \ \forall 1 \leq k \leq n_l \ A_{l,k} = B_{l,k}\sigma_l$
2. $B_1\sigma_1 = B_2\sigma_2 = \cdots = B_m\sigma_m = B$
3. *$D_1, \ldots, D_m$ are the only clauses in P_j whose heads are unifiable with B.*

[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.

4. $\forall 1 \leq l \leq m$, *σ_l substitutes the internal variables*[2] *of D_l to distinct variables which do not appear in $\{A, B, A'_1, \ldots A'_n\}$.*

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

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

Cost of folding The folding rule given above allows the folder clauses to be taken from any P_j where $j \leq i$. This can make the folding operation very costly to implement, since to find applicable folding steps, we must search through the transformation history. To avoid this problem, *all our folding steps pick folder clauses from the initial program P_0*. We will revisit this issue when we discuss transformation strategies in Section 4.

Semantics preservation While unfolding is semantics preserving, indiscriminate folding may introduce circularity and change the program semantics. For this reason, we perform some simple book-keeping with application of each unfold/fold step. The applicability of a folding step is then restricted based on some additional conditions on the book-keeping. A full description of these restrictions is outside the scope of this paper. A complete formal definition of the unfold/fold transformation rules, along with a proof of semantics preservation of any interleaved application of the rules, appears in [55].

2.2. Inferring predicate implications from syntax

In our inductive proof technique, we infer an implication $q \Rightarrow r$ (where q and r are predicates) from the syntax of the transformed definition of q and r. We first describe the intuition behind this syntactic check via an example. Consider the program P in Figure 4. We can infer that $P \models \mathtt{r} \Rightarrow \mathtt{s}$ since for all each clause C of `r`, there is a clause D of `s` such that if `r(t)` is proved using C then `s(t)` is proved using D. Using the implication of `r` and `s` we can infer that $P \models \mathtt{p} \Rightarrow \mathtt{q}$.

We formalize this notion in the following definition. We define a syntactic implication relation $\preceq_P$ on the set of predicates of program P. This relation can be computed from the syntax of the clauses of P. We can infer $p \Rightarrow q$ in program P if $p \preceq_P q$.

DEFINITION 1 (Syntactic Implication). *Given a program P, a syntactic implication relation $\preceq_P$ is a reflexive transitive relation on the set of predicate symbols of P s.t. if $p \preceq_P q$:*

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

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

Figure 4. Syntactic inference of predicate implication $\mathtt{p} \Rightarrow \mathtt{q}$

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 are obtained by replacing every predicate of C_l, D_l by the same predicate symbol. Then there exists a function $f : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$ such that*

   – $\forall 1 \leq l \leq m$ *C'_l is an instance of $D'_{f(l)}$*

   – *Let the clause C_l be $p(\ldots) : -p_1(\ldots), \ldots, p_k(\ldots)$ and the clause $D_{f(l)}$ be $q(\ldots) : -q_1(\ldots), \ldots, q_k(\ldots)$. Then for all $1 \leq j \leq k$ we have $p_j \preceq_P q_j$.*

The existence of the mapping f ensures that for any ground term $\bar{t}$ we have $p(\bar{t}) \in M(P) \Rightarrow q(\bar{t}) \in M(P)$ (a formal proof appears in [54]). Checking $p \preceq_P q$ proceeds by attempting to prove $\neg(p \preceq_P q)$ which is done by showing the absence of a suitable mapping f mentioned above. If such an attempt fails, we the infer that $p \preceq_P q$. Clearly, the check $p \preceq q$ may recursively compute checks for implication among other predicates $p_j \preceq_P q_j$ where p_j (q_j) is used in the definition of p (q). The worst case complexity of checking $p \preceq_P q$ is $O(N^2)$ where N is the number of predicate symbols in P. Details appear in [54].

2.3. Salient features of our inductive proof technique

Our program transformation based proof technique can be viewed as a lightweight inductive theorem proving technique for automating nested induction proofs (where each induction argument proceeds without strengthening of hypothesis). We now study the salient features of our proof technique, as compared to traditional inductive proof techniques (employed in theorem provers). Since our logic program transformations essentially correspond to first order reasoning, we compare with a first order theorem prover (ACL2) [35] to concretely highlight the differences.

Note that in our approach, the induction schema as well as the lemmas to be used in the inductive proof are implicit in the logic program itself. The program transformations make them explicit. Therefore, the proof constructed by our approach has lesser case splits (and hence is more succinct) than the proof constructed by a theorem prover. To explain this point, we consider the following example. The predicate `gen`[3] generates strings of the form `0`* while predicate `bad` captures strings containing at least one occurrence of `1`. The predicate `badgen` captures all strings for which both `bad` and `gen` hold. We want to prove $\forall$`X` $\neg$`badgen(X)` *i.e.* the predicate equivalence `badgen` $\Rightarrow$ `false`.

```
gen([]).
gen([0|X]) :- gen(X).
bad([1|X]).
bad([H|X]) :- bad(X).
badgen(X) :- bad(X), gen(X).
```

Our unfold/fold transformation based proof technique will transform the clauses of `badgen/1`. We first unfold `bad(X)` using the clauses of `bad` to yield

```
badgen([1|X]) :- gen([1|X]).
badgen([H|X]) :- bad(X), gen([H|X]).
```

By unfolding `bad(X)`, the induction schema has been uncovered (which is implicit in the definition of `bad`). The two clauses of `badgen` given above correspond to the *two cases* of this induction schema. Now by unfolding the occurrences of `gen` in these clauses we obtain the following program. These steps correspond to showing the base case and evaluating the finite part of the induction step.

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

Finally, we fold using the original definition of `badgen`. This corresponds to remembering the old definition of `badgen` and using it to uncover the induction hypothesis. Note that the least fixed point semantics of logic programs allows us to conclude that `badgen` $\Rightarrow$ `false`. The induction step in this proof argues that if `badgen(X)` is false then `badgen([0|X])` is also false. This reasoning was enabled by the folding step.

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

In contrast if we input this problem to the ACL2 theorem prover [35] it attempts to prove by an induction schema obtained from `bad`, and generates the following *four cases* for showing $\forall$`X` $\neg$`badgen(X)`.

1. `X` = `[]`

[3] The empty list `[]` denotes the empty string ϵ.

2. $\mathtt{X} = \mathtt{[H|T]} \wedge \mathtt{H} = 1$
3. $\mathtt{X} = \mathtt{[H|T]} \wedge \mathtt{H} \neq \mathtt{1} \wedge \neg\, \mathtt{bad(T)}$
4. $\mathtt{X} = \mathtt{[H|T]} \wedge \mathtt{H} \neq \mathtt{1} \wedge \neg\, \mathtt{gen(T)}$

Note that our proof technique also derived the induction schema from `bad` but considered only *two cases.* The first case in the ACL2 proof is never considered by our proof technique since we discover the induction schema by the process of *unification* inbuilt into unfolding. The absence of types in logic programs enables us to encode `bad` for only non-null lists, and since we obtain the induction schema from this encoding we never consider the first case. The second case above corresponds to the case obtained from the first clause of `bad` in our proof. Finally, the third and fourth cases are combined into a single case in our proof. In fact in our proof, the folding transformation prevents this unnecessary case split and uncovers the induction hypothesis, all in one step.

Role of unfold-fold In the above example, inductive reasoning is performed by syntactic check of the transformed clauses of `badgen/1`. In this proof, the following roles are played by unfolding and folding. Unfolding is used to (i) uncover the induction schema, (ii) dispense the base case and the finite part of the induction step. Given the induction schema (which is discovered by unfolding in the first place), folding attempts to uncover the induction hypothesis.

Summary In summary, our program transformation based proof technique is useful for constructing induction proofs where the induction schema as well as the requisite lemmas are implicitly encoded in the logic program itself. Our syntax based transformations *(i)* make the schema explicit *(ii)* reason about the different cases of the schema by uncovering the requisite lemmas. Furthermore, our transformations do so by (a) not considering spurious cases in the schema, and (b) combining some of the cases of the schema into a single case. Both of these advantages of our proof technique have been illustrated through the example given above.

Remark Note that we have encoded `badgen` as `badgen(X) :- bad(X), gen(X)` instead of `badgen(X) :- gen(X), bad(X)`. This is because ACL2 chooses to induct on a scheme obtained from `bad` in this problem. By encoding `badgen` as `badgen(X) :- bad(X), gen(X)` we can give a point by point comparison with ACL2 since then both proof techniques are inducting on the same schema. In the alternative encoding also our unfold/fold based proof technique generates only two cases in the induction schema (which are obtained from the definition of `gen`).

3. Methodology for Verifying Invariants

In this section, we present our inductive verification methodology based on transformation of definite logic programs. In particular, we describe the use of this methodology in proving invariant properties of parameterized systems. In Section 3.1, we describe how the parameterized system to be verified is encoded as a logic program. Section 3.2 describes the logic program encoding of the invariant to be verified. It also discusses how the task of proving an invariant of a parameterized system can be achieved by proving obligations of the form $p \Rightarrow q$ where p, q are predicates in the logic program encoding. Section 3.3 presents a proof search skeleton for proving such predicate implications. Finally, Section 3.4 employs the verification methodology on a concrete example: proving mutual exclusion in an n process token ring for any n.

3.1. System Specification

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

Example: A Complete Graph Network Consider a network of similar processes where any process may perform an autonomous action or communicate with any other process. This situation may occur in a bus-based protocol, where each unit attached to the bus may communicate with any other unit connected to the bus. It corresponds to a complete graph network since any process can communicate with any other process. Many of our bus based cache coherence protocol examples fall in this category.

We can model the global state of this parameterized network as an unbounded list of the local states of the individual processes. Using Prolog notation, we write a list with head `H` and tail `T` as `[H|T]`. The transition relation `trans` can then be defined over these global states as follows:

```
trans([H|T], [H1|T1]) :-  ltrans(H, in(Act), H1),
                          trans_rest(T, out(Act), T1).
trans([H|T], [H1|T1]) :-  ltrans(H, out(Act), H1),
                          trans_rest(T, in(Act), T1).
trans([H|T], [H1|T]) :-   ltrans(H, self(Act), H1).
trans([H|T], [H|T1]) :-   trans(T, T1).

trans_rest([S|T], A, [S1|T]) :-  ltrans(S, A, S1).
trans_rest([H|T], A, [H|T1]) :-  trans_rest(T, A, T1).
```

Thus, each process can perform an autonomous action (denoted in the above as `self(A)`) or an input/output action (denoted as `in(A)` and `out(A)` respectively) where matching input and output actions synchronize. The predicate `ltrans` encodes the local transition relation of each process. For the global transition relation `trans`, the last rule recursively searches the global state representation until one of the first three rules can be applied. The third rule allows any process to make an autonomous action. The first and second rules correspond to the scenario where *any* two processes communicate with each other (possibly via a bus, which is not modeled). In particular, if the processes are $\mathcal{P}_1, \ldots, \mathcal{P}_n$ for some $n \geq 1$, then the possible transitions are:

1. a process $\mathcal{P}_i$ $(1 \leq i \leq n)$ makes an autonomous action. This is simulated by $i-1$ applications of the fourth rule followed by an application of the third rule.

2. a process $\mathcal{P}_i$ $(1 \leq i < n)$ performs an `in(A)` action and another process $\mathcal{P}_j$ $(i < j \leq n)$ performs an `out(A)` action. This is simulated by $i-1$ applications of the fourth rule followed by an application of the first rule. In the body of the first rule, `ltrans` simulates the `in(A)` transition of $\mathcal{P}_i$, and `trans_rest` simulates the `out(A)` transition of a process in $\mathcal{P}_{i+1}, \ldots, \mathcal{P}_n$.

3. a process $\mathcal{P}_i$ $(1 \leq i < n)$ performs an `out(A)` action and another process $\mathcal{P}_j$ $(i < j \leq n)$ performs an `in(A)` action. This is simulated by $i-1$ applications of the fourth rule followed by an application of the second rule. Again, in the body of the second rule, `ltrans`

simulates the `out(A)` transition of $\mathcal{P}_i$, and `trans_rest` simulates the `in(A)` transition of a process in $\mathcal{P}_{i+1}, \ldots, \mathcal{P}_n$.

Modeling Broadcasts The preceding encoding considered only local actions (in a single process) or binary synchronization (involving two processes). Often in parameterized protocols, a process may broadcast an action A to all other processes (see [24]). This is indeed the case for some of our cache coherence protocol benchmarks *e.g.* the MESI protocol [4, 24]. For any action label Act which is broadcast by some process (denoted as `b_out(Act)`), all processes must always be able to receive broadcast of Act (denoted as `b_in(Act)`). Broadcast of an action Act by a process is modeled by the following predicate `trans_bcast/2`.

```
trans_bcast([H|T], Act, [H1|T1]) :-
        ltrans(H, b_out(Act), H1), trans_all(T, Act, T1).
trans_bcast([H|T], Act, [H1|T1]) :-
        ltrans(H, b_in(Act), H1), trans_bcast(T, Act, T1).
trans_all([], _, []).
trans_all([H|T], Act, [H1|T1]) :-
        ltrans(H, b_in(Act), H1), trans_all(T, Act, T1).
```

If the processes are $\mathcal{P}_1, \ldots, \mathcal{P}_n$ for some $n \geq 1$, then a broadcast of action Act involves a process $\mathcal{P}_i$ ($i \leq n$) performing `b_out(Act)` and all other processes ($\mathcal{P}_1, \ldots, \mathcal{P}_{i-1}, \mathcal{P}_{i+1}, \ldots \mathcal{P}_n$) performing `b_in(Act)`. This is achieved by $i-1$ applications of the second rule of `trans_bcast`/2 (which allows the processes $\mathcal{P}_1, \ldots, \mathcal{P}_{i-1}$ to execute `b_in(Act)`), followed by an application of the first rule of `trans_bcast`/2. The first rule allows process $\mathcal{P}_i$ to execute `b_out(Act)` and invokes `trans_all/2` to ensure that all remaining processes ($\mathcal{P}_{i+1}, \ldots \mathcal{P}_n$) execute `b_in(Act)`.

3.2. Property Specification and Proof obligations

Recall that our inductive proof technique is based on program transformations of definite logic programs.[4] Definite logic program cannot encode temporal properties containing both greatest and least fixed point operators, such as the CTL [21] property $AGEFp$. The logic program encoding of such properties must contain negative atoms in the body of clauses. In the following, we outline the proof obligations for verifying invariant and liveness properties, temporal properties with a single fixed point operator.

[4] In particular, we have extended our transformation rules to logic programs with negation in [57]. However these extended rules have not been implemented in our prover.

A liveness property, denoted in CTL as **EF** `good` is encoded as a program predicate `live` defined by the clauses:

```
live(S) :- good(S).        live(S) :- trans(S, T), live(T).
```

The liveness property is proved by showing the predicate implication `gen` $\Rightarrow$ `thm` where `thm` is defined as `thm(X) :- gen(X),live(X)`. Recall that the predicate `gen/1` captures the set of initial states of the parameterized family.

An invariant property, denoted in CTL as **AG** $\neg$`bad` can be verified in two different ways. We can define predicates `unsafe` and `thm` as:

```
unsafe(S) :- bad(S).
unsafe(S) :- trans(S, T), unsafe(T).
thm(S) :- gen(S), unsafe(S).
```

and prove the implication `thm` $\Rightarrow$ `false`. This constitutes a proof of the invariant property since it shows that none of the initial states are `unsafe` *i.e.* a `bad` state is not reachable from any initial state satisfying `gen`. Note that a proof of the invariant property by showing `thm` $\Rightarrow$ false typically involves two nested inductions. The outer one (uncovered by unfolding `gen`) inducts on the process structure of the parameterized network. The inner one (uncovered by unfolding `unsafe`) is a computational induction which shows that a `bad` state is never reachable.

Alternatively, we can prove the invariant **AG** $\neg$`bad` by proving transition invariance. We prove that (1) a `bad` state is reachable only from a `bad` state, and (2) none of the initial states satisfying `gen` are `bad`. This constitutes a proof of the invariant, and is shown by establishing the predicate implications (1) `bad_dest` $\Rightarrow$ `bad_src`, and (2) `bad_start` $\Rightarrow$ `false` where `bad_dest`, `bad_src` and `bad_start` are defined as:

```
bad_dest(S, T) :- trans(S, T), bad(T).
bad_src(S, T) :- trans(S, T), bad(S).
bad_start(S) :- gen(S), bad(S).
```

This alternative formulation of proving transition invariance simplifies the proof of the invariant (as compared to the previous formulation of proving `thm` $\Rightarrow$ `false`). It breaks the proof into two different obligations `bad_dest` $\Rightarrow$ `bad_src` and `bad_start` $\Rightarrow$ `false`. Both these proof obligations are proved by induction on the structure of the global states of the parameterized network. Since none of the initial states are `bad` and a `bad` state is only reachable from a `bad` state, this means that a `bad` state is never reachable from an initial state. In fact, showing `bad_dest` $\Rightarrow$ `bad_src` and `bad_start` $\Rightarrow$ `false` to prove an invariant corresponds to the INV proof rule of Manna and Pnueli [42].

Proof obligation:
$P_0 \vdash \mathtt{p} \Rightarrow \mathtt{q}$ where P_0 is a program and $\mathtt{p}, \mathtt{q}$ are predicates appearing in P_0.

Proof Search strategy:

1. Repeatedly *unfold* the clauses of $\mathtt{p}$ and $\mathtt{q}$ according to an unfolding strategy which is designed to guarantee termination.
2. a) Apply *folding* steps to the unfolded clauses of $\mathtt{p}$, $\mathtt{q}$.
   b) Perform a clean-up of the clauses in $\mathcal{R}(\mathtt{p})$ and $\mathcal{R}(\mathtt{q})$ to delete failed clauses.
3. (a) *Compare* the transformed definitions of $\mathtt{p}$ and $\mathtt{q}$ to compute a finite set $\{(\mathtt{p_1}, \mathtt{q_1}), \ldots, (\mathtt{p_k}, \mathtt{q_k})\}$ s.t. proving $\bigwedge_{1 \leq 1 \leq k} P_0 \vdash \mathtt{p_1} \Rightarrow \mathtt{q_1}$ completes the proof of $P_0 \vdash \mathtt{p} \Rightarrow \mathtt{q}$ (*i.e.* $\mathtt{p} \Rightarrow \mathtt{q}$ can then be shown via our syntactic check). These k proof obligations are called *conditional implications* since the proof of $\mathtt{p} \Rightarrow \mathtt{q}$ is conditional to proving these implications.
   (b) Prove the conditional implications $P_0 \vdash \mathtt{p_1} \Rightarrow \mathtt{q_1}, \ldots, P_0 \vdash \mathtt{p_k} \Rightarrow \mathtt{q_k}$ via program transformations.

Figure 5. Core skeleton for proof search

*We will follow this formulation of proving invariants (*i.e. *proving invariants by transition invariance) in the rest of this paper.*

3.3. Proof Search Skeletion

We have shown parameterized systems and invariant properties can be encoded as logic program predicates. We have also shown how invariant verification of parameterized systems can be achieved by proving implications among these logic program predicates. We now outline the skeleton of the proof technique which establishes such predicate implications. Given a logic program encoding of the verification problem P_0 and a proof obligation $p \Rightarrow q$, the proof search to show $P_0 \vdash p \Rightarrow q$ proceeds as shown in Figure 5. The proof obligation $P_0 \vdash p \Rightarrow q$ formally means: for all ground terms $\bar{t}$ we have $p(\bar{t}) \in M(P_0) \Rightarrow q(\bar{t}) \in M(P_0)$ where $M(P_0)$ is the set of ground atoms which are logical consequences of the first-order formulae represented by logic program P_0. In logic programming terminology, $M(P_0)$ is also called the least Herbrand model of program P_0.

Since the proof of each predicate implication proceeds by induction (on the structure of their definition), nesting of the proof obligations

$P_0 \vdash \mathtt{p_1} \Rightarrow \mathtt{q_1}, \ldots, P_0 \vdash \mathtt{p_k} \Rightarrow \mathtt{q_k}$ within the proof of $P_0 \vdash \mathtt{p} \Rightarrow \mathtt{q}$ corresponds to nesting of the corresponding induction proofs. The proof search skeleton in Figure 5 uses the notion of *relevant clause set of a predicate* p (denoted $\mathcal{R}(p)$) defined as follows:

DEFINITION 2. **Relevant Clause Set** *Let* P *a program and* p *a predicate symbol in* P*. Let* $reach(p, P)$ *denote the set of predicates which are reachable from the predicate* p *in the predicate dependency graph of* P *(the predicate dependency graph of a program* P *has the predicate symbols of* P *as its vertices, and there is an edge from predicate* p *to predicate* q *if* q *occurs in the body of a clause of* p *in program* P*). Then, the relevant clause set of* p *(denoted* $\mathcal{R}(p, P)$ *) is the set of clauses of the predicates in* $reach(p, P)$*.*

We often denote the relevant clause set of p as $\mathcal{R}(p)$ if the program P is obvious from the content.

The decision to clean-up failed clauses in the relevant clause set of `p`, `q` in step 2(b) of our proof search skeleton is to enable our syntactic implication check. In particular, the clean-up of failed clauses is formally captured by the following transformation.

TRANSFORMATION 3 (**Deletion of failed clauses).** Let q be a n-ary predicate symbol in P_i and let $Cl_i(q)$ be the clauses of q in P_i. Suppose for all $C \in Cl_i(q)$, the body of C contains either the symbol `false` or a recursive occurrence of q.

Then, $P_{i+1} := P_i - Cl_i(q) - Occ_i(q)$ where $Occ_i(q)$ is the set of clauses in P_i in which predicate q appears in the body. □

Suppose after unfolding and folding (steps 1 and 2(a) in Figure 5), the relevant clause set of `p` and `q` are as follows:

```
p(X) :- b(X).
p(X) :- r(X).           q(X) :- b(X).
r(f(X)) :- r(X).
```

Then, we can infer that $\mathtt{r} \Rightarrow \mathtt{false}$. Therefore, after removing the second clause of `p` (which is a failed clause) our syntactic implication check can infer that $\mathtt{p} \Rightarrow \mathtt{q}$.

In our proof search, predicates defined using system/property description are transformed (such as `bad_dest`, `bad_src`) but the system/property description predicates (such as `gen`, `bad`) are not transformed. For example suppose the predicate `gen` denotes a system specification (it generates the global states of the parameterized system) and the predicate `prop` denotes a property specification. If we have a new predicate `new` defined as:

```
new(X)  :- gen(X), prop(X).
```

then we transform `new` by using the definitions of `gen` and `prop`. However the predicates `gen` and `prop` are not transformed. This is because, the purpose of transforming the clauses of a predicate is to expose its underlying recursive structure. The system description is assumed to succintly encode the recursive structure of the parameterized family (the network topology and the pattern of interactions in a global transition). Similarly, the property description is assumed to encode the recursive structure underlying the temporal property being verified.

The above proof search skeleton forms the core of our automated prover which has been implemented on top of the XSB logic programming system [65]. Note that the proof search skeleton is nondeterministic *i.e.* several unfolds or several folds may be applicable at some step. A full discussion on how a transformation step is selected among all applicable transformations is involved, and appears in [54]. However, in the next section (Section 4), we give a flavor of the search heuristics by presenting the strategies for (a) when to stop unfolding (step 1 of the proof search skeleton) (b) choosing folding steps (step 2a of the skeleton), and (c) how to choose conditional implications (step 3a).

3.4. An Example

We now describe our core technique for inductive verification through a simple example. We consider the problem of verifying mutual exclusion in an n-process token ring (refer Figure 6). In working out the proof of mutual exclusion, the non-determinism of our proof search skeleton will become apparent to the reader. The strategies for resolving this non-determinism (*i.e.* choosing an applicable transformation from several candidates) appear in the next section.

The n-process token ring is described by the predicates `gen`, `trans`, `trans1`, `trans2` (refer Figure 6). We represent the global state of a ring as a list of local states, by arbitrarily choosing a 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.

We want to verify that for any n, for any state in a n-process token ring, *always there is at most one token in the ring.* The complement of this property is encoded by the predicate `bad`. We then show transition invariance *i.e.* a `bad` state can be reached only from a `bad` state. This is proved by showing `bad_dest` $\Rightarrow$ `bad_src`. In addition, if we show

```
        Predicates appearing in proof obligations:
C1:     bad_start(X) :- gen(X), bad(X).
C2:     bad_src(X,Y) :- trans(X,Y),bad(X).
C3:     bad_dest(X,Y) :- trans(X,Y),bad(Y).

        System description predicates:
C4:     gen([0,1]).
C5:     gen([0|X]) :- gen(X).
C6:     trans(X, Y) :- trans1(X, Y).
C7:     trans([1|X], [0|Y]) :- trans2(X, Y).
C8:     trans1([0,1|T], [1,0|T]).
C9:     trans1([H|T], [H|T1]) :- trans1(T, T1).
C10:    trans2([0], [1]).
C11:    trans2([H|X], [H|Y]) :- trans2(X, Y).

        Invariant description predicates:
C12:    bad([1|X]) :- one_more(X).
C13:    bad([_|X]) :- bad(X).
C14:    one_more([1|_]).
C15:    one_more([_|X]) :- one_more(X).
```

Figure 6. Mutual exclusion in a n-process token ring

that none of the initial states are `bad`, which is established by showing `bad_start` $\Rightarrow$ `false`, then the proof of mutual exclusion is complete.

We first show the proof of `bad_start` $\Rightarrow$ `false` followed by the proof of `bad_dest` $\Rightarrow$ `bad_src`.

Proof of `bad_start` $\Rightarrow$ `false` The implication `bad_start` $\Rightarrow$ `false` signifies that the initial state of n-process token ring is safe for any n. C_1 is the only clause defining `bad_start` in P_0. We unfold this clause to obtain the clause C_1'

$$\mathtt{bad\_start}([\mathtt{0}|\mathtt{X}]) : -\mathtt{gen}(\mathtt{X}), \mathtt{bad}(\mathtt{X}).$$

A folding step is now applicable. Folding C_1' using C_1 we obtain the clause C_2' given below. No more folding steps are applicable.

$$\mathtt{bad\_start}([\mathtt{0}|\mathtt{X}]) : -\mathtt{bad\_start}(\mathtt{X}).$$

The predicate `bad_start` is now defined by only a self-recursive clause. By removing this failed clause, `bad_start` has no clauses. Our syntactic implication check infers `bad_start` $\Rightarrow$ `false`.

Proof of `bad_dest` $\Rightarrow$ `bad_src` First we unfold clauses C_2, C_3 (the defining clauses of `bad_src`, `bad_dest`) to obtain a program P_{10} where `bad_src`, `bad_dest` are defined as follows.

C_3': `bad_src([0,1,1|X], [1,0,1|X]).`
C_4': `bad_src([0,1,H|T], [1,0,H|T]) :- one_more(T).`
C_5': `bad_src([1|X],[1|Y]) :- trans1(X,Y), one_more(X).`
C_6': `bad_src([H|X],[H|Y]) :- trans1(X,Y), bad(X).`
C_7': `bad_src([1,1|X],[0,1|Y]) :- trans2(X,Y).`
C_8': `bad_src([1,H|X],[0,H|Y]) :- trans2(X,Y), one_more(X).`
C_9': `bad_dest([0,1,1|X], [1,0,1|X]).`
C_{10}': `bad_dest([0,1,H|T], [1,0,H|T]) :- one_more(T).`
C_{11}': `bad_dest([1|X],[1|Y]) :- trans1(X,Y), one_more(Y).`
C_{12}': `bad_dest([H|X],[H|Y]) :- trans1(X,Y), bad(Y).`
C_{13}': `bad_dest([1,1|X],[0,1|Y]) :- trans2(X,Y), one_more(Y).`
C_{14}': `bad_dest([1,H|X],[0,H|Y]) :- trans2(X,Y), bad(Y).`

No more unfolding is applicable without risking non-termination of unfolding (this is clarified further when we discuss the transformation strategies in the next section). Also, no folding steps are applicable. At this stage we find the following nested proof obligations (*i.e.* if they are proved then the proof of `bad_dest` $\Rightarrow$ `bad_src` is complete).

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

Thus, we introduce the following definitions

D_1: `s1(X,Y) :- trans1(X,Y), one_more(X).`
D_2: `s2(X,Y) :- trans1(X,Y), one_more(Y).`
D_3: `f1'(X,Y) :- trans1(X,Y), bad(X).`
D_4: `f2'(X,Y) :- trans1(X,Y), bad(Y).`
D_5: `g2''(X,Y) :- trans2(X,Y), one_more(Y).`
D_6: `g1''(X,Y) :- trans2(X,Y), one_more(X).`
D_7: `f2''(X,Y) :- trans2(X,Y), bad(Y).`

and invoke the following subproofs.

`s2` $\Rightarrow$ `s1`, `f2'` $\Rightarrow$ `f1'`, `g2''` $\Rightarrow$ `trans2`, `f2''` $\Rightarrow$ `g1''`

Once these subproofs are established we can infer `bad_src` $\Rightarrow$ `bad_dest`. We now show the construction of the first subproof. The others follow similarly.

Proof of `s2` $\Rightarrow$ `s1` We perform unfolding to obtain the following clauses for predicates `s1` and `s2`. No further unfolding is possible without risking non-termination.

D'_1: `s1([0,1|X], [1,0|X]).`
D'_2: `s1([1|X], [1|Y]) :- trans1(X, Y).`
D'_3: `s1([H|X], [H|Y]) :- trans1(X, Y), one_more(X).`
D'_5: `s2([0,1|X], [1,0|X]).`
D'_6: `s2([1|X], [1|Y]) :- trans1(X, Y).`
D'_7: `s2([H|X], [H|Y]) :- trans1(X, Y), one_more(Y).`

Folding steps are now applicable. Folding clause D'_3 using clause D_1 as folder we get the following definition of `s1`

D'_1: `s1([0,1|X], [1,0|X]).`
D'_2: `s1([1|X], [1|Y]) :- trans1(X, Y).`
D'_4: `s1([H|X], [H|Y]) :- s1(X, Y).`

Only one more folding step is applicable. We fold D'_7 using clause D_2 as folder. We get:

D'_5: `s2([0,1|X], [1,0|X]).`
D'_6: `s2([1|X], [1|Y]) :- trans1(X, Y).`
D'_8: `s2([H|X], [H|Y]) :- s2(X, Y).`

We now can now infer `s2` $\Rightarrow$ `s1` from syntax (refer Definition 1). In other words, we can map any clause C of `s2` to some clause D of `s1` s.t. for any ground term $\overline{\texttt{t}}$, if `s2`$(\overline{\texttt{t}})$ is proved using C then `s1`$(\overline{\texttt{t}})$ is proved using D. This completes the proof of `s2` $\Rightarrow$ `s1`.

4. Strategies for Verifying Parameterized Systems

The inductive reasoning accomplished by our transformation based proof technique has been exploited to build an automated prover for parameterized protocols. Once the parameterized system and the temporal property are specified, the prover attempts to produce a completely automated proof of the corresponding predicate implications. We now present concrete strategies (which are employed by the prover) to guide the application of the transformation rules in a proof.

Note that to prove the equivalence $P_0 \vdash p \equiv p'$, we need to transform only the relevant clause set of p, p' *i.e.* the clauses in $\mathcal{R}(p)$ and $\mathcal{R}(p')$ (refer Definition 2). However, we choose to transform only the clauses of predicates p and p' at every transformation step. These clauses are denoted as $\mathcal{C}(p)$ and $\mathcal{C}(p')$. Clearly, $\mathcal{C}(p) \subseteq \mathcal{R}(p)$ for any predicate p. However, note that the transformation of predicates defined using $\mathcal{R}(p) - \mathcal{C}(p)$ can be simulated by spawning conditional implications on these predicates. This design decision makes our proof search more *goal directed.* It ensures that we transform clauses in $\mathcal{R}(p) - \mathcal{C}(p)$ and $\mathcal{R}(p') - \mathcal{C}(p')$ only when it is required for proving $p \Rightarrow p'$.

4.1. Controlling Algorithmic Steps

In a predicate implication proof of $P_0 \vdash p \Rightarrow p'$, where P_0 is the initial program, most of the transformation steps are unfolding steps. Moreover, even when we restrict ourselves to unfolding only the clauses of p and p', several unfolding steps are enabled in every step of the proof. The criteria considered for choosing unfolding steps are:

- *Termination* : Unfolding sequences must terminate.
- *Selection Order* : In a clause, body atoms are expanded left to right.
- *Convergence* : Unfolding steps should not disable deductive steps which lead to a proof.

Our unfolding strategy guarantees termination and left-to-right selection order. It also includes heuristics for fast convergence. We now discuss the motivations and the heuristics for following each of the above three criteria. For this purpose, we introduce the following notion of *unfolding tree.* This notion formalizes the repeated unfolding of a clause.

DEFINITION 3 (Unfolding Tree). *Let P be a program and $C \in P$. An unfolding tree of clause C is a tree T s.t.*

1. *each node of T is labeled by a clause; the root is labeled by C.*
2. *let $\mathcal{N}$ be an internal node in T which is labeled by a clause D. Then the children of node $\mathcal{N}$ in T are labeled by the clauses obtained by unfolding A in clause D, where A is any body atom of D. The atom A is called the selected atom of $\mathcal{N}$.*

Termination: To create finite unfolding sequences we impose a finiteness condition *FIN* on transformation sequences. We do not give an exact implementation of *FIN* but only a sufficient condition s.t. the resultant unfolding sequences terminate.

DEFINITION 4 (Finiteness condition). *Given an a-priori fixed constant $k \in \mathbb{N}$, an unfolding transformation sequence $\Gamma = P_0, \ldots, P_n, \ldots$ satisfies the finiteness condition $FIN(\Gamma, k)$ if for the clause C and atom A selected for unfolding at every P_i: (1) A is distinct (modulo variable renaming) from any atom B which was selected in unfolding some clause $D \in P_j (j < i)$ where C is obtained by repeated unfolding of D (2) the term depth of each argument of A is $\leq k$.*

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

The finiteness condition *FIN* of unfolding transformation sequences (presented in Definition 4) can be used to ensure finiteness of unfolding trees.

LEMMA 1. *For any clause C in program P, any unfolding tree of C is finite if every unfolding step satisfies FIN (refer Definition 4).*

The finiteness condition *FIN* ensures termination of any unfolding transformation sequence. As discussed before, condition (1) of *FIN* ensures that in any path of an unfolding tree no two selected atoms are same (modulo variable renaming). Condition (2) of *FIN* ensures that there are finitely many atoms which may appear as selected atom of a node of the unfolding tree. Thus, *FIN* ensures that every path in an unfolding tree is finite. Furthermore, every node of the unfolding tree is finitely branching, thereby establishing Lemma 1.

Now, let us examine the amount of book-keeping needed to ensure termination of a sequence of unfolds. Let $P_0, \ldots, P_n$ be a sequence of unfold steps. Then, for every clause $C \in P_n$ (the current program in the transformation sequence) we need to store the selected atoms of the clauses in $P_0, \ldots, P_{n-1}$ from which C was obtained via repeated unfolding. We store this information efficiently as follows. For an unfold step $P_{n-1} \rightarrow P_n$ where $C \in P_{n-1}$ is unfolded to $C_1, \ldots, C_m$ we simply store the clause identification number of C as the parent of $C_1, \ldots, C_m$ and remember the atom unfolded in this step. Note that this does *not* require us to store all the programs in the program transformation sequence $P_0, \ldots, P_{n-1}$.

Selection Order: The logic program encoding of parameterized system verification problem has a "*producer-consumer*" nature. For proving invariants we need to show `bad_dest` $\Rightarrow$ `bad_src` where

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

During the transformation of `bad_dest`, we intend to produce instantiation for variable `Y` by unfolding the system description (in this case `trans(X,Y)`) and then this instantiation is to be consumed by unfolding the property description (in this case `bad(Y)`). In general, in the proof of a parameterized protocol if p is a predicate being transformed, then any clause of p contains system description predicates *followed* by property description predicates in the body. There is an intended left to right ordering for unfolding the body atoms of the predicates in these predicate implication proofs. Therefore, we always unfold the body atoms of a clause from *left to right*.

Convergence Ensuring termination of unfold sequences is *only one* of the considerations in guiding unfolding. In the proof of $P_0 \vdash p \Rightarrow p'$, our proof search skeleton (refer Figure 5) applies deductive steps like folding subsequent to unfolding. Suppose by guiding unfolding solely based on termination, we produce a transformation sequence $P_0, \ldots, P_i, \ldots, P_n$. However, this might disable certain deductive steps *i.e.* a folding step applicable in P_i which leads to a proof of $P_0 \vdash p \Rightarrow p'$ might be disabled in P_n. In general, to prevent disabling of deductive steps we need to check for applying deductive steps ahead of algorithmic steps. However, this would add theorem proving overheads to model checking. Our goal is to perform zero-overhead theorem proving, where deductive steps are never applied if model checking can complete the verification task. Therefore, we apply the deductive steps on demand and identify "redundant" unfolding steps which are likely to diverge.

As mentioned earlier, the logic program encoding of a parameterized system verification problem possesses a "producer-consumer" nature. Therefore occurrences of predicates appearing in the property description are intended to consume variable instantiations rather than produce it. For this purpose, our prover prohibits the unfolding of any open atom[5] of a predicate appearing in the property description. We now explain in details the justification of this design decision.

Note that atoms of predicates appearing in the system description when unfolded instantiate variables to terms representing global states of the parameterized family. For example to prove `bad_dest` $\Rightarrow$

[5] Atoms of the form $p(X_1, \ldots, X_n)$ where p is a predicate and $X_1, \ldots, X_n$ are variables.

`bad_src`, we first unfold `trans(X,Y)` appearing in the definition clause of `bad_dest` given below.

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

Suppose by repeatedly unfolding `trans` the variable `Y` gets instantiated to a term $\overline{\mathtt{t}}$. Now, by unfolding `bad`($\overline{\mathtt{t}}$) we intend to *test* whether the invariant holds in global states represented by $\overline{\mathtt{t}}$. Therefore, the unfolding of `bad`($\overline{\mathtt{t}}$) should consume the instantiation $\overline{\mathtt{t}}$ rather than instantiating $\overline{\mathtt{t}}$ further. However, since each unfolding step involves unification, it is possible to instantiate $\overline{\mathtt{t}}$ by unfolding `bad`($\overline{\mathtt{t}}$). Therefore, for the logic program encoding of the verification problem for parameterized systems, we mark any predicate describing the temporal property as a *consumer predicate*. Although, we want to prohibit any instantiation of atoms of consumer predicates via unfolding, implementing this is too expensive (since we need to check after each unification step whether any such instantiation occurred). Instead, as a heuristic, we simply prohibit unfolding open atoms of consumer predicates. Since all arguments of such an atom are variables, any unfolding is likely to instantiate them. As system and property description predicates share variables, such an instantiation will enable further unfolding of the system description predicates (*e.g.* `trans`) without violating the finiteness condition *FIN*. This can disable deductive steps converging to a proof *e.g.* folding of conjunction of `trans` and `bad` to `bad_dest`.

An Unfolding Strategy The three criteria of termination, convergence and selection order can now be distilled into the following unfolding strategy. In unfolding a clause C, we select a body atom A of C s.t.

1. A does not violate the finiteness condition *FIN*.
2. A is not an open atom of a predicate appearing in the description of the temporal property.
3. A is the leftmost body atom of C satisfying (1) and (2).

4.2. Controlling Folding Steps

We now discuss control strategies for guiding the folding steps. In the proof of $P_0 \vdash p \Rightarrow p'$ suppose we have constructed a transformation sequence $P_0, P_1, \ldots, P_i$ by unfolding the clauses of p and p'. In the following we present strategies for choosing a step from the different applicable folding steps in P_i (and the subsequent programs formed by repeatedly transforming P_i).

Choosing from applicable fold transformations involves:

- selecting folder clauses $\{D_1, \ldots, D_m\} \subseteq P_j$ $(0 \leq j \leq i)$,
- selecting folded clauses $\{C_1, \ldots, C_m\} \subseteq P_i$, and then
- checking whether $\{C_1, \ldots, C_m\}$ is foldable using $\{D_1, \ldots, D_m\}$ as folder.

By suitably restricting the choice of folder and folded clauses we can avoid expensive foldability checks which fail. Moreover, the number of unfold steps in the proofs of predicate implication in our parameterized protocols are typically > 100. Thus, we do not store the intermediate programs $P_1, \ldots, P_{i-1}$. Only the initial program P_0 and the current program P_i are remembered by the prover (note that all new predicate definitions that are introduced during a proof are assumed to be in P_0). *We always pick the folder clauses from the initial program P_0.* This choice is further restricted by preferentially picking predicates from P_0 which appear in some implication proof obligation encountered so far. This is because our proof technique is geared towards exposing the underlying recursive structure of the predicates appearing in proof obligations. Thus the clauses of these predicates are likely candidates for folder. Once a predicate q is selected, the clause set of q in P_0 then constitutes our choice of folder.

Once the folder clauses are selected, due the blow-up in the number of clauses generated by unfolding, the search for folded clauses is extremely expensive. Therefore, we consider only those clause sets which match with our choice of folder at the propositional level. Note that any clause-set which does not match with the folder at the propositional level does not constitute a valid choice of folded clauses. This can be formally stated as follows.

DEFINITION 5 (Propositional Approximation). *Let P be a definite logic program and $C \in P$ s.t. $C = (p(\ldots) :- p_1(\ldots), \ldots, p_k(\ldots))$. Then the propositional approximation of C denoted $prop(C)$ is the clause $p :- p_1, \ldots, p_k$.*

Now, suppose we are attempting to prove $P_0 \vdash p \Rightarrow p'$ and we have chosen folder clauses $\{D_1, \ldots, D_m\} \subseteq P_0$. Then, we choose a set of clauses $\{C_1, \ldots, C_m\} \subseteq P_i$ as folded clauses s.t. (i) $\{C_1, \ldots, C_m\}$ are drawn from the clause set of p or p', and (ii) $\{prop(C_1), \ldots, prop(C_m)\}$ is foldable using $\{prop(D_1), \ldots, prop(D_m)\}$.

C_1	`p([a|X]) :- a1(X), a2(X).`	C'_1	`q([a|X]) :- a1'(X), a2'(X).`
C_2	`p([a|X]) :- a3(X).`	C'_2	`q([b|X]) :- b1'(X), b2'(X).`
C_3	`p([b|X]) :- b1(X), b2(X).`	C'_3	`q([b|X]) :- b3'(X), b4'(X).`
C_4	`p([c|X]) :- c1(X), c2(X).`	C'_4	`q([c|X]) :- c1(X), c2(X).`
C_5	`p([d|X]) :- p(X).`	C'_5	`q([d|X]) :- q(X).`
C_6	`p([e|X]) :- e1(X), e2(X).`	C'_6	`q([f|X]) :- f1'(X), f2'(X).`

```
new1(X) :- a1(X), a2(X).   new1'(X) :- a1'(X), a2'(X).
new1(X) :- a3(X).          new2'(X) :- b1'(X), b2'(X).
new2(X) :- b1(X), b2(X).   new2'(X) :- b3'(X), b4'(X).
new3(X) :- e1(X), e2(X).
```

Figure 7. Choosing conditions for Conditional implication

4.3. Choosing Conditional Implications

Choosing conditional implications is the final step in proving a predicate implication proof obligation $p \Rightarrow q$ (step 3 of the skeleton in Figure 5). It involves invoking new predicate implication proof obligations, which if proved, establish the implication of p and q.

Let us suppose that the clauses of predicates `p`, `q` after unfolding and folding are as shown in Figure 7. We then define new predicates `new1`, `new2`, `new3`, `new1'`, `new2'` and spawn off the predicate implication obligations `new1` $\Rightarrow$ `new1'`, `new2` $\Rightarrow$ `new2'` and `new3` $\Rightarrow$ `false`. Note that this is sufficient to prove `p` $\Rightarrow$ `q`. Thus, we can infer `p` $\Rightarrow$ `q` from syntax provided `new1` $\Rightarrow$ `new1'`, `new2` $\Rightarrow$ `new2'`, and `new3` $\Rightarrow$ `false`.

The above example illustrates the basic intuition behind choosing the conditional implications in a proof of $P_0 \vdash$ `p` $\Rightarrow$ `q`. A formal treatment of this material appears in [54]. Essentially, we attempt to develop a mapping between the clauses of `p` and the clauses of `q`. This mapping will be used to show `p` $\Rightarrow$ `q` provided the conditional implications hold (refer Definition 1 for Syntactic Implication check). Thus, if clause C of predicate `p` is mapped to clause D of predicate `q` then the heads of C and D are identical modulo variable renaming ("variants" in logic programming terminology). The bodies of C and D are used to define new conditional implications. However, note that we cannot generate a conditional implication whose proof has been initiated but not completed. Our heuristics for generating conditional implications does not generate such conditional implications, in order to avoid non-termination of the proof search.

For example, suppose we are proving `p` $\Rightarrow$ `q` the clauses of `p` and `q` after unfolding and folding are as follows.

```
p(f(X)) :- r(X).                q(f(X)) :- r(X).
p(g(X)) :- p(X).                q(g(X)) :- q(X).
p(h(X)) :- a1(X), a2(X).        q(h(X)) :- b1(X), b2(X).
```

Then, we only generate the conditional implication

$$\forall \mathtt{X}\ \mathtt{a1(X)} \wedge \mathtt{a2(X)} \Rightarrow \mathtt{b1(X)} \wedge \mathtt{b2(X)}$$

(from the third clause of p, q). If this proof obligation is proved then our syntactic implication check is powerful enough to infer $\forall \mathtt{X}\ \mathtt{p(X)} \Rightarrow \mathtt{q(X)}$. The second clauses of p, q given above do not contribute any conditional implications.

Role of parameterized network topology For selecting new proof obligations in step 3 of the proof search skeleton of Figure 5, we can exploit the network topology of the parameterized network being verified. The communication pattern between the different constituent processes of a parameterized network is called its network topology. Standard parameterized network topologies include chains, rings, trees etc.

To illustrate the role of network topology in our proof search let us suppose that we are proving bad_dest$\Rightarrow$ bad_src in order to establish an invariant property (refer Section 3.2). In the proof of bad_dest$\Rightarrow$ bad_src, we first unfold and fold the clauses of bad_dest and bad_src. The prover then compares these transformed clauses and detects new predicate implications to be proved. In this final step, the prover exploits the knowledge of the network topology to choose the new predicate implications.

For example, suppose the parameterized family being verified is a linear network and let the clauses of bad_dest and bad_src after unfolding and folding be:

```
bad_dest([a|X], [b|Y]) :- p(X,Y), q(X,Y).
bad_src([a|X], [b|Y]) :- p'(X,Y), q'(X,Y).
```

then our prover will define new predicates

```
new1(X,Y) :- p(X,Y), q(X,Y).
new2(X,Y) :- p'(X,Y), q'(X,Y).
```

and prove new1 $\Rightarrow$ new2. In other words, it proves $\mathtt{p} \wedge \mathtt{q} \Rightarrow \mathtt{p}' \wedge \mathtt{q}'$.

The situation is different if the parameterized network being verified is a binary tree network (where each node of the tree communicates only with its neighbors). If the clauses of bad_dest and bad_src after unfolding and folding are:

$$\mathtt{bad\_dest}(\mathtt{f}(\mathtt{root1}, \mathtt{L1}, \mathtt{R1}), \mathtt{f}(\mathtt{root2}, \mathtt{L2}, \mathtt{R2})) : -\mathtt{p}(\mathtt{L1}, \mathtt{L2}), \mathtt{q}(\mathtt{R1}, \mathtt{R2}).$$
$$\mathtt{bad\_src}(\mathtt{f}(\mathtt{root1}, \mathtt{L1}, \mathtt{R1}), \mathtt{f}(\mathtt{root2}, \mathtt{L2}, \mathtt{R2})) : -\mathtt{p}'(\mathtt{L1}, \mathtt{L2}), \mathtt{q}'(\mathtt{R1}, \mathtt{R2}).$$

then again $\mathtt{p} \wedge \mathtt{q} \Rightarrow \mathtt{p}' \wedge \mathtt{q}'$ needs to be proved to establish $\mathtt{bad\_dest} \Rightarrow \mathtt{bad\_src}$. Instead, the prover recognizes that $\mathtt{p}$, $\mathtt{p}'$ are proof obligations over the left subtree. Similarly $\mathtt{q}$, $\mathtt{q}'$ are proof obligations over the right subtrees. Furthermore, the transitions of the left and right subtrees are independent of each other. Thus it partitions the proof obligation $\mathtt{p} \wedge \mathtt{q} \Rightarrow \mathtt{p}' \wedge \mathtt{q}'$ into two separate obligations defined over the left and right subtrees: $\mathtt{p} \Rightarrow \mathtt{p}'$ and $\mathtt{q} \Rightarrow \mathtt{q}'$. In other words, knowledge of transition system is exploited by the prover to choose conditional implications. The aim of this heuristic is to facilitate faster convergence of the proof attempt.

5. Implementation and Experimental Results

In this section, we present the experimental results obtained using our predicate implication prover. The prover is built on top of the XSB tabled logic programming system [65] which supports top-down memoized evaluation of logic programs. In particular, the memoized evaluation in XSB is helpful for obtaining a declarative encoding of our syntactic implication check (refer Definition 1) [54].

We report results on parameterized cache coherence protocols, including (a) single bus broadcast protocols *e.g.* Mesi, (b) single bus protocols with global conditions *e.g.* Illinois and (b) multiple bus hierarchical protocols. We also report experimental numbers for the Java Meta-locking algorithm [1], a distributed algorithm to ensure secure access of shared objects by various Java threads. The benchmarks cover various network topologies including star, tree and complete graph networks.

Results In Table I, *Meta-lock* denotes the Java meta-locking algorithm from Sun Microsystems. We discuss this benchmark in more details in the next section. *Mesi* and *Berkeley RISC* are single bus broadcast protocols [4, 24, 25]. *Illinois* is a single bus cache coherence protocol with global conditions which cannot be modeled as a broadcast protocol [18, 50]. *Tree-cache* is a binary tree network which simulates the interactions between the cache agents in a hierarchical cache coherence protocol [54]. Finally *German's client server* is a client-server protocol proposed by Steve German. It involves unbounded number of client accessing a central server (a star network topology) and we need to prove coherence of cached copies across the clients. It was proposed

Table I. Summary of protocol verification results

Protocol	Invariant	Time(sec) in [18]	Our time(secs)	# Unf	#Ded
Meta-Lock	#owner + #handout < 2	-	129.8	1981	311
Mesi	#m + #e < 2	1	3.2	325	69
	#m + #e = 0 ∨ #s = 0	0.5	2.9	308	63
Illinois	#dirty < 2	5.3	35.7	2501	137
Berkeley	#dirty < 2	0.6	6.8	503	146
German's client-server	# ex < 2	-	8.8	404	204
	#ex + #sh < 2	-	Fails	-	-
Tree-cache	#bus_with_data < 2	-	9.9	178	18

as a benchmark for parameterized system verification by Ruah, Pnueli and Zuck in [49].

Table I presents experimental results obtained using our prover: a summary of the invariants proved along with the time taken, the number of unfolding steps and the number of deductive steps (*i.e.* folding, and comparison of predicate definitions) performed in constructing the proof. The total time involves time taken by (a) unfolding steps (b) deductive steps, and (c) the time to invoke nested proof obligations. All experiments reported here were conducted on a Sun Ultra-Enterprise workstation with two 336 MHz CPUs and 2 GB of RAM. In the table, we have used the following notational shorthand: $\#s$ denotes the number of processes in local state s. In column 3 of the table, we have shown the timings for the same proofs using the constraint logic program evaluation based checker of [18]. To the best of our knowledge, [18] is the only other work which employs logic programming technology for parameterized system verification, and reports detailed experimental results. Note that the timings of [18] on a Pentium 133 with Linux 2.0.32.

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

Note that for one of the benchmarks (German's client server protocol), our prover fails to construct a proof of the invariant that at most one one client can be in shared or exclusive state. In this case our prover terminates and reports its failure to find a proof (even though one exists). This needs to remedied by manually strengthening the invariant to be proved. However, the desired strengthening turns out to be non-obvious in this case.

Comparison with [18] The running times of our prover are slower than the times for verifying single bus cache coherence protocols reported in [18]. In fact, there is up to an order of magnitude difference between the time taken by our prover and the time taken by the prover of [18]. The reason for this can be explained as follows. The constraint logic program evaluation based model checker of [18] proceeds essentially by unfolding which is performed by the underlying abstract machine. On the other hand, the prototype implementation of our transformation based prover implements both the unfolding and folding steps via meta-programming. Even though our folding steps should ideally be implemented by meta-programming, the proof search via unfolding steps can be conducted at the level of the underlying abstract machine (which should substantially reduce our running times). It would be more interesting to compare the running times of such a prover with the Constraint logic program execution in [18]. Such a comparison would determine whether the time overhead due to folding steps exceeds the time overhead due to constraint solving in Constraint logic program evaluation. Also, note that the count based abstraction technique of [18] is restricted to only certain parameterized network topologies, *e.g.* it cannot be used to verify parameterized tree networks such as *Tree-cache* (which can be verified by our inductive proof technique).

6. A case study

In recent years, Java has gained popularity as a concurrent object oriented language, and hence substantial research efforts have been directed to efficiently implementing the different language features. One of the features which is commonly used in the execution of most Java programs are the synchronization operations. In Java language, any object can be synchronized upon by different threads via synchronized methods and synchronized statements. Mutual exclusion in the access of an object is ensured since a synchronized method first acquires a lock on the object, executes the method and then releases the lock. To ensure fairness and efficiency in accessing any object, each object maintains some *synchronization data.* Typically this synchronization data is a FIFO queue of the threads requesting the object. Note that to ensure mutually exclusive access of an object, it is necessary to observe a protocol while different threads access this synchronization data. The Java meta-locking algorithm [1] solves this problem. It is a distributed algorithm which is observed by each thread and any object for accessing the synchronization data of that object. It is a time and space efficient scheme to ensure mutually exclusive access of the synchronization data, thereby ensuring mutually exclusive access of any object. Below, we outline the modeling of the algorithm and its verification in our prover.

6.1. Modeling the Protocol

First we informally describe the meta-locking algorithm and then present the formal model used by us for verification. As mentioned above, the meta-locking algorithm is a protocol observed by threads in accessing an object's synchronization data.

Informal description We consider the protocol for a single object and an arbitrary number of threads. The pattern of a synchronization operation executed by a thread is as follows:

1. *Get* the object's meta-lock if no thread is accessing the synchronization data; otherwise wait for a *hand-off*.
2. *Manipulate* the synchronization data.
3. *Release* the meta-lock if no other thread is waiting; otherwise *hand-off* the meta-lock to a waiting thread.

Thus, both the acquisition as well as the release of the meta-lock by a thread may involve contention from other threads. A thread that faces no contention from other threads while acquiring/releasing the meta-lock is said to execute a *fast path*. Otherwise, it executes a *slow path* involving a hand-off from another thread.

For the purposes of describing the meta-locking scheme, an object can be assumed to be either busy (when a thread is modifying its synchronization data) or not busy (otherwise). The synchronization data maintained by the object is a FIFO queue of lock records. Each lock record denotes a thread waiting to access the object. A thread is allowed to access the object once it reached the head of the queue. Since we are only interested in modeling and verifying the meta-locking scheme for accessing this FIFO queue, we do not model the queue itself.

A thread attempts to acquire the meta-lock by performing an atomic swap operation of its thread id with the object's internal data-structures, called the *multi-use word* by the designers of the protocol. If the contents of the multi-use word indicate that the object is not busy then the thread has acquired the meta-lock via a fast path. It now proceeds to manipulate the synchronization data. Otherwise, the thread must wait for a hand-off by the thread currently owning the meta-lock. In this case, threads contending for the meta-lock acquire it in the order in which they executed their swap instructions. Note that at any point of time the object's multi-use word records which thread last tried to acquire the meta-lock. When several threads are contending to acquire the meta-lock, each of them knows its predecessor via the swap operation *i.e.* each thread knows the thread which tried to acquire the meta-lock just before it. The first one to execute the swap operation has no predecessor and acquires the meta-lock.

To release the meta-lock, thread i executes an atomic compare-and-swap operation to compare the current value recorded in the multi-use word with its own identifier. If the two values are the same, then the thread infers that no other thread is waiting to access the meta-lock. It then releases the meta-lock via the fast path, and the object is restored to non-busy state. Otherwise, if other threads are trying to acquire the meta-lock, thread i obtains the id of the thread which last tried to acquire the meta-lock. This however, need not be the thread which tried to acquire the meta-lock immediately after thread i, *i.e.* the successor of thread i. In other words, thread i is ready to hand-off the meta-lock but does not know which thread to wake up. For this reason, the algorithm requires the threads acquiring and releasing the meta-lock to enter a race. The purpose of this race is to eliminate busy-waiting from the meta-locking algorithm. The race proceeds as follows. Every thread attempting to acquire the meta-lock executes its portion of the hand-off thereby informing its predecessor of its identity. Simultaneously, the thread releasing the meta-lock executes its portion of the hand-off thereby informing its desire to relinquish the meta-lock. Note that this is effectively a race between the releasing thread i and the successor of i (say thread j) where both i and j compete to execute its portion of the hand-off first. Through the race, thread i releases the meta-lock. Also thread j is identified as the successor of thread i, and thread j acquires the meta-lock.

Formal model The formal model of the protocol consists of the parallel composition of an object process, a hand-off process and an arbitrary number of thread processes. Each process either performs an autonomous action or communicates with another process via synchronization. Two processes synchronize as follows : one of them makes an `in(a)` action and the other makes an `out(a)` action where `a` is an action label. Any process makes an autonomous action by performing a `self(a)` action where `a` is an action label.

We model the object process without the synchronization data (the FIFO queue of lock records) since we are only interested in verifying mutually exclusive access of this data. Thus we do not model the synchronization data and its manipulation by the threads. Apart from the synchronization data, note that the meta-locking algorithm *implicitly* maintains another queue : the queue of threads currently contending for the meta-lock. Modeling this queue of contending threads is essential for a faithful modeling of the acquisition and the release of the meta-lock. However, for verifying mutual exclusion we do not need to model the actual contents of the queue *i.e.* the identifiers of the threads currently contending for the meta-lock. Instead we only model the length of the queue, a natural number. This abstraction of modeling the queue of contending threads via the queue length does not allow us to prove properties like : if thread i requested the meta-lock, it will eventually get the meta-lock. However, we can verify global consistency properties which can be inferred from the global state of the protocol without referring to individual threads. In this example we seek to verify one such property, namely mutual exclusion: no two threads ever own the meta-lock together.

The object can therefore be modeled as a process with a boolean control variable and a data variable which is a natural number. The control variable records whether the object is busy, *i.e.* whether any thread currently possesses the meta-lock. The data variable records the number of processes currently waiting for the meta-lock. The states of the object are represented by the infinite set of terms $\{$ `not_busy` $\} \cup \{$ `busy(N)` $\mid \mathtt{N} \in \mathbb{N}$ $\}$. The state of the object is `not_busy` if no thread owns the meta-lock, and none is waiting for the meta-lock. Otherwise, if a thread owns the meta-lock and n others are waiting for it, the state of the object is `busy(s^n(0))`. The transition relation of the object is described by the following set of logic program clauses. `obj_trans(S, A, T)` is true if the object makes a transition from state `S` to state `T` on action `A`. Note that the object itself is an *infinite state system* since it has a data variable which is a natural number. We assume that natural numbers are represented by the terms $\{0, s(0), s(s(0)), \ldots\}$ as captured by the predicate `nat` shown below.

```
obj_trans(not_busy, in(get_fast), busy(0)).
obj_trans(busy(0), in(put_fast), not_busy).
obj_trans(busy(X), in(get_slow), busy(s(X))) :- nat(X).
obj_trans(busy(s(X)), in(put_slow), busy(X)) :- nat(X).

nat(0).
nat(s(X)) :- nat(X).
```

On the other hand, the thread and the hand-off process are finite state systems. Figure 8 shows the labeled transition systems of a thread and the hand-off process. The logic program encoding of these relations is simply a set of facts[6].

All the transitions of a thread involve synchronizing with either the object process or the hand-off process. When the object is in `not_busy` state, it synchronizes with a thread in `idle` state using a `get_fast` signal and moves to state `busy(0)`. The thread moves to `owner` state and this corresponds to the acquisition of the meta-lock by the thread via fast path. Note that no other thread is waiting for the meta-lock at this time. Similarly, a thread releases a meta-lock via the fast path by synchronizing with the object using a `put_fast` signal. The object state is then restored to `not_busy`. While the object is meta-locked in a state $\mathtt{busy}(\mathtt{s}^{\mathtt{n}}(0))$ if another thread tries to acquire the meta-lock via the slow path it synchronizes with the object using a `get_slow` signal. The number of threads waiting for the meta-lock is then incremented and the object moves to state $\mathtt{busy}(\mathtt{s}^{\mathtt{n}+1}(0))$. Similarly, if the object is state $\mathtt{busy}(\mathtt{s}^{\mathtt{n}}(0))$, and a thread releases the meta-lock via the slow path using a `put_slow` signal, then the number of threads waiting for the meta-lock is decremented and the object moves to state $\mathtt{busy}(\mathtt{s}^{\mathtt{n}-1}(0))$.

The race between the acquiring and the releasing threads is modeled by the hand-off process. The acquiring (releasing) thread synchronizes with the hand-off process using the `request` (`release`) signal. Through this synchronization, the releasing thread moves back to `idle` state

[6] Logic program clauses with empty body

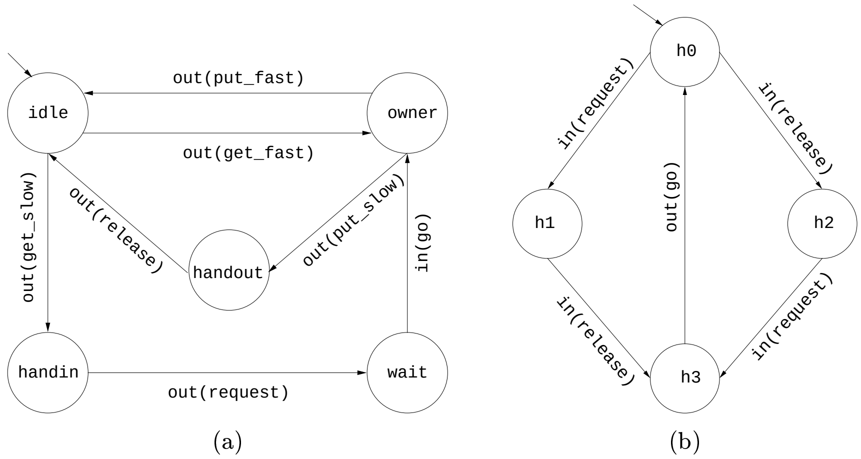

Figure 8. Labeled Transition System of (a) any thread (b) hand-off process in the Java Meta-locking algorithm

since it no longer possesses the meta-lock. The acquiring thread on the other hand waits for a go-ahead from the hand-off process. The hand-off process waits for both the `request` and the `release` signals (which may arrive in any order) after which it synchronizes with the acquiring thread using the `go` signal. Having obtained the go-ahead, the acquiring thread gets the meta-lock and moves to `owner` state.

Having modeled the thread, object and hand-off processes, the parameterized system that we need to verify is a parallel composition (in the sense of Milner's CCS [43]) of one object process, one hand-off process and ≥ 2 thread processes. Initially, the object is `not_busy`, the hand-off process is in state `h0` and all threads are `idle` (refer Figure 8).

6.2. Verifying Mutual Exclusion

We straightforwardly encoded the state representations and the transitions in the formal model of the protocol as a logic program. A global state of the protocol in our logic program encoding is a 3-tuple `(Th, Obj, H)` where `Th` is a list of thread states (of length ≥ 2) , `Obj` is a state of the object process and `H` is a state of the hand-off process. The global transition relation is straightforwardly programmed as a predicate `trans` which allows synchronization between a thread and the hand-off process, or between a thread and the object. We then verified mutual exclusion using our prover. In the following, we clarify the degree of automation in the proof.

Degree of Automation We now describe the manual work involved in the proof of mutual exclusion. The logic program description of the formal model of the protocol is constructed by hand. As in many exercises in modeling/ formal validation, constructing the formal model from the informal specification is a non-trivial task. In our case, this task took less than a week with the help of a colleague who had previously modeled it in a CCS-like language for model checking. Furthermore, recall that our prover does not strengthen invariants. So we provide a strengthening of the mutual exclusion invariant (described in the following) to the prover. Once the formal model of the protocol and the strengthened invariant are fed, the prover proves the invariant completely automatically (*i.e.* without any human intervention).

Invariant property From the labeled transition system of a thread process in Figure 8 we observe that a thread possesses the meta-lock when it is in state `owner` or `handout`. Thus, we need to prove the invariant $\mathbf{AG}$ $\neg$`bad` where `bad` is a logic program predicate which is true in all global states where ≥ 2 threads are in `owner` or `handout` states. This corresponds to the *indexed CTL* (refer [10]) formula $\mathbf{AG}$ $\oplus_i \mathtt{owner}_i \vee \mathtt{handout}_i$. The program predicate `bad` encodes the proposition $\neg(\oplus_i \mathtt{owner}_i \vee \mathtt{handout}_i)$.

Strengthening of the invariant We prove this invariant by showing transition invariance, *i.e.* by showing that none of the initial states are `bad`, and each transition preserves the invariant. Recall from Section 3.2 that this is proved by showing `bad_start` $\Rightarrow$ `false` and `bad_dest` $\Rightarrow$ `bad_src`.

The predicate `trans` encodes the global transition relation of the protocol. The encoding of the predicate `bad` is obtained by strengthening the invariant we need to prove. Note that the mutual exclusion property is expressed only in terms of the local states of the threads. However, the global state of the protocol is not simply the collection of the local states of the threads but also that of the object and the hand-off process. In the protocol description, the local states of the object and the hand-off process also bear an intended meaning in terms of contention for the meta-lock. For example, if the object is in state `not_busy` then no thread possesses the meta-lock. This is an assertion stronger than $\neg$`bad` which only requires ≤ 1 thread to posses the meta-lock. Similarly, once the hand-off process obtains the `release` signal from the releasing thread (states `h2` and `h3` in Figure 8(b)), no thread must posses the meta-lock, since the next thread has not yet been given a go-ahead. By making these two assertions explicit, we obtain the definition of predicate `bad` given below. The predicates `zero_more`, `one_more` and two_more are defined as follows: zero_more(Th), one_more(Th) and two_more(Th) are true if Th is a list of thread states and contains ≥ 0, ≥ 1 and ≥ 2 threads possesing the meta-lock respectively. Thus we prove the safety property $\mathbf{AG}\ \neg$bad where predicate bad is defined below.

```
bad((Th, not_busy, h0)) :- one_more(Th).
bad((Th, not_busy, h1)) :- one_more(Th).
bad((Th, not_busy, h2)) :- zero_more(Th).
bad((Th, not_busy, h3)) :- zero_more(Th).
bad((Th, busy(N), h0)) :- nat(N), two_more(Th).
bad((Th, busy(N), h1)) :- nat(N), two_more(Th).
bad((Th, busy(N), h2)) :- nat(N), one_more(Th).
bad((Th, busy(N), h3)) :- nat(N), one_more(Th).
```

The property $\mathbf{AG}\ \neg$bad is a strengthening of the mutual exclusion invariant which only forbids ≥ 2 threads to possess the meta-lock irrespective of the object and hand-off states. Note however that the strengthening of the invariant was performed by reasoning *only* about the local states of the object and hand-off processes. We did not need to consider interleavings of interactions between the constituent processes to strengthen the invariant. The strengthening was needed since our proof technique does not support strengthening of induction hypothesis in an induction proof. The mutual exclusion property is not preserved by every transition (even though a state violating mutual exclusion is never reached from the initial state of the protocol). This is because the mutual exclusion property only restricts the local states of the threads without taking into account the intended semantics of the local states of the other processes: object and hand-off. Therefore, we formulated a stronger invariant on the local states of all the constituent processes. However, once the strengthened invariant is fed, the proof proceeds completely automatically.

Some Statistics The timings and the number of proof steps were reported in Table I. Recall from section 3.2 that for transition invariance we need to show two predicate implications bad_start $\Rightarrow$ false and bad_src $\Rightarrow$ bad_dest. Since our proof technique supports nested induction, our prover proves 39 predicate implications (including these two) in the mutual exclusion proof of Java meta-lock. The 37 other predicate implications are automatically discovered and proved by our prover. The nesting depth of the inductive mutual exclusion proof is 3.

7. Related Work on Parameterized System Verification

Formal verification of parameterized systems has been researched widely in the last decade, and a plethora of verification techniques have been proposed. We review them below, with comparisons to our work.

Developing Decision Procedures for Subclasses: Techniques in this class [22, 23, 26, 30] reduce the problem of verifying a parameterized system to the verification of an "equivalent" finite-state system. These techniques consider specific classes of parameterized systems and temporal properties, and present fully automated verification techniques for these classes. Typically, this is achieved by establishing a "cutoff number", *i.e.* the property holds for every member of the parameterized system, if it holds for members up to the cutoff number. The only human reasoning involved is in establishing this cutoff number. Once this is established for a specific class of systems and formula, the verification problem reduces to model checking. The classes of parameterized systems for which the verification problem has been thus reduced include: (a) systems with a single control process and arbitrary number of user processes [23, 26] (b) unidirectional rings with arbitrary number of processes which communicate by passing a token [22], and (c) systems formed by composing an arbitrary number of identical finite state processes in parallel [30]. Systems covered by classes (a) and (c) commonly occur in bus-based parameterized protocols. Some of these techniques [22] establish a stuttering bisimulation equivalence (refer [10]) between a parameterized family with arbitrary number of processes and the member of the family with the cutoff number. Others [30] establish only a simulation between the parameterized family and its member with the cutoff number.

Network Invariants: The network invariant approach [14, 37, 38, 64] is applicable to parameterized systems consisting of an arbitrary number of copies of identical finite state processes (or finite state processes drawn from some finite set) that are composed in parallel. This approach relies on the existence of a pre-order relation $\preceq$ over processes, s.t. if P, P' are processes and φ is a temporal property then $P \preceq P' \wedge P' \models \varphi \Rightarrow P \models \varphi$. Furthermore, the parallel composition operator of processes is assumed to be monotonic w.r.t. $\preceq$. Under these assumptions, the network invariant approach constructs a finite state process I s.t. $P_i \preceq I$ for any member P_i of the parameterized family being verified. Verifying a temporal property φ for a parameterized family then involves:

1. synthesizing a network invariant I

2. checking $P_i \preceq I$ for any member P_i of the family
3. checking $I \models \varphi$

Step 1 involves substantial human ingenuity, and efforts have been made to partially automate this step. Step 2 proceeds by a routine induction on the "structure" of the parameterized family. Such induction proofs are amenable to automated inductive theorem proving techniques including the one developed in this paper. Finally, step 3 is accomplished by model checking. Clearly, the main difficulty in using this approach is the synthesis of the network invariant, and the need to refine the invariant if $I \not\models \varphi$. [38] tackles the second problem by viewing network invariant synthesis as least fixed point construction (which may not terminate) and using human assisted fixed point approximation heuristics such as widening (refer [16]) to compute the fixed point. Refining of the widening operator then yields a refined invariant. [14] tackles the problem of automating invariant construction by considering only those invariants which can be gleaned from the "base case" of the recursive definition of the parameterized family. In contrast, our proof technique is geared to automate proofs of temporal properties of parameterized systems where: *(i)* the proof proceeds by (nested) induction *(ii)* each of the induction arguments proceed without strengthening of induction hypothesis. There is no restriction on the existence of a network invariant which can be gleaned from the syntax of the parameterized family definition.

Rich Language Model Checking: This approach [18, 24, 25, 36] aims to *finitely* represent the state space and transition relation of the entire infinite family of finite-state systems comprising a given parameterized system. Clearly, finite representation of the global states of a parameterized system requires a richer language (than the direct term representation). For this purpose, [36] uses regular and tree-regular languages to represent global states of linear and tree networks of processes respectively. [24] constructs a covering graph which (if the construction terminates) can be used to decide safety and liveness properties of parameterized systems. The nodes of the covering graph represent a set of reachable states but the exact choice of the representation is not specified. [18, 25] use equality and inequality *constraints* on the counts of the local states (*i.e.* states of the constituent finite state processes in a network) to represent sets of global states of the network. In short, this approach requires a finite representation of infinite sets of states, and employs state space traversal techniques (just like model checking) over this finite representation to construct a proof for the parameterized network. Since, the proof is constructed by state space traversal and/or methods similar to constraint solving, these proofs are uniform proofs, rather than induction proofs. However, an induction proof over the direct term representation can (in certain cases) be achieved by a uniform proof over this finite representation. In the technique proposed in this paper, a global state of a parameterized network is represented by an unbounded term. The state space, transition relation as well as the property to be verified are still represented finitely by means of (recursively defined) logic program predicates. These predicates are then transformed to yield an induction proof over the term representation of global states. In comparison to the rich language model checking approach, our technique is not restricted to specific classes of networks based on the choice of the rich language. For example, using our technique we constructed proofs of temporal properties of cache coherence and broadcast protocols (which can be handled by the count based constraint representation of [18, 25]) as well proofs of properties of tree networks (which cannot be handled by the count based constraint representation).

Verification via Theorem Proving: The work closest to ours involves the use of inductive theorem provers for verifying parameterized systems. Some of the theorem provers which have been used for verification are: Nqthm and ACL2 [8, 9, 35], Oyster-CLAM [11, 12], PVS [45, 52, 60] and RRL [34, 33]. Nqthm and ACL2 are provers based on first-order logic, RRL and Oyster-CLAM are based on term rewriting, and PVS is based on a higher order logic. These provers have been successfully employed to construct verification proofs (including verification of complete microprocessors) by using substantial human assistance. Combining decision procedures like model checking with theorem provers is thus important for reducing the human assistance required. Rajan et. al. [52] have incorporated a finite-state model checker for the modal mu-calculus [21] as a decision procedure within the PVS theorem prover [45]. Inductive proofs can be established by the prover via calls to the model checker to verify finite subparts. Graf and Saidi [27] 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 like automated proof techniques with deductive capabilities, rather than implementing model checking as a decision procedure in a deductive system. Moreover, deductive steps are deployed only on demand and hence do not affect the efficacy of the algorithmic model checking. In particular, the underlying evaluation mechanism for model checking in the logic programming based XMC model checker [53] is essentially unfolding.

We have enhanced this mechanism with deductive transformations like folding. These transformations augment model checking with inductive reasoning.

8. Discussion

In this paper we have presented a logic program transformation framework which unifies algorithmic and deductive verification techniques for formally analyzing parameterized systems. The program transformations are designed to achieve both state space exploration which is accomplished by algorithmic techniques as well as (limited) inductive reasoning. In particular, our framework is geared to automate nested induction proofs where each of the inductive subproofs proceed without strengthening of hypothesis. This gives us a lightweight (semi)-automated theorem proving framework which is used to prove invariant properties of parameterized systems.

Development of the logic program transformation based proof technique involves development of *transformation rules* and *strategies*. The transformation rules (unfolding, folding) used in this paper were developed in [55]. In terms of transformation strategies, the paper presents a proof search framework (Figure 5) which tightly integrates algorithmic and deductive steps in a verification proof. Concrete proof search strategies can be obtained by instantiating this framework. We discussed concrete proof strategies in Section 4. Our strategies exploit the knowledge of the interconnection pattern of the parameterized network to be verified and description of the invariant property to be proved. The transformation strategies developed in this paper have been implemented to yield a prover for parameterized systems. The prover has been used to automatically verify critical properties of real-life distributed algorithms including the recently developed Java meta-locking algorithm from Sun Microsystems [1].

As future work, we plan to integrate automated invariant strengthening techniques [6] into our proof technique. Note that our proof technique corresponds to nested induction proof of an invariant property $AG\varphi$ without strengthening. If such a proof attempt fails, we need to strengthen our invariant property to $AG\varphi'$ (where $\varphi' \Rightarrow \varphi$) and re-run our transformation based proof technique for the strengthened property. Currently we are developing techniques for automating this strengthening step. In particular, the failed transformation based proof attempt of $AG\varphi$ can yield valuable information for strengthening φ.

Acknowledgements

The authors would like to thank Samik Basu for his help in modeling the Java meta-locking algorithm. Kedar Namjoshi suggested the need for comparing our technique with traditional inductive theorem proving, and provided an ACL2 proof script. Many discussions leading to this work benefitted from the feedback given by K. Narayan Kumar, C.R. Ramakrishnan and Scott A. Smolka. This work was partially supported by NSF grants CCR-9711386, CCR-9876242, EIA-9705998.

References

1. Agesen, O. et al.: 1999, 'An Efficient Meta-lock for Implementing Ubiquitous Synchronization'. In: *ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA).* Technical report available from `http://www.sun.com/research/techrep/1999/abstract-76.html`.
2. Alur, R. and T. A. Henzinger (eds.): 1996, 'Computer Aided Verification (CAV '96)', Vol. 1102 of *Lecture Notes in Computer Science.* New Brunswick, New Jersey:, Springer-Verlag.
3. Apt, K. and D. Kozen: 1986, 'Limits for automatic verification of finite-state systems'. *Information Processing Letters* **15**, 307–309.
4. Archibald, J. and J.-L. Baer: 1986, 'Cache coherence protocols: Evaluation using a multi-processor simulation model'. *ACM Transactions on Computer Systems* **4**.
5. Basu, S., S. Smolka, and O. Ward: 2000, 'Model Checking the Java Meta-Locking Algorithm'. In: *IEEE International Conference on the Engineering of Computer Based Systems.*
6. Bjorner, N., I. Browne, and Z. Manna: 1997, 'Automatic Generation of Invariants and Intermediate Assertions'. *Theoretical Computer Science* **173**(1), 49–87.
7. Bossi, A., N. Cocco, and S. Dulli: 1990, 'A Method of Specializing Logic Programs'. *ACM Transactions on Programming Languages and Systems* pp. 253–302.
8. Boyer, R. and J. Moore: 1975, 'Proving theorems about Lisp functions'. *Journal of the ACM* **22**(1), 129–144.
9. Boyer, R. and J. Moore: 1990, 'A Theorem Prover for a Computational Logic'. In: *International Conference on Automated Deduction (CADE), LNAI 449.* pp. 1–15.
10. Browne, M., E. Clarke, and O. Grumberg: 1989, 'Reasoning about networks with many identical finite-state processes'. *Information and Computation* **81**(1), 13–31.
11. Bundy, A.: 1988, 'The use of explicit plans to guide inductive proofs'. In: *International Conference on Automated Deduction (CADE).* pp. 111–120.
12. Bundy, A. et al.: 1990, 'The Oyster-CLAM system'. In: *International Conference on Automated Deduction (CADE), LNAI 449.* pp. 647–648.

13. Clarke, E., E. Emerson, and A. Sistla: 1986, 'Automatic verification of finite-state concurrent systems using temporal logic specifications'. *ACM Transactions on Programming Languages and Systems* **8**(2).
14. Clarke, E., O. Grumberg, and S. Jha: 1997, 'Verifying parameterized networks'. *ACM Transactions on Programming Languages and Systems* **19**(5).
15. Coq: 1999, 'The Coq Proof Assistant : Reference Manual'. INRIA Rocquencourt, URL `http://pauillac.inria.fr/coq/doc/main.html`, Paris, France.
16. Cousot, P. and R. Cousot: 1992, 'Comparing the Galois connection and widening/narrowing approaches to abstract interpretation'. In: *International Symposium on Programming Language Implementation and Logic Programming (PLILP), LNCS 631.*
17. Das, S.: 1992, *Deductive Databases and Logic Programming.* Addison-Wesley.
18. Delzanno, G.: 2000, 'Automatic Verification of Parameterized Cache Coherence Protocols'. In: *Computer Aided Verification (CAV), LNCS 1855.*
19. Delzanno, G. and A. Podelski: 1999, 'Model Checking in CLP'. In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Vol. LNCS 1579. pp. 74–88.
20. Dill, D. L.: 1996, 'The Murφ Verification System'. in [2], pp. 390–393.
21. Emerson, E.: 1990, *Temporal and Modal Logic*, Vol. B of *Handbook of Theoretical Computer Science*, pp. 995–1072. Elsevier/North-Holland.
22. Emerson, E. and K. Namjoshi: 1995, 'Reasoning about rings'. In: *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL).*
23. Emerson, E. and K. Namjoshi: 1996, 'Automated Verification of Parameterized Synchronous Systems'. In: *Computer Aided Verification (CAV), LNCS 1102.*
24. Emerson, E. and K. Namjoshi: 1998, 'On Model Checking for Non-Deterministic Infinite State Systems'. In: *IEEE Symposium on Logic in Computer Science (LICS).*
25. Esparza, J., A. Finkel, and R. Mayr: 1999, 'On the Verification of Broadcast Protocols'. In: *IEEE Symposium on Logic in Computer Science (LICS).*
26. German, S. and A. Sistla: 1992, 'Reasoning about systems with many processes'. *Journal of the ACM* **39**, 675–735.
27. Graf, S. and H. Saidi: 1996, 'Verifying invariants using theorem proving'. in [2].
28. Holzmann, G. J.: 1997, 'The Model Checker SPIN'. *IEEE Transactions on Software Engineering* **23**(5), 279–295.
29. Hsiang, J. and M. Srivas: 1987, 'Automatic inductive theorem proving using Prolog'. *Theoretical Computer Science* **54**, 3–28.
30. Ip, C. N. and D. L. Dill: 1999, 'Verifying Systems with Replicated Components in Murφ'. *Formal Methods in System Design* **14**(3).
31. Jonsson, B. and M. Nilsson: 2000, 'Transitive closures of regular relations for verifying infinite-state systems'. In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 1785.*
32. Kanamori, T. and H. Fujita: 1986, 'Formulation of Induction Formulas in Verification of Prolog Programs'. In: *International Conference on Automated Deduction (CADE).* pp. 281–299.
33. Kapur, D. and M. Subramanium: 1998, 'Mechanical Verification of Adder Circuits using Rewrite Rule Laboratory'. *Formal Methods in System Design* **13**(2), 127–158.

34. Kapur, D. and H. Zhang: 1989, 'An overview of rewrite rule laboratory (RRL)'. In: *International Conference on Rewriting Techniques and Applications (RTA)*. pp. 559–563.
35. Kaufmann, M., P. Manolis, and J. Moore: 2000, *Computer-Aided Reasoning: An approach*. Kluwer Academic.
36. Kesten, Y., O. Maler, M. Marcus, A. Pnueli, and E. Shahar: 1997, 'Symbolic model checking with rich assertional languages'. In: *Computer Aided Verification (CAV), LNCS 1254*.
37. Kurshan, R. and K. Mcmillan: 1995, 'A structural induction theorem for processes'. *Information and Computation* **117**, 1–11.
38. Lesens, D., N. Halbwachs, and P. Raymond: 1997, 'Automatic Verification of Parameterized Linear Networks of Processes'. In: *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*. pp. 346–357.
39. Leuschel, M., D. D. Schreye, and A. D. Waal: 1996, 'A Conceptual Embedding of Folding into Partial Deduction : Towards a Maximal Integration'. In: *Joint International Conference and Symposium on Logic Programming*. pp. 319–332.
40. Lichtenstein, O. and A. Pnueli: 1985, 'Checking that finite state concurrent programs satisfy their linear specification'. In: *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*.
41. Lloyd, J.: 1993, *Foundations of Logic Programming*. Springer-Verlag.
42. Manna, Z. and A. Pnueli: 1995, *The Temporal Logic of Reactive and Concurrent Systems- Safety*. Springer Verlag.
43. Milner, R.: 1989, *Communication and Concurrency*, International Series in Computer Science. Prentice Hall.
44. Nilsson, U. and J. Lubcke: 2000, 'Constraint logic programming for local and symbolic model checking'. In: *Computational Logic, LNCS 1861*.
45. Owre, S., N. Shankar, and J. Rushby: 1992, 'PVS: A Prototype Verification System'. In: *International Conference on Automated Deduction (CADE)*.
46. Pettorossi, A. and M. Proietti: 1998, *Transformation of logic programs*, Vol. 5 of *Handbook of Logic in Artificial Intelligence*, pp. 697–787. Oxford University Press.
47. Pettorossi, A. and M. Proietti: 1999, 'Synthesis and transformation of logic programs using unfold/fold proofs'. *Journal of Logic Programming* **41**(2–3), 197–230.
48. Pettorossi, A., M. Proietti, and S. Renault: 1997, 'Reducing Nondeterminism while Specializing Logic Programs'. In: *ACM SIGPLAN International Conference on Principles of Programming Languages (POPL)*. pp. 414–427.
49. Pnueli, A., S. Ruah, and L. Zuck: 2001, 'Automatic Deductive Verification with Invisible Invariants'. In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 2031*.
50. Pong, F. and M. Dubois: 1995, 'A new approach for the verification of cache coherence protocols'. *IEEE Transacations on Parallel and Distributed Systems* **6**(8).
51. Queille, J. and J. Sifakis: 1982, 'Specification and Verification of Concurrent Programs in CESAR'. In: *International Symposium on Programming, LNCS 137*.
52. Rajan, S., N. Shankar, and M. K. Srivas: 1995, 'An Integration of Model Checking with Automated Proof Checking'. In: P. Wolper (ed.): *Computer Aided Verification (CAV '95)*, Vol. 939 of *Lecture Notes in Computer Science*. Liége, Belgium, pp. 84–97.

53. Ramakrishna, Y. S., C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. L. Swift, and D. S. Warren: 1997, 'Efficient Model Checking Using Tabled Resolution'. In: *Computer Aided Verification (CAV), LNCS 1254*.
54. Roychoudhury, A.: 2000, 'Program Transformations for Verifying Parameterized Systems'. Ph.D. thesis, State University of New York at Stony Brook, Available from `http://www.comp.nus.edu.sg/~abhik/papers.html`.
55. Roychoudhury, A., K. N. Kumar, C. Ramakrishnan, and I. Ramakrishnan: 1999, 'A parameterized unfold/fold transformation framework for definite logic programs'. In: *International Conference on Principles and Practice of Declarative Programming (PPDP), LNCS 1702*. pp. 396–413.
56. Roychoudhury, A., K. N. Kumar, C. R. Ramakrishnan, I. Ramakrishnan, and S. A. Smolka: 2000, 'Verification of Parameterized Systems using Logic Program Transformations'. In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS), LNCS 1785*. pp. 172–187.
57. Roychoudhury, A., K. N. Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan: 2002, 'Beyond Tamaki-Sato style Unfold/fold Transformations for Normal Logic Programs'. *International Journal on Foundations of Computer Science* **13**(3), 387–403.
58. Roychoudhury, A. and I. Ramakrishnan: 2001, 'Automated Inductive Verification of Parameterized Protocols'. In: *International Conference on Computer Aided Verification (CAV), LNCS 2102*. pp. 25–37.
59. Sato, T.: 1992, 'Equivalence-preserving first-order unfold/fold transformation systems'. *Theoretical Computer Science* **105**, 57–84.
60. Shankar, N.: 1996, 'PVS: Combining Specification, Proof Checking, and Model Checking'. In: *International Conference on Formal Methods in Computer-Aided Design (FMCAD), LNCS 1166*. pp. 257–264.
61. Tamaki, H. and T. Sato: 1984, 'Unfold/fold transformations of logic programs'. In: *Proceedings of International Conference on Logic Programming*. pp. 127–138.
62. Urbina, L.: 1996, 'Analysis of Hybrid Systems in CLP(R)'. In: *Constraint Programming (CP'96)*, Vol. LNCS 1102.
63. Urbina, L. and G. Riedewald: 1995, 'Simulation and Verification in Constraint Logic Programming'. In: *2nd European Workshop on Real Time and Hybrid Systems*.
64. Wolper, P. and V. Lovinfosse: 1989, 'Verifying properties of large sets of processes with network invariants'. In: *LNCS 407*.
65. XSB: 2000, 'The XSB Logic Programming System v2.2'. Available for downloading from `http://xsb.sourceforge.net/`.